Progress In Electromagnetics Research B, Vol. 16, 351–369, 2009

ELECTROMAGNETIC SCATTERING USING GPU-BASED FINITE DIFFERENCE FREQUENCY DOMAIN METHOD

S. H. Zainud-Deen and E. El-Deen

Faculty of Electronic Engineering
Menoufia University
Egypt

M. S. Ibrahim

Faculty of Engineering
Cairo University
Egypt

K. H. Awadalla

Faculty of Electronic Engineering
Menoufia University
Egypt

A. Z. Botros

Faculty of Engineering
Cairo University
Egypt

Abstract—This paper presents a graphics processing based implementation of the Finite-Difference Frequency-Domain (FDFD) method, which uses a central finite differencing scheme for solving Maxwell's equations for electromagnetics. The radar cross section for different structures in 2D and 3D has been calculated using the FDFD method. The FDFD code has been implemented for the CPU calculations and the same code is implemented for the GPU calculations using the Brook+ developed by AMD. The solution obtained by using the GPU based-code showed more than 40 times speed over the CPU code.

Corresponding author: E. El-Deen (emadtork@yahoo.com).

1. INTRODUCTION

Over the last decade, Graphics Processing Units (GPUs) have developed rapidly being primitive drawing devices to being major computing resources. The newest GPUs have as many as 220 million transistors, approximately twice as many as a typical Central Processing Unit (CPU) in a PC. Moreover, the cache consumes most of the transistors in a CPU, while GPUs use only small caches and devote the majority of transistors to computation. This large number of parallel processing elements converts the GPU into a parallel computing system [1, 2]. Recently, the development of the GPUs has reached a new high-point with the addition of double precision 64 bit floating point capabilities and availability of high level language programming interface [3]. This development has facilitated the abstraction of the modern GPU as a stream processor. Compared with the CPU, the GPU is better suited for parallel processing and vector processing and has evolved to perform various types of numerical computations. In general, there are two factors that in some cases make GPU an attractive target architecture for accelerating general purpose computations. First, the raw throughput speed of GPU ($\sim$ 1.2 TFLOPS) compared with CPU ($\sim$ 12 GFLOPS) [3, 4]. Second, the GPU acts as a co-processor, which is the key motivation fact for using it for parallel processing.

The high performance of GPU has been reported in many applications such as sparse linear system solvers, physical simulations, signal processing, and image processing. These achievements have demonstrated the potential power of GPU in the field of scientific computations. The applications of GPU in the area of computational electromagnetics started in the finite-difference time-domain method (FDTD) where the acceleration ratio reached approximately 25 times compared with the CPU performance [5–7]. The graphics processing unit has also been used to speed up the method of moments (MoM) calculations for electromagnetic scattering from arbitrary three-dimensional conducting objects where 30 times acceleration ratio is achieved [8]. Further, the GPU has been used to move all electromagnetic computing code to graphical hardware using the Graphical Electromagnetic Computing (GRECO) method where it has achieved approximately 30 times faster results [9].

The finite-difference frequency-domain method (FDFD) has received considerable interest as an efficient, full-wave solution method for electromagnetic problems [10–15]. The finite-difference frequency-domain is simple in formulation and most flexible in modeling arbitrary shaped inhomogeneous filled and anisotropic scatterers. FDFD solves

Maxwell's equations in the frequency domain by replacing the partial derivatives with center difference approximation as the FDTD method. Although the final solution of FDTD is obtained by iterating the solution through the time, the result in the FDFD method is achieved by solving a system of linear equations $\mathbf{AX} = \mathbf{Y}$, where $\mathbf{A}$ is the sparse coefficient matrix, $\mathbf{Y}$ is the excitation vector and $\mathbf{X}$ is the unknown vector as the case in the MoM.

In this work, the FDFD method is implemented on a GPU. To the best of the authors' knowledge, this is the first time to implement the FDFD to run on the GPU. The iterative bi-conjugate gradient method (BICG) [16] is used to solve the sparse matrix for 2D and 3D scattering problems using the FDFD method. The use of the GPU-based computation made it quite feasible to study some complex problems like the scattering of electromagnetic waves on different structures metamaterial layers. The work done here has revealed that it is only one parameter which significantly affects the scattering from metamaterial layers. The rest of the paper is organized as follows. Section 2 presents a brief description of the FDFD method. Section 3 presents an overview of using GPUs for general purpose computation. Section 4 discusses the implementation of the FDFD on GPU. In Section 5, numerical results are given and investigated. Finally, Section 6 concludes this study.

2. FDFD METHOD

The finite-difference frequency-domain method solves Maxwell's equations in the frequency domain by replacing the partial derivatives with center-difference approximation. The first step in constructing FDFD algorithm is to discretize the computational domain into N_x, N_y, and N_z cells in the x-, y-, and z-directions, respectively, and define the locations of the electric and magnetic field vectors on each cell. The Yee [17] cell will be used in this work where the locations of the vector components are defined. Because the basic building block (Yee cell) is a cube, curved surfaces of a scatterers are staircased. Through this paper there is no special treatment for material discontinuities. The uniaxial perfectly matched layer (UPML) [18], as an absorbing boundary, is used to terminate the computational domain. Maxwell's curl equations for the total electric and magnetic field components and using the UPML formulation can be written as [18],

$$\nabla \times \bar{H}^{total} = (j\omega\varepsilon + \sigma^e)\bar{S}\,\bar{E}^{total} \tag{1}$$

$$\nabla \times \bar{E}^{total} = -(j\omega\mu + \sigma^m)\bar{S}^*\bar{H}^{total} \tag{2}$$

where $\bar{S}$ and $\bar{S}^*$ are a diagonal tensors defined by:

$$\bar{S}=\begin{bmatrix} s_y s_z/s_x & 0 & 0 \\ 0 & s_x s_z/s_y & 0 \\ 0 & 0 & s_x s_y/s_z \end{bmatrix} \text{ and } \bar{S}^*=\begin{bmatrix} s_y^* s_z^*/s_x^* & 0 & 0 \\ 0 & s_x^* s_z^*/s_y^* & 0 \\ 0 & 0 & s_x^* s_y^*/s_z^* \end{bmatrix}$$

and

$$s_x = 1+\sigma_x^{PML}/j\omega\varepsilon_0, \quad s_y = 1+\sigma_y^{PML}/j\omega\varepsilon_0, \quad s_z = 1+\sigma_z^{PML}/j\omega\varepsilon_0,$$

$$s_x^* = 1+\sigma_x^{*PML}/j\omega\mu_0, \quad s_y^* = 1+\sigma_y^{*PML}/j\omega\mu_0, \quad s_z^* = 1+\sigma_z^{*PML}/j\omega\mu_0$$

For perfect matching between the free space and the UPML the following condition should be verified [18]:

$$\frac{\sigma^{PML}}{\varepsilon_0} = \frac{\sigma^{*PML}}{\mu_0} \tag{3}$$

In Equations (1) and (2), by separating the total fields into incident and scattered field components, then:

$$\nabla \times (\bar{H}^{inc} + \bar{H}^{scat}) = (j\omega\varepsilon + \sigma^e)\bar{S}\,(\bar{E}^{inc} + \bar{E}^{scat}) \tag{4}$$

$$\nabla \times (\bar{E}^{inc} + \bar{E}^{scat}) = -(j\omega\mu + \sigma^m)\bar{S}^*(\bar{H}^{inc} + \bar{H}^{scat}) \tag{5}$$

where, E^{inc} and H^{inc} are the incident fields that would exist in the computational domain with no scatterers. If the computational domain is free space, the incident field satisfies Maxwell's equations, such that:

$$\nabla \times \bar{H}^{inc} = j\omega\varepsilon_o\bar{S}\bar{E}^{inc} \tag{6}$$

$$\nabla \times \bar{E}^{inc} = -j\omega\mu_o\bar{S}^*\bar{H}^{inc} \tag{7}$$

Substituting of (6) and (7) in (4) and (5), yields:

$$E^{scat} = \frac{1}{(j\omega\varepsilon+\sigma^e)\bar{S}}\nabla\times H^{scat} - \frac{j\omega(\varepsilon-\varepsilon_0)+\sigma^e}{(j\omega\varepsilon+\sigma^e)}E^{inc} \tag{8}$$

$$H^{scat} = -\frac{1}{(j\omega\mu+\sigma^m)\bar{S}^*}\nabla\times E^{scat} - \frac{j\omega(\mu-\mu_0)+\sigma^m}{(j\omega\mu+\sigma^m)}H^{inc} \tag{9}$$

Applying the central difference approximations on the above equations yields:

$$\begin{aligned} E_x^{scat}(i,j,k) = {} & \frac{1}{L_x(j\omega\varepsilon_x+\sigma_x^e)}\left[\frac{H_z^{scat}(i,j,k)-H_z^{scat}(i,j-1,k)}{\Delta y}\right. \\ & \left. - \frac{H_y^{scat}(i,j,k)-H_y^{scat}(i,j,k-1)}{\Delta z}\right] \\ & - \frac{(j\omega(\varepsilon_x-\varepsilon_0)+\sigma_x^e)}{(j\omega\varepsilon_x+\sigma_x^e)}E_x^{inc}(i,j,k) \end{aligned} \tag{10}$$

$$E_y^{scat}(i,j,k) = \frac{1}{L_y(j\omega\varepsilon_y+\sigma_y^e)}\left[\frac{H_x^{scat}(i,j,k)-H_x^{scat}(i,j,k-1)}{\Delta z} - \frac{H_z^{scat}(i,j,k)-H_z^{scat}(i-1,j,k)}{\Delta x}\right] - \frac{(j\omega(\varepsilon_y-\varepsilon_0)+\sigma_y^e)}{(j\omega\varepsilon_y+\sigma_y^e)}E_y^{inc}(i,j,k) \quad (11)$$

$$E_z^{scat}(i,j,k) = \frac{1}{L_z(j\omega\varepsilon_z+\sigma_z^e)}\left[\frac{H_y^{scat}(i,j,k)-H_y^{scat}(i-1,j,k)}{\Delta x} - \frac{H_x^{scat}(i,j,k)-H_x^{scat}(i,j-1,k)}{\Delta y}\right] - \frac{(j\omega(\varepsilon_z-\varepsilon_0)+\sigma_z^e)}{(j\omega\varepsilon_z+\sigma_z^e)}E_z^{inc}(i,j,k) \quad (12)$$

$$H_x^{scat}(i,j,k) = \frac{1}{M_x(j\omega\mu_x+\sigma_x^m)}\left[\frac{E_z^{scat}(i,j+1,k)-E_z^{scat}(i,j,k)}{\Delta y} - \frac{E_y^{scat}(i,j,k+1)-E_y^{scat}(i,j,k)}{\Delta z}\right] - \frac{(j\omega(\mu_x-\mu_0)+\sigma_x^m)}{(j\omega\mu_x+\sigma_x^m)}H_x^{inc}(i,j,k) \quad (13)$$

$$H_y^{scat}(i,j,k) = \frac{1}{M_y(j\omega\mu_y+\sigma_y^m)}\left[\frac{E_x^{scat}(i,j,k+1)-E_x^{scat}(i,j,k)}{\Delta z} - \frac{E_z^{scat}(i+1,j,k)-E_z^{scat}(i,j,k)}{\Delta x}\right] - \frac{(j\omega(\mu_y-\mu_0)+\sigma_y^m)}{(j\omega\mu_y+\sigma_y^m)}H_y^{inc}(i,j,k) \quad (14)$$

$$H_z^{scat}(i,j,k) = \frac{1}{M_z(j\omega\mu_z+\sigma_z^m)}\left[\frac{E_y^{scat}(i+1,j,k)-E_y^{scat}(i,j,k)}{\Delta x} - \frac{E_x^{scat}(i,j+1,k)-E_x^{scat}(i,j,k)}{\Delta y}\right] - \frac{(j\omega(\mu_z-\mu_0)+\sigma_z^m)}{(j\omega\mu_z+\sigma_z^m)}H_z^{inc}(i,j,k) \quad (15)$$

where $L_x = s_y s_z / s_x$, $L_y = s_x s_z / s_y$, $L_z = s_x s_y / s_z$, $M_x = s_y^* s_z^* / s_x^*$, $M_y = s_x^* s_z^* / s_y^*$, and $M_z = s_x^* s_y^* / s_z^*$. Equations (10) to (15) can be reduced to three equations, in terms of the three scattered electric field components. More details about the final expressions can be found in [15]. The BICG is implemented on a GPU to fit the solution of the FDFD method both in 2D and 3D cases as will be illustrated in the next section. Once the scattered field is obtained in the near field zone, the near-to-far-field transformation [19] is used to find the far-zone scattered field.

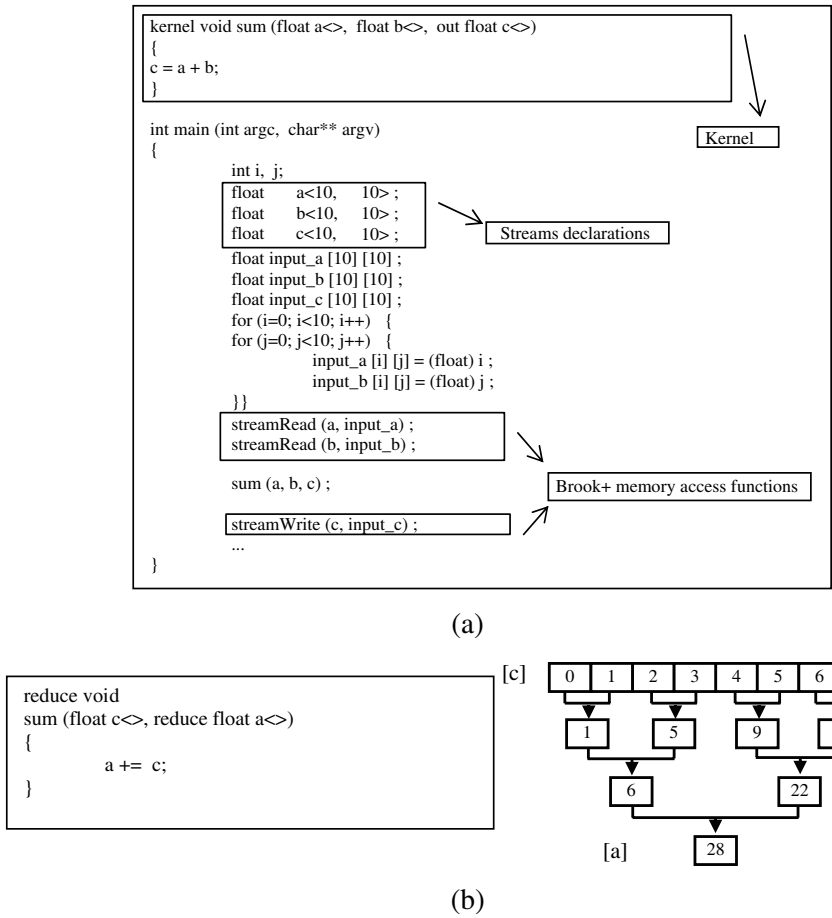

Figure 1. (a) A sample for a program written to sum two arrays on the GPU using brook+. (b) A sample of a one dimension reduce kernel.

3. GPUs BASED ALGORITHMS

GPU memory is organized in textures. In computer graphics these textures are used to store color images. The GPU is able to perform a variety of arithmetic functions on the texture. For example, it may add, subtract, multiply, or divide across the texture, which is simply done using vector math on these textures. This vector math occurs in a section of the GPU named fragment processor [6]. In ATI Radeon 4870 graphical card, which is used in this paper, there are 800 of these fragment processors. For general purpose computations, the texture forms a 2D array that can be accessed by the GPU shaders. Fortunately, in the past few years there were some high level languages that allowed developers to write their applications at an abstract level without having to worry about the exact details of the hardware. One of these languages is the Brook which is an extension to the C-language for stream programming originally developed by Stanford University [1].

In this work, the programming of the GPU is accomplished using Brook+. Brook+ is an implementation by AMD Corporation for the Brook specification on AMD's compute abstraction layer with some enhancements [3]. One of these enhancements is the introduction of the double precision data type that enables the implementation of the iterative BICG method. In Brook+, all data are presented in the form of streams, which is defined as a collection of data records that can be mapped into memory textures and operated on concurrently. Streams are processed in kernels. Kernels can be considered as functions, which receive streams, execute operations on them and send out result streams. Figure 1(a) shows a sample for a program written to sum two arrays on the GPU using Brook+. The kernel used for the summation could be used easily for multiplication of two arrays just by changing the addition sign by a multiplication sign inside the kernel. Another important kernel that is usually used for general purpose computations is the reduce kernel. Reduction on GPU provides a data-parallel method for computing a single value from a set of records. The process of reduction depends on the size of the output stream (i.e., the reduction occurs concurrently until the size of the output stream is reached). Figure 1(b) shows an example of a reduction process of a one dimension vector of size eight to an output stream of size one. More details can be found in [3].

4. IMPLEMENTATION OF FDFD METHOD ON GPU

For the solution of the linear system of equations results from the FDFD technique, the BICG method is adapted as an iterative method to generate two sequences of conjugate (or orthogonal) vectors. These vectors are the residuals of these iterates. One vector is based on the coefficient matrix $\mathbf{A}$, and the other one is based on the transpose of the coefficient matrix $\mathbf{A^T}$. Figure 2 shows the BICG algorithm used in this paper [16].

The FDFD method results in a regular sparse coefficient matrix $\mathbf{A}$ both in the 2D (only 5 non-zero elements per row in the coefficient matrix) and 3D (only 13 non-zero element per row in the coefficient matrix) problems. The sparse coefficient matrix $\mathbf{A}$ contains maximum non-zero elements equal to $5\mathrm{N_{2D}}$ for the 2D case and $13\mathrm{N_{3D}}$ for 3D case, where $\mathrm{N_{2D}} = N_x N_y$ and $\mathrm{N_{3D}} = 3N_x N_y N_z$. The sparse coefficient matrix $\mathbf{A}$ is represented by two vectors:

1. A vector containing the values of the non-zero elements.
2. A vector containing the index of the column of the non-zero elements of the coefficient matrix $\mathbf{A}$.

The vector of the unknown $\mathbf{X}$ and the vector of the excitation $\mathbf{Y}$ contain $\mathrm{N_{2D}}$ elements for the 2D case and $\mathrm{N_{3D}}$ elements for the 3D case.

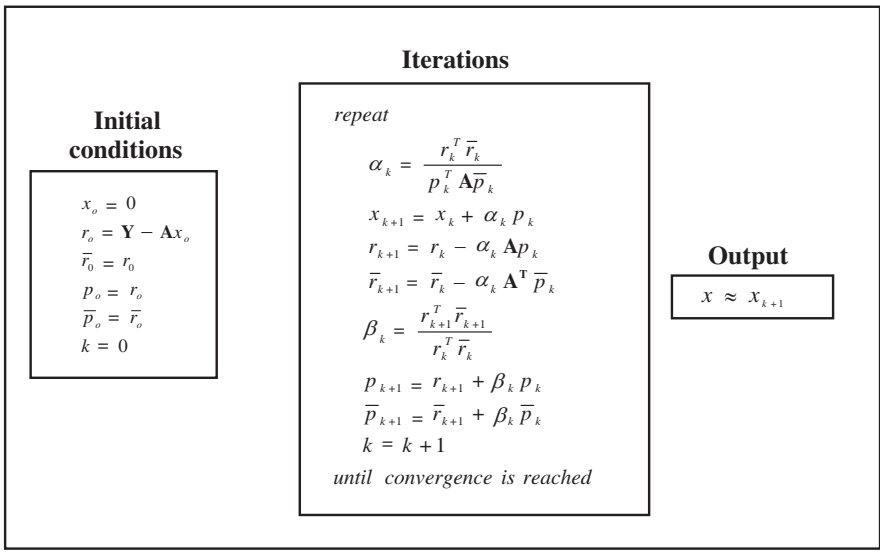

Figure 2. The bi-conjugate gradient method algorithm.

The sparse coefficient matrix **A** is mapped to a stream on the GPU with dimension selected to be $5N_x \times N_y$ for the 2D case and $39N_x \times (N_yN_z)$ for the 3D case. Further, the unknown vector **X** and the excitation vector **Y** are mapped to a stream of dimension $N_x \times N_y$ for the 2D case and to a stream of $3N_x \times (N_yN_z)$ for the 3D case. This semi-square shape enabled exploiting the maximum 2D texture size on the GPU. Two streams ***index_x*** and ***index_y*** are used to gather the elements of the unknown stream (corresponding to **X**) to the corresponding elements of the coefficient stream (corresponding to **A**) for multiplication process. The key acceleration point in solving the FDFD is the matrix vector multiplication ($\mathbf{AX} = \mathbf{Y}$) that is iteratively calculated inside the BICG algorithm and performed on the GPU. The kernel that is used for the multiplication is shown in Figure 3(a). The output stream from this kernel has the same dimension as the coefficient stream. After performing the multiplication of the two streams, the reduce kernel shown in Figure 3(b) is used to complete the original problem of the matrix vector multiplication. The reduction process represents the addition after the row-column multiplication in the matrix vector multiplications process. In the BICG algorithm, the process of the matrix vector multiplications starts by an initial guess $\mathbf{X}_o$ and go through a set of iterations until convergence is achieved and the approximated solution of the unknown vector **X** is obtained. It is noticed that, as the size of the coefficient matrix **A** increase, the BICG algorithm takes more iterations for convergence, and as a result, the simulation time increase significantly both for CPU and GPU.

```
kernel void sparse_matrix_multiplication ( float index_x<>, float  index_y<>,  double2 X[ ][ ], double2 A<>, out double2 c<> )
{
    c.x=A.x*X[index_x][index_y].x-A.y*X[index_x][index_y].y;  // real part
    c.y=A.y*X[index_x][index_y].x+A.x*X[index_x][index_y].y; // imaginary part
}
```

(a)

```
reduce void sum (double2 c<>, reduce double2 result<>)
 {
  result.x += c.x;
  result.y += c.y;
}
```

(b)

Figure 3. A sample for kernels written for sparse matrix vector product on the GPU using brook+. (a) Sparse matrix vector multiplication kernel. (b) Reduce kernel.

5. NUMERICAL RESULTS

In this section, the performance of the FDFD GPU-based code will be investigated and compared with the FDFD CPU-based code both for 2D case and the 3D case. Comparison will include the runtimes for the calculations of the radar cross section area from 2D and 3D structures. For the 2D case, the radar cross section area is given by:

$$\sigma_{2D-TM} = \lim_{\rho\to\infty} 2\pi\rho \frac{\left|E^{scat}\right|^2}{\left|E^{inc}\right|^2}, \quad \sigma_{2D-TE} = \lim_{\rho\to\infty} 2\pi\rho \frac{\left|H^{scat}\right|^2}{\left|H^{inc}\right|^2} \tag{16}$$

and for 3D case

$$\sigma_{3D} = \lim_{\rho\to\infty} 4\pi r^2 \frac{\left|E^{scat}\right|^2}{\left|E^{inc}\right|^2} \tag{17}$$

A Pentium 4 computer with 3.4 GHz processor and 2 GB DDR2 type RAM is used. It has ATI Radeon^{TM} HD 4870 graphical card with the specifications are shown in Table 1.

Figure 4 illustrates a 2D metamaterial layer contains 64 cells. Each cell consists of 3×3 C-shape conducting cylinders. The C-shape cylinder has inner radius $r_a = 0.08\lambda$, outer radius $r_b = 0.1\lambda$ and notch of size 0.02λ. For all the 2D calculations, 10 UPML cells are used. In order to represent the notch of the size $0.02\,\lambda$, the space discretization steps are selected to be $\Delta x = \Delta y = 0.0067\,\lambda$. Each 3×3 C-shape conducting cylinders occupy a domain size $N_x \times N_y$ equal to 160×160 Yee cells. The bistatic RCS of one cell is calculated and compared with [20], as shown in Figure 5. An incident TM-plane wave at ($\phi = 90°$, and $f = 300$ MHz) is used. The FDFD method implemented

Table 1. ATI Radeon^{TM} HD 4870-GPU specifications.

Graphical Bus	2.0×16 bus interface
VRAM	1024 MB GDDR5 RAM
Flops	1200 GFlops
GPU's core clock	750 MHz
GPU's memory clock	1800 MHz
Memory Bandwidth (GB/s)	115.2 GB/sec
Texture fill rate	30000 Mtexels/sec
Maximum texture size	8192×8192
Number of stream processors	800 stream processors

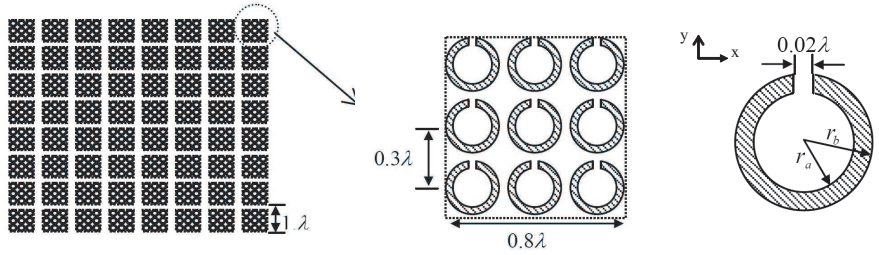

Figure 4. Two dimension metamaterial layers consisting of 8 blocks each block is a group of 3×3 C-shape conducting cylinders. (a) 2D layer consists of 8 × 8 cells, (b) one cell consists of 3 × 3 C-shape, (c) C-shape conducting cylinder with $r_b = 0.1\lambda$, $r_a = 0.08\lambda$.

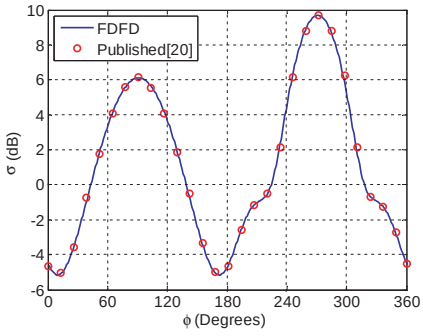

Figure 5. Bistatic RCS of one cell 2D metamaterial layer, $\phi_i = 90°$ and $f = 300\,\text{MHz}$.

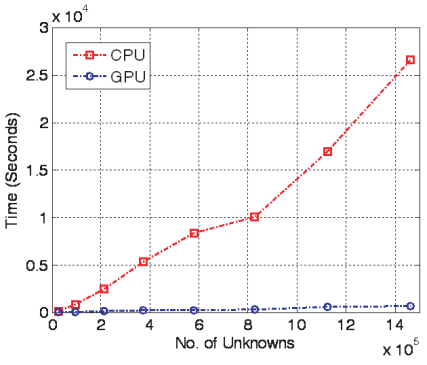

Figure 6. Performance comparison between the 2D-FDFD GPU-based and the 2D-FDFD CPU based.

on both the GPU and the CPU is considered for the calculated results. Good agreements are obtained with the published results.

As the number of cells increase, the number of the unknowns increases. Figure 6 shows a comparison between the runtime of the results of the FDFD GPU-based and that for FDFD CPU-based as the number of unknowns increases. The distance between any two successive cells is 1λ.

Table 2 shows an exhaustive test of the time cost and acceleration ratio of the algorithm. The maximum acceleration ratio for the 2D case is beyond 41. Table 3 shows the results obtained in Table 2 when it's normalized to the values of the first row. It is clear that the increase in the number of unknowns causes a real fast increase in the CPU time

needed, but it has a slow increase in the GPU time needed. This is of course attributed to the parallel computing base for the GPU.

Table 2. Performance comparison between GPU/CPU FDFD.

2D			
Number of Unknowns	Time (seconds)		Acceleration Ratio
	CPU	GPU	
160×160	109.81	42.08	2.61
310×310	866.44	74.78	11.59
460×460	2482.60	109.28	22.72
610×610	5329.50	209.18	25.48
760×760	8337.40	238.23	35
910×910	10002.00	322.50	31.01
1060×1060	16914.00	574.53	29.44
1210×1210	26591.00	646.37	41.13

Table 3. Normalized performance.

Normalized number of unknowns	Normalized CPU time	Normalized GPU Time
1	1	1
3.75	7.89	1.77
8.26	22.60	2.59
14.53	48.53	4.97
22.56	75.92	5.66
32.34	91.08	7.66
43.89	154.02	154.02
57.19	242.15	15.36

Figure 7 illustrates the geometry of one cell 2D layer when the shape of the conducting cylinders is changed to be square and triangle shapes. Figure 8 shows the comparison of bistatic radar cross section (BiRCS) of one cell 2D layer, ($\phi_i = 90°$, and $f = 300\,\text{MHz}$,) for conducting cylinders with different shapes, C-shape, square shape, triangle shape for the dimensions given in Figure 4 and Figure 7. Little

variations in the RCS are noticed when the shape is changed. The change is mainly at $\phi = 90$ and 270. Also, it is clear that the radar cross section is related to the physical dimensions of the conducting cylinder cross sectional area. It is higher for the square and lower for the equilateral triangle. The effect of changing the notch size d_0 on the RCS calculations for the case of the square shape conducting cylinder is investigated. Three cases are considered for $d_0 = 0.0\lambda$, 0.02λ, and 0.2λ in Figure 9. The BiRCS has approximately the same shape. This is due to the small current distribution along the notch element that results in little variations in the RCS. Figure 10 shows BiRCS for the conducting cylinder with square shape for different edge lengths d_1 with $d_2 = 0.3\lambda$ and $d_0 = d_1/10$. As the edge length d_1 decrease the RCS decrease. The effect of changing the spacing between the square elements d_2 in the unit cell with the other dimensions fixed as in Figure 7 is shown in Figure 11, there is a significant change in the

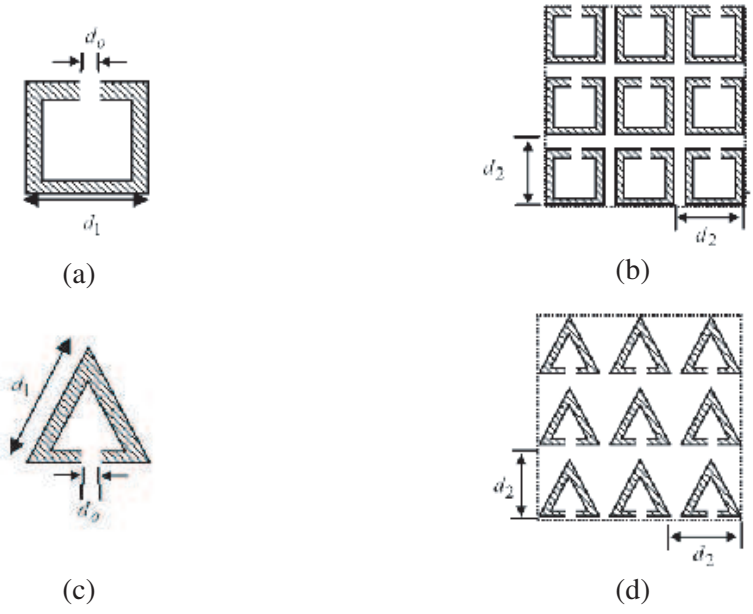

Figure 7. Geometry of one cell consisting of conducting cylinder with square shape and conducting cylinder with triangle shape. (a) Conducting cylinder with square shape with $d_0 = 0.02\lambda$, $d_1 = 0.2\lambda$, (b) one cell 2D layer consists of 3×3 elements $d_2 = 0.3\lambda$, (c) conducting cylinder with triangle shape $d_0 = 0.02\lambda$, $d_1 = 0.2\lambda$, (d) one cell 2D layer consists of 3×3 elements $d_2 = 0.3\lambda$.

RCS of the unit cell. This is because when the separation between the elements is small compared to the wavelength, there is an induced current that is concentrated mainly around the contour that surrounds the unit cell. As the separation between the elements increases, the induced current distribution on the unit cell changes and there will be an induced current that is distributed over the whole elements of the unit cell.

Figure 12 compares the measured bistatic radar cross section of a metallic cube [21] with the corresponding bistatic radar cross

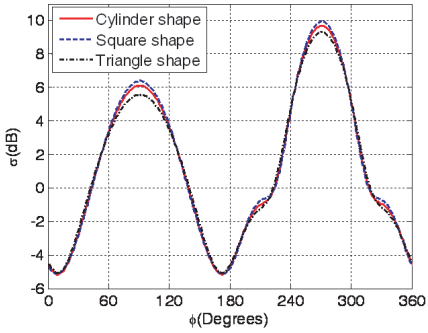

Figure 8. Comparison of bistatic RCS of one cell 2D layer, $\phi_i = 90°$, and $f = 300\,\text{MHz}$ for cylindrical, square, and triangle element shapes.

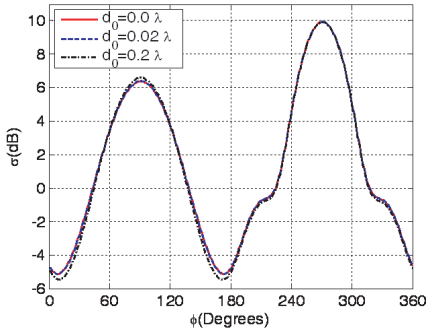

Figure 9. The effect of the notch size on the Bistatic RCS for one cell consisting of 3×3 conducting cylinder with square shape.

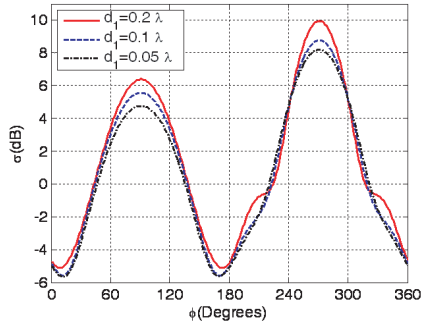

Figure 10. The effect of the edge length d_1 on the Bistatic RCS for one cell consisting of 3×3 square elements.

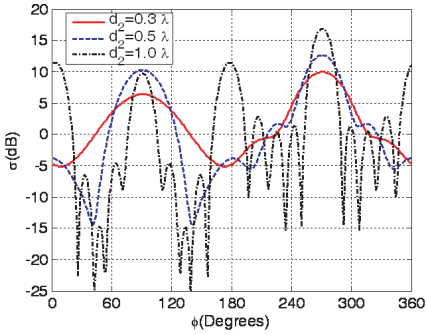

Figure 11. The effect of the elements spacing d_2 on the Bistatic RCS for one cell consisting of 3×3 square elements.

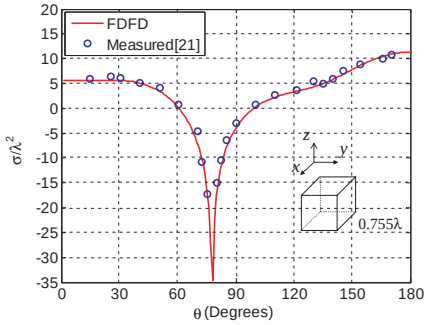

Figure 12. Bistatic RCS of a metallic cube with an edge length of 0.755λ.

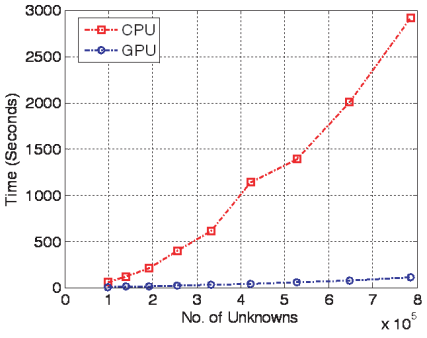

Figure 13. Performance comparison between the 3D-FDFD GPU based and the 3D-FDFD CPU based.

section computed by the FDFD method. The metallic cube have an edge length of 0.755λ and it is illuminated by an incident plane wave with ($\theta = 180°$, $\phi_i = 0°$ and $f = 300\,\text{MHz}$). The space discretization steps used to represent the cube in the FDFD method are $\Delta x = \Delta y = \Delta z = 0.126\lambda$ and 7 UPML cells are used to truncate the domain. Again, good agreements are obtained with the published results. To compare the performance of the 3D FDFD CPU-based code and the 3D FDFD GPU-based code the same cube is used but

Table 4. Performance comparison between GPU/CPU FDFD.

3D			
Number of Unknowns	Time (seconds)		Acceleration Ratio
	CPU	GPU	
$32 \times 32 \times 32 \times 3$	57.20	7.87	7.27
$36 \times 36 \times 36 \times 3$	119.36	12.90	9.25
$40 \times 40 \times 40 \times 3$	211.77	15.37	13.78
$44 \times 44 \times 44 \times 3$	393.39	20.92	18.8
$48 \times 48 \times 48 \times 3$	607.31	33.07	18.36
$52 \times 52 \times 52 \times 3$	1141.50	42.20	27.05
$56 \times 56 \times 56 \times 3$	1391.00	61.87	22.48
$60 \times 60 \times 60 \times 3$	2002.20	80.17	24.97
$64 \times 64 \times 64 \times 3$	2913.80	114.73	25.40

Table 5. Normalized performance.

Normalized number of unknowns	Normalized CPU time	Normalized GPU time
1	1	1
1.42	2.08	1.63
1.95	3.70	1.95
2.59	6.87	2.65
3.37	10.61	4.20
4.29	19.95	5.36
5.35	24.31	7.86
6.59	35.00	10.18
8	50.94	14.57

the number of the unknowns to be solved is increased gradually by reducing the space discretization steps. Figure 13 shows a comparison between the time period that the 3D-FDFD GPU based code consumes compared with the time period for the 3D FDFD CPU-based code for different number of unknowns. As the number of unknowns increases the performance of the GPU exceeds that of the CPU and a speed up factor of about 25 is achieved for the considered case. It can be noticed that for the 3D case, the maximum number of unknowns that can be carried on the available GPU is less than that of the 2D case because of the difference in the amount of storage space required for the coefficient matrix **A** on the GPU RAM. A cache miss problem will arise when the number of the unknowns exceeds the on board RAM of the GPU. Table 4 shows the time cost and the acceleration ratio of the algorithm as the number of unknowns is increased. Table 5 shows the results obtained in Table 4 when it's normalized to the values of the first row. It is clear that the increase in the number of unknowns causes a real fast increase in the CPU time needed compared with the slow increase in the GPU time needed. For example, when the number of the unknowns is doubled, the convergence rate of the CPU time increased by 3.7 times, but for the GPU, the convergence rate took 1.95 times. Also, it is noticed that the convergence rate is almost linear for the GPU-based code. Figure 14 shows the bistatic RCS for a conducting sphere of diameter 0.5λ located inside a dielectric cube of edge length $= 0.755\lambda$ with relative dielectric constant $\varepsilon_r = 4$ and

the incident plane wave has ($\theta = 180°$, $\phi = 0°$, and $f = 300\,\text{MHz}$). The FDFD CPU-based code took 5054 second to converge while the FDFD GPU-based code took 361 second. The speed up factor in this example is 14 times. This means that the speed up factor depends on the shape, the material, and the size of the object.

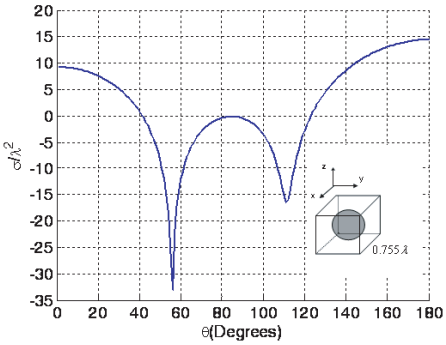

Figure 14. Bistatic RCS of a conducting sphere of a diameter $= 0.5\lambda$ inside a dielectric cube with an edge length of 0.755λ and $\varepsilon_r = 4$.

6. CONCLUSION

In this paper, 2D and 3D FDFD codes are developed to run on GPU. A comparison between the CPU based-code and the GPU based code, based on the calculation of RCS for targets in 2D and 3D cases, showed a speed up factor up to 40 for the calculations performed on the GPU for the selected examples. If the problem size (number of unknowns) increases the speed up factor could increase more. The two limiting factors that could limit the current problem size are the on board RAM of the graphical cards and the maximum texture size that can be handled. However the on board RAM of the graphical cards is the more effective for the FDFD problem. Also, the use of the GPU based computation, allows the change of the different parameters of the metamaterial structure. It has become clear that the main parameter that significantly affects the RCS of the metamaterial structure is the separation between the elements of the metamaterial structure and the details of the structure has no-significant effect on the RCS.

REFERENCES

1. "GPGPU," http://www.gpgpu.org.

2. Owens, J. D., M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proceedings of the IEEE*, Vol. 96, No. 5, 879–899, 2008.
3. "Streamcomputing," http://ati.amd.com/technology.
4. "Intel processors product list," http://support.intel.com/support/processor.
5. Inman, M. J. and A. Z. Elsherbeni, "Optimization and parameter exploration using GPU based FDTD solvers," *IEEE MTT-S Symposium*, 149–152, June 2008.
6. Inman, M. J. and A. Z. Elsherbeni, "Programming video cards for computational electromagnetics applications," *IEEE Antennas Propag. Mag.*, Vol. 47, 71–78, 2005.
7. Inman, M. J., A. Z. Elsherbeni, and C. E. Smith, "FDTD calculations using graphical processing units," *Proceedings of IEEE/ACES International Conference on Wireless Communications and Applied Computational Electromagnetics*, 728–731, Honolulu, HI, USA, 2005.
8. Peng, S. and Z. Nie, "Acceleration of the method of moments calculations by using graphics processing units," *IEEE Trans. Antennas Propag.*, Vol. 56, No. 7, 2130–2133, 2008.
9. 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.
10. Al-Sharkawy, M., 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*, PIER 57, 19–32, 2006.
11. Zainud-Deen, S. H., M. S. Ibrahim, and E. El-Deen, "A hybrid finite difference frequency domain and particle swarm optimization techniques for forward and inverse electromagnetic scattering problems," *The 23rd Annual Review of Progress in Applied Computational Electromagnetics*, 1575–1580, Verona, Italy, March 2007.
12. Zainud-Deen, S. H., E. El-Deen, and M. S. Ibrahim, "Electromagnetic scattering by conducting/dielectric objects," *The 23rd Annual Review of Progress in Applied Computational Electromagnetics*, 1866–1871, Verona, Italy, March 2007.
13. Zainud-Deen, S. H., A. Z. Botros, and M. S. Ibrahim, "Scattering from bodies coated with metamaterial using FDFD method," *Progress In Electromagnetics Research B*, Vol. 2, 279–290, 2008.
14. Zainud-Deen, S. H., W. M. Hassen, E. M. Ali, K. H. Awadalla,

and H. A. Sharshar, "Breast cancer detection using a hybrid finite difference frequency domain and particle swarm optimization techniques," *Progress In Electromagnetics Research B*, Vol. 3, 35–46, 2008.

15. Al-Sharkawy, M. H., V. Demir, and A. Z. Elsherbeni, *Electromagnetic Scattering Using the Iterative Multiregion Technique*, Morgain & Claypool Publishers, USA, 2007.
16. Vesely, B. F., "Iterative GPGPU linear solvers for sparse matrices," MSc. Thesis, Faculty of Electrical Engineering, Czech Technical University, Prague, 2008.
17. Yee, K. S., "Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media," *IEEE Trans. Antennas Propag.*, Vol. 14, 302–307, 1966.
18. Taflove, A., *Computational Electrodynamics: The Finite-difference Time-domain Method*, Artech House, Norwood, MA, USA, 2005.
19. Balanis, C. A., *Antenna Theory, Analysis and Design*, John Wiley & Sons, Inc., New York, 2005.
20. Xiao, G., J. Mao, and B. Yuan, "Generalized transition matrix for arbitrarily shaped scatterers or scatterer groups," *IEEE Trans. Antennas Propag.*, Vol. 56, No. 12, 3723–3732, 2008.
21. Chatterjee, A., J. M. Jin, and J. L. Volakis, "Edge-based finite elements and vector ABC's applied to 3-D scattering," *IEEE Trans. Antennas Propag.*, Vol. 41, No. 2, 221–226, 1993.