Progress In Electromagnetics Research M, Vol. 23, 29–51, 2012

GRAPHICS PROCESSOR UNIT (GPU) ACCELERATION OF FINITE-DIFFERENCE FREQUENCY-DOMAIN (FDFD) METHOD

V. Demir*

Northern Illinois University, Department of Electrical Engineering, EB352 DeKalb, IL 60115, USA

Abstract—Recently, many numerical methods that are developed for the solution of electromagnetic problems have greatly benefited from the hardware accelerated scientific computing capability provided by graphics processing units (GPUs) and orders of magnitude speed-up factors have been reported. Among these methods, the finite-difference frequency-domain (FDFD) method as well can be accelerated substantially by utilizing an efficient algorithm customized for GPU computing. In this contribution, an algorithm is presented that treats iterative solution of the FDFD linear equation system similar to solution of three-dimensional Finite-Difference Time-Domain (FDTD) method, which inherently yields itself to high level parallelization. The presented algorithm uses BICGSTAB iterative solver. Integrated with BICGSTAB, an efficient method of performing matrix-vector products for the linear system of FDFD equations is adapted and implemented in Compute Unified Device Architecture (CUDA). It is shown that FDFD can be solved with a speed-up factor of more than 20 on a GPU compared with the solution on a central processing unit (CPU), while memory usage as well can be reduced substantially with the presented algorithm.

1. INTRODUCTION

Recently, it has been realized that graphics processing units (GPUs) can be used not only for acceleration of graphics computations, but acceleration of scientific computations as well. Though GPU processors are built to perform simpler tasks compared with central processing unit (CPU) processors, having a large number of GPU processors on

Received 9 September 2011, Accepted 16 December 2011, Scheduled 8 January 2012

* Corresponding author: Veysel Demir (vdemir@niu.edu).

a graphics card gives it a huge computation power. If an algorithm involves data parallel computations, i.e., the same instructions are applied to a large set of data, the algorithm can be implemented to run faster on a GPU card. Graphics card vendors have been improving their hardware architectures to further address the needs of scientific computations, thus promoting GPU based computation cards as the new-generation computation platforms.

Developments in GPU hardware have been accompanied by introduction of software platforms to facilitate programming for GPU cards. OpenGL, Brook, and High Level Shader Language (HLSL) were the programming languages available previously to develop codes to run on GPUs; however, they are outdated by the introduction of new software development environments such as Compute Unified Device Architecture (CUDA), Open Computing Language (OpenCL), and DirectCompute. These new languages are developed to facilitate the use of new-generation graphics cards and let the programmers focus on the task of parallelization of their algorithms rather than spending time on dealing with the intrinsics of underlying GPU hardware.

Recently, introduction of CUDA by Nvidia caused a major shift toward use of GPU cards in scientific computing. CUDA can be run only on CUDA enabled Nvidia cards, which is a major disadvantage, however, extensive support from Nvidia and accumulated knowledge within the programming community is expected to keep CUDA as one of the main programming platforms in the near future besides the other more general application programming interfaces (APIs) such as OpenCL.

CUDA has been extensively used to develop GPU accelerated electromagnetic simulation codes. Finite-difference time-domain (FDTD) method is one of the methods that has benefited from GPU computing the most since it is a data parallel algorithm. While implementations of FDTD using CUDA have been reported in several publications, [1–4] are among those that discuss details on development of efficient FDTD codes and can be used as guidelines by developers. Similarly, MoM and FEM also have been implemented to run on GPU and several reports have been published to demonstrate effectiveness of GPU computations of these methods. All these publications report significant speed-up factors. For instance, [5] reports a speed-up factor of about 20 in FEM, [6] reports a speed-up factor of 17 in MoM, while [3] reports a factor of about 30 in FDTD, when GPU codes are compared with CPU codes. It should also be noted that these numbers of speed-up factors may not be directly comparable with each other since several factors, such as types of problems involved, types of GPU and CPU architectures, single or double precision computations,

etc., will affect the speed-up factors. However, it can be claimed that the FDTD method has been the method benefiting from GPU computations the most since it inherently yields itself to parallelization, unlike other frequency-domain methods which require solution of large-scale matrix equations that require more elaborate work to solve using parallel processing.

If transient results or wideband results are sought in the solution of an electromagnetic problem, a time-domain method, such as FDTD, can be used. Time domain methods may not be preferable for some types of problems. For example, time-domain methods are computationally expensive for modeling dispersive materials, or computations take very long time for highly resonant structures. If results at a single frequency or a small number of frequencies is sought, a frequency domain method may be preferred depending on the type of the problem geometry. For instance, finite-difference frequency-domain (FDFD) might be preferred for a highly inhomogeneous, dispersive, or highly resonant geometry.

FDFD has been receiving attention as an alternative electromagnetic solution method for various types of problems. Besides being used for solution of regular problems, for instance, FDFD has been used as the solver in iterative multi region (IMR) for the solution of large-scale problems in [7–9], while it has been used to model chiral materials in [10, 11]. FDFD requires iterative solution of a very large sparse linear equation system, which is usually very inefficient. Solution of FDFD as well can significantly benefit from GPU acceleration.

So far, the only contribution published in the literature on GPU acceleration of FDFD equations is the one by Zainud-Deen et al. [12], where Brook is used to implement the FDFD code. While solving large sparse matrix equations, customarily only the non-zero coefficients of a matrix would be stored along with the coordinates of the coefficients in the matrix. In [12], a similar coordinate format storage scheme is used, and the respective calculations are based on this storage scheme.

It should be noted that a good choice of the storage scheme is an important factor for achieving an efficient solution. For instance, [13] presents the effective use of two matrix storage schemes, Compressed Sparse Row (CSR) and Hybrid (HYB) Ellpack Coordinate formats, for the solution of MoM equations on GPU using CUDA.

It has been shown in [14] that the coefficients that describe FDFD equations can be stored in three-dimensional arrays, similar to the updating coefficients of the FDTD method, and an algorithm based on this scheme can improve the efficiency of FDFD solution both in terms of computation time and memory usage. Moreover, this scheme presented in [14] inherently lends itself to high level of parallelism,

similar to FDTD method, on a GPU architecture. In the current contribution, we show that the speed of FDFD calculations can further be improved on a GPU by using the algorithm presented in [14].

Section 2 presents the FDFD formulation considered in this contribution. Section 3 summarizes the algorithm that is presented in [14] and adopted here for CUDA implementation. Section 4 presents the CUDA implementation of iterative FDFD solution. Section 5 illustrates the speed-up factors achieved by the presented algorithm.

2. FDFD FORMULATION

FDFD formulation is based on Maxwell's curl equations expressed in frequency domain

$$\nabla \times \bar{E}_{\text{total}} = -j\omega\mu\bar{H}_{\text{total}}, \tag{1}$$

$$\nabla \times \bar{H}_{\text{total}} = j\omega\epsilon\bar{E}_{\text{total}}, \tag{2}$$

where $\bar{E}_{\text{total}}$ and $\bar{H}_{\text{total}}$ are the electric and magnetic fields, and ϵ and μ are permittivity and permeability parameters. The time harmonic convention used in (1) and (2) is $e^{j\omega t}$. Scattered field formulation [15] can be used for scattering problems, where the total fields ($\bar{E}_{\text{total}}$ and $\bar{H}_{\text{total}}$) are sum of incident fields ($\bar{E}_{\text{inc}}$ and $\bar{H}_{\text{inc}}$) and scattered fields ($\bar{E}_{\text{scat}}$ and $\bar{H}_{\text{scat}}$). Incident fields are the fields that propagate in free space in which no scatterers exist, thus they satisfy Maxwell's equations in free space, and they excite the problem space in consideration. After some manipulations (1) and (2) can be written in terms of incident and scattered fields as

$$\bar{H}_{\text{scat}} = -\frac{1}{j\omega\mu}\nabla \times \bar{E}_{\text{scat}} + \frac{\mu_0 - \mu}{\mu}\bar{H}_{\text{inc}}, \tag{3}$$

$$\bar{E}_{\text{scat}} = \frac{1}{j\omega\epsilon}\nabla \times \bar{H}_{\text{scat}} - \frac{\epsilon_0 - \epsilon}{\epsilon}\bar{E}_{\text{inc}}, \tag{4}$$

where ϵ_0 and μ_0 are free space permittivity and permeability parameters. Equations (3) and (4) are vector equations, and they can be expressed in terms of six scalar equations imposed on Cartesian coordinate system. For instance, one of these equations is

$$H_{\text{scat},x} = \frac{1}{j\omega\mu_x}\frac{\partial E_{\text{scat},y}}{\partial z} - \frac{1}{j\omega\mu_x}\frac{\partial E_{\text{scat},z}}{\partial y} + \frac{\mu_0 - \mu_x}{\mu_x}H_{\text{inc},x}, \tag{5}$$

At this point, these equations can be slightly modified to account for perfectly matched layer (PML) [16] absorbing boundary conditions as shown in [7] or [12]. Since only the material parameters ϵ and μ are slightly modified, the form of equations will stay the same. Therefore, to keep the following discussions simple, the PML-related modifications will be omitted in the following equations.

The partial differential equations (PDEs) as (5) can be imposed on a grid composed of Yee cells [17] on which electric and magnetic field components are specified at discrete spatial positions. Yee cells are used as the basis of FDTD method as well, therefore, the same type of geometrical modeling can be done in FDFD as in FDTD. As will be discussed later, the same type of spatial dependence of fields between FDTD and FDFD will allow us to develop a very similar type of GPU acceleration code for FDFD as in FDTD.

Expressing equations as (5) on a discrete Yee grid using central difference approximation to partial derivatives yields

$$\begin{aligned} &H_{\text{scat},x}(i,j,k) - \frac{1}{j\omega\mu_x(i,j,k)\Delta z}E_{\text{scat},y}(i,j,k+1) \\ &+\frac{1}{j\omega\mu_x(i,j,k)\Delta z}E_{\text{scat},y}(i,j,k) + \frac{1}{j\omega\mu_x(i,j,k)\Delta y}E_{\text{scat},z}(i,j+1,k) \\ &-\frac{1}{j\omega\mu_x(i,j,k)\Delta y}E_{\text{scat},z}(i,j,k) = \frac{\mu_0-\mu_x(i,j,k)}{\mu_x(i,j,k)}H_{\text{inc},x}(i,j,k), \end{aligned} \tag{6}$$

$$\begin{aligned} &H_{\text{scat},y}(i,j,k) - \frac{1}{j\omega\mu_y(i,j,k)\Delta x}E_{\text{scat},z}(i+1,j,k) \\ &+\frac{1}{j\omega\mu_y(i,j,k)\Delta x}E_{\text{scat},z}(i,j,k) + \frac{1}{j\omega\mu_y(i,j,k)\Delta z}E_{\text{scat},x}(i,j,k+1) \\ &-\frac{1}{j\omega\mu_y(i,j,k)\Delta z}E_{\text{scat},x}(i,j,k) = \frac{\mu_0-\mu_y(i,j,k)}{\mu_y(i,j,k)}H_{\text{inc},y}(i,j,k), \end{aligned} \tag{7}$$

$$\begin{aligned} &H_{\text{scat},z}(i,j,k) - \frac{1}{j\omega\mu_z(i,j,k)\Delta y}E_{\text{scat},x}(i,j+1,k) \\ &+\frac{1}{j\omega\mu_z(i,j,k)\Delta y}E_{\text{scat},x}(i,j,k) + \frac{1}{j\omega\mu_z(i,j,k)\Delta x}E_{\text{scat},y}(i+1,j,k) \\ &-\frac{1}{j\omega\mu_z(i,j,k)\Delta x}E_{\text{scat},y}(i,j,k) = \frac{\mu_0-\mu_z(i,j,k)}{\mu_z(i,j,k)}H_{\text{inc},z}(i,j,k), \end{aligned} \tag{8}$$

$$\begin{aligned} &E_{\text{scat},x}(i,j,k) + \frac{1}{j\omega\epsilon_x(i,j,k)\Delta z}H_{\text{scat},y}(i,j,k) \\ &-\frac{1}{j\omega\epsilon_x(i,j,k)\Delta z}H_{\text{scat},y}(i,j,k-1) - \frac{1}{j\omega\epsilon_x(i,j,k)\Delta y}H_{\text{scat},z}(i,j,k) \\ &+\frac{1}{j\omega\epsilon_x(i,j,k)\Delta y}H_{\text{scat},z}(i,j-1,k) = \frac{\epsilon_0-\epsilon_x(i,j,k)}{\epsilon_x(i,j,k)}E_{\text{inc},x}(i,j,k), \end{aligned} \tag{9}$$

$$E_{\text{scat},y}(i,j,k) + \frac{1}{j\omega\epsilon_y(i,j,k)\Delta x} H_{\text{scat},z}(i,j,k)$$
$$- \frac{1}{j\omega\epsilon_y(i,j,k)\Delta x} H_{\text{scat},z}(i-1,j,k) - \frac{1}{j\omega\epsilon_y(i,j,k)\Delta z} H_{\text{scat},x}(i,j,k)$$
$$+ \frac{1}{j\omega\epsilon_y(i,j,k)\Delta z} H_{\text{scat},x}(i,j,k-1) = \frac{\epsilon_0 - \epsilon_y(i,j,k)}{\epsilon_y(i,j,k)} E_{\text{inc},y}(i,j,k), \quad (10)$$

$$E_{\text{scat},z}(i,j,k) + \frac{1}{j\omega\epsilon_z(i,j,k)\Delta y} H_{\text{scat},x}(i,j,k)$$
$$- \frac{1}{j\omega\epsilon_z(i,j,k)\Delta y} H_{\text{scat},x}(i,j-1,k) - \frac{1}{j\omega\epsilon_z(i,j,k)\Delta x} H_{\text{scat},y}(i,j,k)$$
$$+ \frac{1}{j\omega\epsilon_z(i,j,k)\Delta x} H_{\text{scat},y}(i-1,j,k) = \frac{\epsilon_0 - \epsilon_z(i,j,k)}{\epsilon_z(i,j,k)} E_{\text{inc},z}(i,j,k). \quad (11)$$

In (6)–(11) scattered field components on the left-hand sides are the unknowns to be computed, while the right-hand sides include the excitations. At this point, (6)–(11) can be used to construct a linear set of equations, such as

$$Ax = y, \quad (12)$$

where A is a coefficient matrix, x is a vector including all scattered electric and magnetic field components, while y is the excitation vector. If a three-dimensional computational domain is composed of, for instance, N cells, the vectors x and y will be of size $6N$, while matrix A will be of size $6N \times 6N$, since both the electric and magnetic field components are the unknowns.

It is possible to obtain $H_{\text{scat},x}$, $H_{\text{scat},y}$, and $H_{\text{scat},z}$ from (6)–(8) and use them in (9)–(11) to obtain three equations in which $E_{\text{scat},x}$, $E_{\text{scat},y}$, and $E_{\text{scat},z}$ are the only unknowns. Such equations are shown in [7] (Equations (2.27), (2.28), and (2.29)). Using these reduced equations, a matrix equation as (12) can be obtained, where the vectors x and y will be of size $3N$, while matrix A will be of size $3N \times 3N$ for a computational domain of N cells.

Here it should be noted that the coefficient matrix A is highly sparse and it includes only 13 nonzero coefficients per row as shown in [7]. Since A is highly sparse, it is sufficient to store only its nonzero components on computer memory while processing it in a program. Special storage schemes are used to store such sparse matrices, details of which are discussed in [18]. For instance, [7] uses a scheme referred to as *coordinate format*, in which the data structure consists of three arrays: (1) an array containing all the complex values of the nonzero

elements of A; (2) an integer array containing their row indices; and (3) a second integer array containing their column indices. All three arrays are of length $39N$. It should be noted that [12] uses the coordinate format storage scheme with two arrays: (1) an array containing the values of the nonzero elements; and (2) an integer array containing their column indices. Each of these arrays are of size $39N$. Since it is already known that each row contains 13 elements, there is no need to store row indices. Although two other sparse storage schemes, named as compressed sparse row (CSR) format and modified sparse row (MSR) format, are available and slightly more efficient, the memory requirement is still very high for the storage of A. We discuss a storage scheme presented in [14] that stores the coefficients in three-dimensional arrays, which leads to improved efficiency in computation time and memory usage in the next section.

3. AN ALGORITHM FOR EFFICIENT SOLUTION OF FDFD

The solution of large sparse systems is very costly, if not impossible, in terms of computer time and memory if direct linear system solution methods (i.e., Gaussian elimination, LU decomposition, etc.) are used. These large systems can often be solved only by using iterative methods. There are several iterative techniques proposed for solving linear systems [18, 19]. Among these techniques, Generalized Minimal Residual (GMRES) method [20] and Biconjugate Gradients Stabilized (BICGSTAB) [21, 22] method are the most commonly used for numerical solution of Maxwell's equations [14].

It is widely recognized that preconditioning is the most critical ingredient in the development of efficient solvers for challenging problems in scientific computation, and that the importance of preconditioning is destined to increase even further [30]. A well-chosen preconditioner can significantly improve the efficiency of an iterative solver. The current contribution addresses the GPU acceleration of FDFD, where the BICGSTAB algorithm is utilized. One should consider finding and using an efficient preconditioner as well to further accelerate the computations both on the CPU and GPU platforms.

An iterative solver starts with an initial guess x_0 and minimizes the residual $r = y - Ax_k = y - y'$ as the iterations proceed, where x_k is the solution at the kth iteration. As the residual minimizes the x_k converges to the solution x. This process requires a multiplication of A by x_k to produce the next residual [14]. The matrix-vector product Ax_k is the most time consuming stage in the iterative procedure. As discussed in Section 2, generally one of the sparse matrix storage

schemes is used to store A and the operation Ax_k is performed in a function, usually referred to as matvec, that is based on the employed scheme. However, it is not necessary to employ one of these storage schemes mentioned above; one can develop an alternative storage scheme and develop an algorithm based on this new scheme. Then a matvec function that computes $y' = Ax_k$ based on this new algorithm and returns the result to the iterative solver can be implemented. Such an algorithm to improve the efficiency of Ax_k stage is proposed in [14]. A summary of this algorithm is provided next.

3.1. Storage of Coefficients in Three-Dimensional Arrays

One can notice that (6)–(8) can be cast in a form as

$$x_e + A_e x_h = y_e, \tag{13}$$

whereas (9)–(11) can be given as

$$x_h + A_h x_e = y_h, \tag{14}$$

where A_e and A_h are coefficient matrices, x_e and x_h are vectors of electric and magnetic field components, respectively, and y_e and y_h are excitation vectors. One can use x_h from (14) in (13) and obtain

$$x_e - A_e A_h x_e = y_e - A_e y_h, \tag{15}$$

which is equivalent to (12). For a given x_k, $y' = Ax_k$ can be performed in multiple steps using (15) such that

$$x_t = A_h x_k, \quad x_t = A_e x_t, \quad y' = x_k - x_t, \tag{16}$$

where x_t is a vector used to store intermediate results. The advantage of this scheme is that instead of using a matrix A with $39N$ nonzero coefficients, we use two matrices, A_e and A_h, each with $12N$ (3 fields $\times$ 4 nonzero coefficients per row $\times N$ cells) nonzero coefficients. This implies a reduction in storage of coefficients from $39N$ to $24N$. Examining (6)–(11) one can notice that we have coefficient pairs in which the pairs are different only by their signs. For instance, (6) can be expressed as

$$\begin{aligned} &H_{\text{scat},x}(i,j,k) - C_{hxey}(i,j,k)(E_{\text{scat},y}(i,j,k+1) - E_{\text{scat},y}(i,j,k)) \\ &+C_{hxez}(i,j,k)(E_{\text{scat},z}(i,j+1,k) - E_{\text{scat},z}(i,j,k)) \\ &= \frac{\mu_0 - \mu_x(i,j,k)}{\mu_x(i,j,k)} H_{\text{inc},x}(i,j,k), \end{aligned} \tag{17}$$

which implies that the coefficient storage requirements can further be reduced by half, and we need to store only $12N$ coefficients. This is done by storing the coefficients, such as $C_{hxey}(i,j,k)$ and $C_{hxez}(i,j,k)$, as three-dimensional arrays each with size $N_x \times N_y \times N_z$, where N_x, N_y, and N_z are the numbers of cells in x, y, and z directions, respectively. Thus, the total number of cells becomes $N = N_x \times N_y \times N_z$. This way of coefficient storage also eliminates the need for integer arrays that contain row and column indices discussed in Section 2. It has been shown in [14] that the required amount of computer memory to store coefficients can be reduced by 80%. To comply with the presented scheme, the unknown vector x is stored as well as a three-dimensional array with size $N_x \times N_y \times 3N_z$. The dimension in the z direction is 3 times N_z, since first section stores $E_{\text{scat},x}$, second section stores $E_{\text{scat},y}$, and the third section stores $E_{\text{scat},z}$. Fig. 1 shows this storage scheme and how field indices in three-dimensional space relate to field indices in the x vector.

Another advantage with this scheme is that, when storing the parameters in arrays we can directly do array operations, i.e., element-wise addition, multiplication, etc., which can be easily optimized for speed by compilers. It is shown in [14] that FDFD solution can be achieved 30% faster with the presented algorithm. The presented storage scheme will be referred to as *coefficient arrays storage* in the subsequent sections.

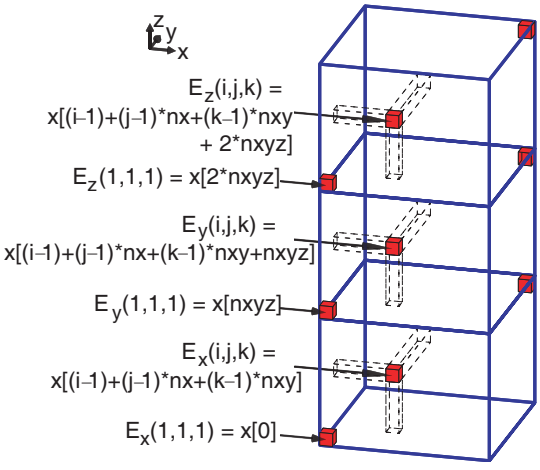

Figure 1. Field indices mapping from three-dimensional space to x vector. All fields are scattered electric fields. Domain size is $N = Nx \times Ny \times Nz$ cells. Here $nxy = Nx \times Ny$ and $nxyz = Nx \times Ny \times Nz$.

Presented algorithm can particularly benefit for efficiency of computations on GPU for two main reasons. (1) Storing the data as three-dimensional arrays naturally represents a three-dimensional space and converts FDFD to a data-parallel algorithm. (2) As will be discussed later, one major bottleneck in efficiency of computations on a GPU is data transfers from and to GPU global memory. Using less number of coefficients means less data transfer from the GPU global memory, and higher efficiency.

3.2. Coefficient Indices Storage Scheme

In this section, we present another improvement on the coefficient arrays storage scheme that utilizes the additional cached memory spaces available on CUDA enabled GPU and further reduces the data storage on GPU global memory and data transfers from the GPU memory. This scheme further reduces the memory requirements and speeds up the calculations.

The coefficient arrays storage scheme can handle solution of highly inhomogeneous problems even in which every single coefficient associated with the field components in the Yee grid is different from the others in value. If there are a small number of unique values of coefficients, it is better to store these unique values in a separate small size coefficient array, and store the indices of these coefficients in indices arrays associated with field components (or in other terms the edge components) in the Yee grid.

Since two coefficient values are needed per field component, as in (17), unique combination of coefficient pairs needs to be stored instead of single coefficient values. Then the indices to these coefficient pairs need to be stored in 6 arrays, each with size N, where each index is type 'unsigned short' of size 2 bytes. The total storage requirement for indices becomes $12N$ bytes. This storage scheme is refereed to as *coefficient indices storage* in this contribution.

In the coefficient arrays storage scheme, we need to store $12N$ coefficients, where each coefficient is type 'cuDoubleComplex' of size 16 bytes. These numbers indicate that coefficient indices storage scheme requires 1/16 the memory space compared with the coefficient arrays storage scheme. The smaller size indices arrays lead to less data transfer from the GPU global memory during the computations on GPU, thus lead to higher efficiency. This kind of coefficient indices storage scheme is often employed in CUDA implementations of FDTD as well.

The next section presents the CUDA implementation of the FDFD solution based on, first, the coefficient arrays and, second, the coefficient indices storage schemes.

4. IMPLEMENTATION OF FDFD USING CUDA

4.1. CUDA Concepts

In this section, a brief description of some concepts in CUDA is summarized from [23] in order to prepare the reader for the discussions that follow.

To program using CUDA the programmer defines C functions, called *kernels*, that, when called, are executed N times in parallel by N different CUDA *threads*. Each of the threads that executes a kernel is given a unique thread ID that is accessible within the kernel through the built-in threadIdx variable. In CUDA, a number of threads form a *thread block*. A kernel function can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks. Then a number of thread blocks are organized to form a *grid* of thread blocks. Each block within the grid can be identified by an index accessible within the kernel through the built-in blockIdx variable. The dimension of the thread block is accessible within the kernel through the built-in blockDim variable.

CUDA threads may access data from multiple memory spaces during their execution. Each thread has a private local memory and a *shared memory* visible to all threads of the block and with the same lifetime as the block. Finally, all threads have access to the same *global memory*. Global memory is the main memory space on the device to store the application data. However, data access to global memory is very slow and that inefficiency becomes the main bottleneck in the execution of a kernel. On the other hand, the shared memory is much faster to access but the size of the shared memory is very limited. However, though very limited in size, the shared memory can provide the means for data reuse and improve the efficiency of a kernel.

4.2. Thread to Cell Mapping

The presented FDFD algorithm in Section 3 uses three-dimensional arrays that provide direct mapping between the coefficients that are used to calculate field components and the cells in which these field components exist. This scheme maximizes data parallelization and lets one-to-one mapping between cells and the CUDA threads that process the data in these cells. One can construct a grid of thread blocks in which each block consists of a number of threads such that the total number of cells is equal to the total number of threads in the grid. However, a different approach is followed in this contribution: instead of mapping threads in a grid to cells in a three-dimensional

computational domain, threads in a grid are mapped to cells in an xy plane cut of the computational domain. Then, each thread is used to process all other cells in the same column by iterating the cells in the z direction, thus the entire FDFD domain is covered. As will be illustrated later, this algorithm helps for global memory reuse, which improves efficiency. This scheme of thread-to-cell mapping is used and illustrated in [2] for a CUDA implementation of FDTD method, and it is referred to as xy-mapping.

4.3. Coalesced Global Memory Access

In CUDA, when accessing global memory, there are 400 to 600 clock cycles of memory latency [23]. If a code is dominated by memory accesses rather than arithmetic instructions the memory access inefficiency becomes the bottle-neck for the efficiency of a program on GPU. Global memory bandwidth is used most efficiently when the simultaneous memory accesses by threads in a half-warp (during the execution of a single read or write instruction) can be coalesced into a single memory transaction of 32, 64, or 128 bytes [23].

The three-dimensional field and coefficient arrays are actually stored as one-dimensional arrays on computer memory, and they are accessed as one-dimensional arrays in kernel functions. It should be noted that the base code of FDFD considered in this contribution is implemented in FORTRAN. In FORTRAN, the first array index varies most rapidly in multi-dimensional arrays, i.e., for an array of $A\,(i,\ j,\ k)$, i index varies most rapidly, then j, and then k, as illustrated in Fig. 1. This ordering is retained after the arrays are transferred to GPU. The size of the three-dimensional arrays is the same as the size of the FDFD domain in number of cells. If the size of the FDFD domain in the x and y directions is multiple of 16, then the coalesced memory access is ensured [2]. If a problem space is composed of $N_x \times N_y \times N_z$ cells and N_x or N_y is not a multiple of 16, then the computational domain is enlarged in the respective direction to have the number of cells as a multiple of 16. Since the problem space is terminated by PML, expansion in domain size will only move the PML away from the scattering objects in the problem space, which will not affect the solution.

Once it is ensured that N_x and N_y are multiples of 16, then thread blocks of size 256 can be used to map threads in a grid to cells in an xy plane cut using the xy mapping scheme.

4.4. BICGSTAB Using CUDA

The iterative solver used to solve the presented FDFD equations is the "vanilla" version of BICGSTAB [22]. A FORTRAN implementation

of BICGSTAB, developed by Botchev and Fokkema, is obtained from the authors' website [24]. This FORTRAN code is first converted to a C code so that it can be modified and transformed into a CUDA implementation. The BICGSTAB code includes several array instructions, i.e., element-wise multiplication and addition of arrays. These operations are the time-consuming parts of the BICGSTAB, so they are ported to run on GPU using CUDA. To facilitate this all arrays, including coefficient, field, and temporary arrays, are copied to GPU global memory. Many of these array instructions are available in Basic Linear Algebra Subprograms (BLAS) [25] libraries. At this point, cuBlas, a BLAS library ported to CUDA, is utilized and cuBlas functions are called in the developed C version of the BICGSTAB where applicable.

4.5. Matrix-vector Product Using CUDA: Coefficient Arrays Storage Scheme

As mentioned earlier, calculation of $y' = Ax_k$ is the stage which takes the most significant computation time during the iterative solution. Kernel functions based on the algorithm presented in Section 3 are developed to speed-up the calculations on GPU. Listing 1 shows a C function that initializes and launches these kernel functions. Here the arrays that reside on GPU global memory are indicated with a dv prefix. The function cuda_matvec basically calculates y for a given x. Here, the pointers to the arrays that reside on GPU global memory are indicated with a dv prefix. The pointers to coefficient arrays are indicated with a dvC prefix. The first three kernels perform the first line of (16), whereas the following three kernels perform the second and third lines of (16). In other terms, the first three kernels perform operations associated with (6)–(8), respectively, while the following three kernels perform operations associated with (9)–(11).

Listing 2 shows the details of the second kernel, which is associated with the operation in (7) based on the coefficient arrays storage scheme, where all three-dimensional coefficient arrays are copied to the global memory before the iterative solution is performed. Due to (16), the required operation of (7) translates to

$$\begin{aligned}\mathrm{tmpy}(i,j,k) = {} & C_{\mathrm{hyez}}(i,j,k)(-E_{\mathrm{scat},z}(i+1,j,k) + E_{\mathrm{scat},z}(i,j,k)) \\ & + C_{\mathrm{hyex}}(i,j,k)(E_{\mathrm{scat},x}(i,j,k+1) - E_{\mathrm{scat},x}(i,j,k)). \quad (18)\end{aligned}$$

In Listing 2, first the index of the cell in an xy plane cut that maps to the active thread is calculated and stored as ci. Then an index k is used to iterate in a *for* loop. While k iterates in the z direction, an index i traverses all the cells in the column of cell ci. As i proceeds, the respective value of array tmpy[i] is calculated as required by (18).

Listing 1. Launching kernels for $y' = Ax_k$.

```
void cuda_matvec (double2 * dvx, double2 * dvy)
{
int number_of_threads = 256;
int number_of_blocks = nx * ny/number_of_threads;
dim3 threads = dim3 (number_of_threads, 1, 1);
dim3 grid = dim3 (number_of_blocks, 1, 1);
cuda_matvec_tmpx_kernel <<< grid, threads >>>
(dvx, dvy, dvtmpx, dvChxey, dvChxez, nx, ny, nz);
cuda_matvec_tmpy_kernel <<< grid, threads >>>
(dvx, dvy, dvtmpy, dvChyex, dvChyez, nx, ny, nz);
cuda_matvec_tmpz_kernel <<< grid, threads >>>
(dvx, dvy, dvtmpz, dvChzey, dvChzex, nx, ny, nz);
cuda_matvec_yex_kernel <<< grid, threads >>>
(dvx, dvy, dvtmpx, dvtmpy, dvtmpz, dvCexhy, dvCexhz, nx, ny, nz);
cuda_matvec_yey_kernel <<< grid, threads >>>
(dvx, dvy, dvtmpx, dvtmpy, dvtmpz, dvCeyhx, dvCeyhz, nx, ny, nz);
cuda_matvec_yez_kernel <<< grid, threads >>>
(dvx, dvy, dvtmpx, dvtmpy, dvtmpz, dvCezhy, dvCezhx, nx, ny, nz);
}
```

In (18) one can notice that $E_{\text{scat},x}(i, j, k)$ and $E_{\text{scat},x}(i, j, k + 1)$ are used together. Therefore, both values will be fetched from the global memory during an iteration of k. At the next iteration of k, the previous value of $E_{\text{scat},x}\,(i,\ j,\ k + 1)$ becomes the new value of $E_{\text{scat},x}\,(i,\ j,\ k)$, and it is already available in the local memory, so there is no need to fetch it again from the global memory. Thus, this iterative procedure saves a global memory fetch and slightly improves the speed of the calculations.

In (18) $E_{\text{scat},z}$ is first copied into shared memory that is indicated with a pointer S. Here $E_{\text{scat},z}\,(i,\ j,\ k)$ and $E_{\text{scat},z}\,(i + 1,\ j,\ k)$ are needed together. When the threads in a thread block fetch respective elements of $E_{\text{scat},z}\,(i,\ j,\ k)$ from global memory, the next element $E_{\text{scat},z}\,(i + 1,\ j,\ k)$ on the boundary of the thread block will not be available to use yet. Therefore, another fetch, controlled by if $(ti < 16)$ statement, from global memory is made and the data is copied into the shared memory. Once all data are available on the shared memory, the respective instructions afterwards are performed. Other kernels are implemented similarly. This type of implementation is very similar to that of FDTD discussed in [2].

Listing 2. Second kernel in Listing 1.

```
__global__ void cuda_matvec_tmpy_kernel
(cuDoubleComplex * x, cuDoubleComplex * y, cuDoubleComplex * tmpy,
cuDoubleComplex * chyex, cuDoubleComplex * chyez,  int nx, int ny, int nz)
{
__shared__ cuDoubleComplex S[256 + 16];
int ti = threadIdx.x;   int ci = blockIdx.x * blockDim.x + threadIdx.x;
int i, k;   int nxy = nx * ny;   int nxyz = nxy * nz;
cuDoubleComplex * ex = x;   cuDoubleComplex * ey = &x[nxyz];
cuDoubleComplex * ez = &x[2 * nxyz];
double2 a, b, c, exi, exi_p;   i = ci;   exi = ex[i];
for (k = 0; k < nz - 1; k++)
{
exi_p = ex[i + nxy];   S[ti] = ez[i];
if (ti < 16)
{
S[ti + blockDim.x] = ez[i + blockDim.x];
}
__syncthreads();
a.x = exi_p.x - exi.x;          a.y = exi_p.y - exi.y;
b.x = S[ti].x - S[ti + 1].x;   b.y = S[ti].y - S[ti + 1].y;
__syncthreads();
c.x = chyex[i].x * a.x - chyex[i].y * a.y +chyez[i].x * b.x - chyez[i].y * b.y;
c.y = chyex[i].x * a.y + chyex[i].y * a.x +chyez[i].x * b.y + chyez[i].y * b.x;
tmpy[i] = c; exi = exi_p; i = i + nxy;
   }
}
```

4.6. Matrix-vector Product Using CUDA: Coefficient Indices Storage Scheme

As discussed before, employing the coefficient indices scheme reduces the required memory on the global memory space of GPU; Instead of storing the coefficients, one can store the indices to coefficient pairs on the global memory. Moreover, unique coefficient pairs as well need to be copied to the GPU memory. The arrays that store each of the unique coefficient pairs are not suitable for coalesced memory access, thus it is inefficient to store them on the global memory. Instead, CUDA enabled GPU devices provide two additional memory spaces, the texture and

constant memory, which are cached to enable fast access to unordered data. However, the sizes of these memory spaces are small. As long as the unique coefficient pair arrays can fit in the limited size of these memory spaces, the coefficient indices storage scheme can be used.

The coefficients used in this contribution are with double precision. CUDA does not support double precision on the texture memory, while it supports double precision on the constant memory. Therefore, constant memory is used to store the coefficient arrays. The presented algorithm is implemented and executed on a Nvidia Tesla C1060 card which holds 64 KB constant memory. Since each coefficient pair needs 32 bytes, 2048 coefficient pairs in total can be stored on the constant memory.

Listing 3. Second kernel in Listing 1 based on coefficient indices storage scheme.

```
__global__ void
cuda_matvec_tmpy_kernel_with_indexing
(cuDoubleComplex * x, cuDoubleComplex * y,
cuDoubleComplex * tmpy, unsigned short * chy_ind,  int nx, int ny, int nz)
{ __shared__ cuDoubleComplex  S[TILE_SIZE * TILE_SIZE + TILESIZE];
int ti = threadIdx.x;  int ci = blockIdx.x * blockDim.x + threadIdx.x;
int i, k;                  int nxy = nx * ny;  int nxyz = nxy * nz;
cuDoubleComplex chyez, chyex;              unsigned short edge_index;
cuDoubleComplex * ex = x;                  cuDoubleComplex * ey = &x[nxyz];
cuDoubleComplex * ez = &x[2 * nxyz];  cuDoubleComplex a, b, c, tx, txp;
i = ci;  tx = ex[i];
for (k = 0; k < nz - 1; k++)
{ txp = ex[i + nxy];  S[ti] = ez[i];
if (ti < TILE_SIZE)
{ S[ti + blockDim.x] = ez[i + blockDim.x]; }
__syncthreads();
edge_index = chy_ind[i];  chyez = dvCh1[edge_index];  chyex = dvCh2[edge_index];
a.x = txp.x - tx.x;            a.y = txp.y - tx.y;
b.x = S[ti].x - S[ti + 1].x;   b.y = S[ti].y - S[ti + 1].y;
__syncthreads();
c.x = chyex.x * a.x - chyex.y * a.y + chyez.x * b.x - chyez.y * b.y;
c.y = chyex.x * a.y + chyex.y * a.x + chyez.x * b.y + chyez.y * b.x;
tmpy[i] = c;  tx = txp;  i = i + nxy;
} }
```

Listing 3 shows the details of the second kernel in Listing 1, based on the coefficient indices storage scheme. Here chy_ind is a coefficient indices array that reside on the global memory, while dvCh1 and dvCh2 are two coefficient arrays, each holding one of the pairs, that reside on the constant memory.

5. RESULTS

The performance of the presented algorithm is analyzed through two examples in this section. The solution times on GPU and CPU are compared. The following codes are considered in the analyses:

CPU: A BLAS version of BICGSTAB developed by Fokkema is obtained from [24]. This code is in FORTRAN. Single-threaded version of Intel Math Kernel Library (Intel MKL) is used to execute the BLAS routines. The function that performs the matrix-vector product (matvec) is implemented in FORTRAN as presented in [14]. The code is run on an Intel Xeon E5405 CPU at 2 GHz.

GPU: The codes of the presented algorithm based on the coefficient arrays and coefficient indices storage schemes are run on an Nvidia Tesla C1060 computation card at 1.3 GHz. The abbreviations CAS and CIS denote the coefficient arrays and coefficient indices storage schemes, respectively, in the following discussions.

5.1. Scattering from a Dielectric Sphere

First, scattering from a dielectric sphere is calculated. The sphere has 7.2 cm radius and dielectric constant of 4. The incident field is an x polarized plane wave at 1 GHz traveling in $+z$ direction. The cubic computational domain is 32 cm on a side. FDFD calculations are performed, both on CPU and GPU, each time with a different cell size, thus with a different number of cells. The results are shown in Table 1. It can be seen that as problem size gets bigger, the efficiency of the CUDA program increases. It is possible to solve FDFD on a GPU 20 times faster than on a CPU. Moreover, the coefficient indices storage scheme is slightly faster than the coefficient arrays storage scheme. Fig. 2 shows bistatic radar cross-section (RCS) of the sphere calculated by CUDA FDFD and compared with an analytical solution obtained from a Matlab program presented in [26].

5.2. Scattering from a Human Head

As a second example, calculation of scattered fields from a human head is presented here. First, an updated, dedicated MRI head phantom

Table 1. Results for a sphere.

cell size (mm)	total number of cells	time on CPU (seconds)	time on GPU (CAS) (seconds)	time on GPU (CIS) (seconds)
5	$64 \times 64 \times 64$	337	22	21
4	$80 \times 80 \times 80$	851	49	42
3.333	$96 \times 96 \times 96$	1901	115	99

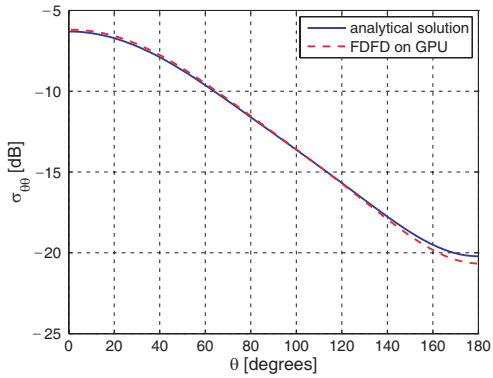

Figure 2. Bistatic radar cross-section of a sphere with 7.2 cm radius and dielectric constant of 4 at 1 GHz. Cell size is 4 mm on a side. Domain size is $N = 80 \times 80 \times 80$ cells.

made available by techniques described in [27] is downloaded from [28], which is $256 \times 256 \times 128$ cells in size. This anatomical model data is decimated by half in each direction and a $128 \times 128 \times 64$ cells model is obtained. A cross-section view of the head that shows tissue distribution is shown in Fig. 3. In this model the cell size is 2.2 mm $\times$ 2.2 mm $\times$ 2.8 mm, where each cell is filled with a tissue material. Then, permittivity and conductivity values of these tissues are obtained from [29]. This head model is placed in an FDFD problem space with size $160 \times 160 \times 93$, and FDFD calculations are performed to calculate scattered electric fields due to an incident plane wave with z polarization at 900 MHz that hits the head on its side. Fig. 4 shows the z component of scattered electric field on an xy cross-section.

Table 2 shows solution times for this case. Solution of FDFD is achieved about 28 times faster on a GPU compared with that on a CPU. One can notice that, as problem size gets larger, solution speed-up factor gets higher on GPU.

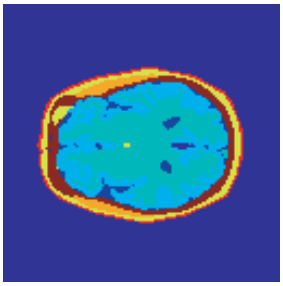

Figure 3. An xy cross-section view of an anatomical head model used in FDFD simulation.

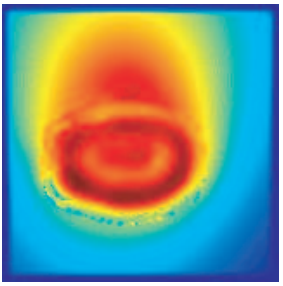

Figure 4. Scattered electric field ($E_{\text{scat},z}$) distribution from a human head on an xy cross-section.

Table 2. Results for a human head.

total number of cells	time on GPU (minutes)	time on GPU (CAS) (minutes)	time on CPU (CIS) (minutes)
$160 \times 160 \times 93$	130	5.4	4.6

5.3. Use of Preconditioners

It is widely recognized that preconditioning is the most critical ingredient in the development of efficient solvers for challenging problems in scientific computation, and that the importance of preconditioning is destined to increase even further [23]. A well-chosen preconditioner can significantly improve the efficiency of an iterative solver. The current contribution addresses the GPU acceleration of FDFD, where the BICGSTAB algorithm is utilized. One should consider finding and using an efficient preconditioner as well to further accelerate the computations both on the CPU and GPU platforms.

Analyses of simple preconditioners, such as Jacobi, ILU(0), and SSOR [18], are performed to evaluate their effectiveness to improve the solution speed. First, Jacobi and ILU(0) preconditioners are considered. It has been found that these preconditioners do not improve the solution. Moreover, sometimes they yield divergent solutions. The SSOR preconditioner is found to improve convergence rate significantly, i.e., convergence is achieved with a much less number of iterations, however, the overhead introduced by the application of the preconditioner adversely affects the solution speed, and the overall solution time becomes more with the preconditioner.

These preconditioner analyses are performed on a CPU platform. It should be noted that an effective implementation of a preconditioner is more difficult on a GPU platform. Most preconditioners are based on LU decomposition and forward and backward solutions of resulting triangular matrix equations. In the standard forward and backward substitution algorithms for solving triangular systems, the outer loop of the substitution for each unknown is sequential. Therefore, these algorithms are not suitable for parallel processing. Some other methods, such as level scheduling, which exploits topological sorting [18], need to be employed for better parallelism [31].

6. CONCLUSION

A CUDA implementation of FDFD method is presented in this contribution to speed-up the solution of electromagnetics problems. The presented algorithm uses BICGSTAB iterative solver. The BICGSTAB solver is ported to run on a GPU using CUDA. Then, integrated with BICGSTAB, an efficient method of performing matrix-vector products for the linear system of FDFD equations is adapted from [14] and implemented in CUDA based on two coefficient schemes to further improve the speed of calculations. It has been shown that, with the presented algorithm, FDFD can be solved more than 20 times faster on a GPU than on a CPU.

REFERENCES

1. Sypek, P., A. Dziekonski, and M. Mrozowski, "How to render FDTD computations more effective using a graphics accelerator," *IEEE Transactions on Magnetics*, Vol. 45, No. 3, 1324–1327, 2009.
2. Demir, V. and A. Z. Elsherbeni, "Compute unified device architecture (CUDA) based finite-difference time-domain (FDTD) implementation," *Journal of the Applied Computational Electromagnetics Society (ACES)*, Vol. 25, No. 4, 303–314, April 2010.
3. Ong, C. Y., M. Weldon, S. Quiring, L. Maxwell, M. C. Hughes, C. Whelan, and M. Okoniewski, "Speed it up," *IEEE Microwave Magazine*, Vol. 11, No. 2, 70–78, April 2010.
4. De Donno, D., A. Esposito, L. Tarricone, and L. Catarinucci, "Introduction to GPU computing and CUDA programming: A case study on FDTD," *IEEE Antennas and Propagation Magazine*, EM Programmer's Notebook, Vol. 52, No. 3, 116–122, June 2010.

5. Gödel, N., N. Nunn, T. Warburton, and T. Clemens, "Scalability of higher-order discontinuous galerkin FEM computations for solving electromagnetic wave propagation problems on GPU clusters," *IEEE Transactions on Magnetics*, Vol. 46, No. 8, August 2010.
6. De Donno, D., A. Esposito, G. Monti, and L. Tarricone, "GPU-based acceleration of the MPIE/MoM matrix calculation for the analysis of microstrip circuits," *Proceedings of the European Conference on Antennas and Propagation (EuCAP 2011)*, Rome, Italy, April 2011.
7. Al Sharkawy, M. H., V. Demir, and A. Z. Elsherbeni, *Electromagnetic Scattering Using the Iterative Multi-Region Technique (Synthesis Lectures on Computational Electromagnetics)*, Morgan and Claypool Publishers, November 2007.
8. Al Sharkawy, M. H., V. Demir, and A. Z. Elsherbeni, "The iterative multi-region algorithm using a hybrid finite difference frequency domain and method of moments techniques," *Progress In Electromagnetics Research*, Vol. 57, 19–32, 2006.
9. Al Sharkawy, M. H., V. Demir, and A. Z. Elsherbeni, "Plane wave scattering from three dimensional multiple objects using the Iterative Multi-Region technique based on the FDFD method," *IEEE Transactions on Antennas and Propagation*, Vol. 54, No. 2, 666–673, February 2006.
10. Kuzu, L., V. Demir, A. Z. Elsherbeni, and E. Arvas, "Electromagnetic scattering from arbitrarily shaped chiral objects using the finite difference frequency domain method," *Progress In Electromagnetics Research*, Vol. 67, 1–24, 2007.
11. Alkan, E., V. Demir, A. Z. Elsherbeni, and E. Arvas, "Double-grid finite-difference frequency-domain method for modeling chiral medium," *IEEE Transactions on Antennas and Propagation*, Vol. 58, No. 3, 817–823, March 2010.
12. Zainud-Deen, S. H., E. El-Deen, M. S. Ibrahim, K. H. Awadalla, and A. Z. Botros, "Electromagnetic scattering using GPU-based finite difference frequency domain method," *Progress In Electromagnetics Research B*, Vol. 16, 351–369, 2009.
13. De Donno, D., A. Esposito, G. Monti, and L. Tarricone, "Iterative solution of linear systems in electromagnetics (and not only): Experiences with CUDA," *UnConventional High Performance Computing, Lecture Notes in Computer Science (LNCS)*, Vol. 6586, Springer, 2011.
14. Demir, V., E. Alkan, A. Z. Elsherbeni, and E. Arvas, "An algorithm for efficient solution of finite-difference frequency-

domain (FDFD) methods," *IEEE Antennas and Propagation Magazine*, Vol. 61, No. 6, 143–150, December 2009.

15. Kunz, K. S. and R. J. Luebbers, *The Finite Difference Time Domain Method for Electromagnetics*, CRC Press, 1993
16. Berenger, J. P., "A perfectly matched layer for the absorption of electromagnetic waves," *Journal of Computational Physics*, Vol. 114, No. 2, 185–200, October 1994.
17. Yee, K. S., "Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media," *IEEE Transactions on Antennas and Propagation*, Vol. 14, 302–307, May 1966.
18. Saad, Y., *Iterative Methods for Sparse Linear Systems*, Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, USA, 2003.
19. Barrett, R., M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, 2nd Edition, Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, USA, 1994.
20. Saad, Y. and M. H. Schultz, "GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems," *SIAM Journal on Scientific and Statistical Computing*, Vol. 7, No. 3, 856–869, July 1986.
21. Van der Vorst, H. A., "BI-CGSTAB: A fast and smoothly converging variant of BI-CG for the solution of nonsymmetric linear systems," *SIAM Journal on Scientific and Statistical Computing*, Vol. 13, No. 2, 631–644, March 1992.
22. Sleijpen, G. L. G. and D. R. Fokkema, "BICGSTAB(l) for linear equations involving unsymmetric matrices with complex spectrum," *Electronic Transactions on Numerical Analysis*, Vol. 1, 11–32, 1993.
23. CUDA 2.1 Programming Guide, http://www.nvidia.com/object/-cuda_develop.html.
24. http://www.staff.science.uu.nl/∼vorst102/software.html.
25. Blackford, L. S., J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley, "An updated set of basic linear algebra subprograms (BLAS)," *ACM Trans. Math. Soft.*, Vol. 28, No. 2, 135–151, 2002.
26. Demir, V., A. Z. Elsherbeni, D. Worasawate, and E. Arvas, "A graphical user interface (GUI) for plane wave scattering from a

conducting, dielectric or a chiral sphere," *IEEE Antennas and Propagation Magazine*, Vol. 46, No. 5, 94–99, October 2004.

27. Zubal, I. G., C. R. Harrell, E. O. Smith, Z. Rattner, G. R. Gindi, and P. B. Hoffer, "Computerized three-dimensional segmented human anatomy," *Medical Physics*, Vol. 21, No. 2, 299–302, February 1994.
28. The Zubal Phantom, http://noodle.med.yale.edu/zubal/index.h-tm.
29. "An internet resource for the calculation of the dielectric properties of body tissues in the frequency range 10 Hz–100 GHz," 2011, http://niremf.ifac.cnr.it/tissprop/.
30. Benzi, M., "Preconditioning techniques for large linear systems: A survey," *Journal of Computational Physics*, Vol. 182, No. 2, 418–477, November 2002.
31. Li, R. and Y. Saad, "GPU-accelerated preconditioned iterative linear solvers," CUDA_ITSOL Technical Report, available at http://www-users.cs.umn.edu/∼rli/, 2011.