Hindawi Publishing Corporation
International Journal of Antennas and Propagation
Volume 2012, Article ID 280359, 6 pages
doi:10.1155/2012/280359

Research Article

The Case for Higher Computational Density in the Memory-Bound FDTD Method within Multicore Environments

Mohammed F. Hadi

Electrical Engineering Department, Kuwait University, P.O. Box 5969, Safat 13060, Kuwait

Correspondence should be addressed to Mohammed F. Hadi, mohammed.hadi@ku.edu.kw

Received 6 October 2011; Revised 27 January 2012; Accepted 31 January 2012

Academic Editor: Stefano Selleri

It is argued here that more accurate though more compute-intensive alternate algorithms to certain computational methods which are deemed too inefficient and wasteful when implemented within serial codes can be more efficient and cost-effective when implemented in parallel codes designed to run on today's multicore and many-core environments. This argument is most germane to methods that involve large data sets with relatively limited computational density—in other words, algorithms with small ratios of floating point operations to memory accesses. The examples chosen here to support this argument represent a variety of high-order finite-difference time-domain algorithms. It will be demonstrated that a three- to eightfold increase in floating-point operations due to higher-order finite-differences will translate to only two- to threefold increases in actual run times using either graphical or central processing units of today. It is hoped that this argument will convince researchers to revisit certain numerical techniques that have long been shelved and reevaluate them for multicore usability.

1. Introduction

General-purpose graphical processing units (GPUs) have emerged in recent years as a viable vehicle for high-performance computing. The emergence of a simplified programming language for GPU computing in the form of CUDA C along with dedicated GPU hardware specifically designed for high-end workstations as well as supercomputing platforms [1] was keenly received by the scientific computing community. A literature survey covering almost any field of scientific computing will reveal the unmistakable keen interest in porting codes from central processing units (CPUs) to GPUs. While the fastest CPUs today (e.g., Intel Xeon 8870) are capable of speeds of up to 200-giga-floating point operations per second (GFlops) in singleprecision [2], the fastest dedicated GPUs can deliver performance beyond the tera-Flops (TFlops) mark. Not all numerical methods, however, are created equal in terms of their abilities to approach the GPUs' theoretical performance limits. While some methods fit the GPUs' memory and computing model nicely, others require serious innovation to reshape them for optimum GPU use, whereas others still have to settle for a fraction of the theoretical performance limits.

The finite-difference time-domain (FDTD) method investigated here, which is a dominant method in the field of computational electromagnetics, belongs somewhere between the latter two categories. This is due to the fact that FDTD is a memory intensive algorithm with relatively low computational density (low ratio of floating-point operations applied on a set of data to number of read/write memory accesses of these data). In GPU computing, this can cause a serious performance penalty in the form of long idle times for the computing cores as they wait for data to process. Most of the efforts reported in the GPU/FDTD literature [3–13] were focused on how to best use the available GPU memory bandwidth through exploiting its on-chip shared and texture memory banks, as opposed to the much slower off-chip global memory. To a slightly lesser degree, this memory-bandwidth limitation issue applies to today's high-end CPUs as well.

An alternate route, however, is to turn the disadvantage of memory latency (slowness) into an advantage by asking

the GPU (or CPU) computing cores to perform more useful computations (instead of idling) on the same data they have at hand. This could be achieved by opting for high-order FDTD algorithms instead of the standard second-order algorithm. The extra computations required by these high-order algorithms go a long way to alleviate FDTD's biggest shortcoming when modeling electrically large problems (structures with dimensions much larger than the wavelength of interest) which is excessive accumulated phase errors in numerically propagated waves. There are several reported flavors of high-order FDTD algorithms [14–20] and they all share the advantage (with various degrees of success) of reduced numerical dispersion, which allows them to model larger electromagnetic structures with better accuracy compared to the standard method when using the same computing resources.

Four different high-order FDTD algorithms will be investigated in this work to determine their suitability to achieving a favorable balance between computing speed and memory speed. For a fair and straightforward comparison between the different algorithms, only algorithms with explicit update equations and applicable on Yee's standard cubical and staggered mesh are considered for the comparison. Shared memory use will not be considered since its use for high-order FDTD algorithms has not been reported in the literature yet. Indeed, this presentation is probably the first reported GPU implementation of a high-order FDTD algorithm. Furthermore, the GPU experiments will be conducted using CUDA Fortran [21] instead of CUDA C, which has not been reported in the FDTD literature either.

2. FDTD Update Equations

We begin by first sampling the update equations of the various FDTD algorithms which will be used to test the proposed hypothesis. Only one of the six field components will be detailed in this section for each algorithm. Furthermore, the various coefficients which are specific to each algorithm will not be detailed here to simplify the presentation. The object of this study, after all, is not to investigate the relative accuracy of the various algorithms. Rather, it is to test the relative efficiency in memory structure manipulations within CPUs and GPUs.

2.1. The Standard FDTD Algorithm. The standard FDTD algorithm [22] uses second-order finite differences to approximate Maxwell's differential equations which govern electromagnetic propagation and interaction with its surroundings. These finite-difference versions are rearranged as explicit update equations which are in turn evaluated iteratively as waves propagate within the numerical grid. A typical update equation (there are six total per time step) is given by

$$H_x\big|_{i,j,k}^{n+1/2} = D_{ax} H_x\big|_{i,j,k}^{n-1/2} + D_{bx}\begin{bmatrix} E_y\big|_{i,j,k+1/2}^{n} - E_y\big|_{i,j,k-1/2}^{n} \\ -E_z\big|_{i,j+1/2,k}^{n} + E_z\big|_{i,j-1/2,k}^{n} \end{bmatrix}, \quad (1)$$

where H_x is a typical magnetic field node within the FDTD grid and the E's are the surrounding electric field nodes. In this simplified form, the update equation involves 6 floating-point operations per invocation and requires 7 memory reads and one write. Note that some of the i, j, k space and n time indices within the discrete curl operator (all E terms) are not stepped. In the remaining update equations in this work, only stepped indices will be shown to simplify the expressions.

2.2. The FV24 Algorithm. The high-order FV24 algorithm is based on converting an integral form of Maxwell's equations into multiple weighted volume and surface integrals around the field node of interest before discretizing them [20]. Among all the other high-order FDTD algorithms, this particular one has the advantage of an almost completely developed suite of tools (see e.g. [23] and the references listed therein). The corresponding FV24 update equation for H_x is given by

$$\begin{aligned} H_x\big|_{i,j,k}^{n+1/2} = {} & D_{ax} H_x\big|_{i,j,k}^{n-1/2} \\ & + D_{bx1}\begin{bmatrix} E_y\big|_{k+1/2} - E_y\big|_{k-1/2} \\ -E_z\big|_{j+1/2} + E_z\big|_{j-1/2} \end{bmatrix} \\ & + D_{bx2}\begin{bmatrix} E_y\big|_{k+3/2} - E_y\big|_{k-3/2} \\ -E_z\big|_{j+3/2} + E_z\big|_{j-3/2} \end{bmatrix} \\ & + D_{bx3}\begin{bmatrix} E_y\big|_{j+1,k+3/2} + E_y\big|_{j-1,k+3/2} \\ +E_y\big|_{i+1,k+3/2} + E_y\big|_{i-1,k+3/2} \\ -E_y\big|_{j+1,k-3/2} - E_y\big|_{j-1,k-3/2} \\ -E_y\big|_{i+1,k-3/2} - E_y\big|_{i-1,k-3/2} \\ -E_z\big|_{i+1,j+3/2} - E_z\big|_{i-1,j+3/2} \\ -E_z\big|_{j+3/2,k+1} - E_z\big|_{j+3/2,k-1} \\ +E_z\big|_{i+1,j-3/2} + E_z\big|_{i-1,j-3/2} \\ +E_z\big|_{j-3/2,k+1} + E_z\big|_{j-3/2,k-1} \end{bmatrix} \\ & + D_{bx4}\begin{bmatrix} E_y\big|_{i+1,j+1,k+3/2} + E_y\big|_{i-1,j+1,k+3/2} \\ +E_y\big|_{i+1,j-1,k+3/2} + E_y\big|_{i-1,j-1,k+3/2} \\ -E_y\big|_{i+1,j+1,k-3/2} - E_y\big|_{i-1,j+1,k-3/2} \\ -E_y\big|_{i+1,j-1,k-3/2} - E_y\big|_{i-1,j-1,k-3/2} \\ -E_z\big|_{i+1,j+3/2,k+1} - E_z\big|_{i-1,j+3/2,k+1} \\ -E_z\big|_{i+1,j+3/2,k-1} - E_z\big|_{i-1,j+3/2,k-1} \\ +E_z\big|_{i+1,j-3/2,k+1} + E_z\big|_{i-1,j-3/2,k+1} \\ +E_z\big|_{i+1,j-3/2,k-1} + E_z\big|_{i-1,j-3/2,k-1} \end{bmatrix}. \end{aligned} \quad (2)$$

Table 1: Comparison of the theoretical computational costs for the various FDTD algorithms.

Algorithm	Floats	Reads	Writes	Floats/FDTD floats
FDTD	6	7	1	1
FV24	45	46	1	7.5
FV24-WB	28	29	1	4.7
S{2, 2}	45	46	1	7.5
S44	17	18	1	2.8

This update equation has 45 floating-point operations and involves 42 reads and one write.

2.3. The FV24-WB Algorithm. The terms multiplied by the D_{bx3} coefficient in the above equation are mainly needed to achieve extreme phase accuracy for a narrow band around the design frequency [20]. If wideband analyses are of interest then setting D_{bx3} to zero will simplify the FV24 algorithm without sacrificing overall accuracy across the frequency bandwidth of interest. The resulting, FV24-WB, update equation will have 28 floating-point operations and involves 29 reads and one write.

2.4. The S{2, 2} Algorithm. This algorithm is selected from a class of high-order FDTD algorithms developed by Zygiridis and Tsiboukis [19] and has the same number of floating-point operations and memory reads and writes as the FV24 algorithm. This particular version is selected, however, to illustrate the effect of different memory access patterns between otherwise identical algorithms from a computational point of view. The corresponding update equation for the balanced S{2, 2} algorithm is given by

$$\begin{aligned} H_x|_{i,j,k}^{n+1/2} &= D_{ax}\, H_x|_{i,j,k}^{n-1/2} \\ &+ D_{bx1}\begin{bmatrix} E_y\big|_{k+1/2} - E_y\big|_{k-1/2} \\ -E_z|_{j+1/2} + E_z|_{j-1/2} \end{bmatrix} \\ &+ D_{bx2}\begin{bmatrix} E_y\big|_{k+3/2} - E_y\big|_{k-3/2} \\ -E_z|_{j+3/2} + E_z|_{j-3/2} \end{bmatrix} \\ &+ D_{bx3}\begin{bmatrix} E_y\big|_{j+1,k+1/2} + E_y\big|_{j-1,k+1/2} \\ +E_y\big|_{i+1,k+1/2} + E_y\big|_{i-1,k+1/2} \\ -E_y\big|_{j+1,k-1/2} - E_y\big|_{j-1,k-1/2} \\ -E_y\big|_{i+1,k-1/2} - E_y\big|_{i-1,k-1/2} \\ -E_z|_{j-1/2,k+1} - E_z|_{j-1/2,k-1} \\ -E_z|_{i+1,j-1/2} - E_z|_{i-1,j-1/2} \\ +E_z|_{j+1/2,k+1} + E_z|_{j+1/2,k-1} \\ +E_z|_{i+1,j+1/2} + E_z|_{i-1,j+1/2} \end{bmatrix} \\ &+ D_{bx4}\begin{bmatrix} E_y\big|_{j+1,k+3/2} + E_y\big|_{j-1,k+3/2} \\ +E_y\big|_{i+1,k+3/2} + E_y\big|_{i-1,k+3/2} \\ -E_y\big|_{j+1,k-3/2} - E_y\big|_{j-1,k-3/2} \\ -E_y\big|_{i+1,k-3/2} - E_y\big|_{i-1,k-3/2} \\ -E_z|_{j-3/2,k+1} - E_z|_{j-3/2,k-1} \\ -E_z|_{i+1,j-3/2} - E_z|_{i-1,j-3/2} \\ +E_z|_{j+3/2,k+1} + E_z|_{j+3/2,k-1} \\ +E_z|_{i+1,j+3/2} + E_z|_{i-1,j+3/2} \end{bmatrix}. \end{aligned} \tag{3}$$

2.5. The S44 Algorithm. This algorithm is based on straightforward fourth-order central finite-differences in space. It also differs from the other algorithms in that it has backward fourth-order finite differences in time as suggested by Hwang and Ihm [18]. The corresponding update equation is given by

$$\begin{aligned} H_x|_{i,j,k}^{n+1/2} &= D_{ax1}\, H_x|_{i,j,k}^{n-1/2} + D_{ax2}\, H_x|_{i,j,k}^{n-3/2} \\ &+ D_{ax3}\, H_x|_{i,j,k}^{n-5/2} + D_{ax4}\, H_x|_{i,j,k}^{n-7/2} \\ &+ D_{bx1}\begin{bmatrix} E_y\big|_{k+1/2} - E_y\big|_{k-1/2} \\ -E_z|_{j+1/2} + E_z|_{j-1/2} \end{bmatrix} \\ &+ D_{bx2}\begin{bmatrix} E_y\big|_{k+3/2} - E_y\big|_{k-3/2} \\ -E_z|_{j+3/2} + E_z|_{j-3/2} \end{bmatrix}. \end{aligned} \tag{4}$$

This update equation has 17 floating-point operations with 18 memory reads and one write.

Table 1 summarizes the computational costs of the update equations listed in this section. In listing the memory reads column, in particular, all the D_{ax} and D_{bx} coefficients were assumed position dependent to accommodate modeling inhomogeneous domains. From the table, it is obvious that the high-order algorithms are handicapped by as much as 7.5-fold increase in computational intensity and up to 5.9-fold increase in combined (read/write) memory accesses.

3. CUDA Fortran Implementation

It is somewhat surprising that there is yet to appear in the literature an FDTD GPU implementation using CUDA Fortran. All the relevant references mentioned at the end of this work, for example, unanimously used CUDA C. To add value to this presentation, CUDA Fortan will be used instead, which is a product developed by the Portland Group [21]. It would be the obvious choice for Fortran programmers as it affords most of the functionality of CUDA C, coupled with the simplicity and efficiency of Fortran's multidimensional array management. A glaring omission from CUDA Fortran at this time is the lack of texture memory accessibility, though it is expected to be implemented in due course.

The GPU kernel that computes the standard FDTD update equation can be written as

```
attributes (global) subroutine
Yee_kernel(Hx, Ey, Ez, dax, dbx, x1, x2,
y1, y2, z1, z2, Bx, By, Bz, II, JJ, KK)
integer, value:: II, JJ, KK
real, dimension (II+1, JJ, KK):: Hx, dax,
db1x
real, dimension (II+1, JJ, KK+1):: Ey
real, dimension (II+1, JJ+1, KK):: Ez
integer:: i, j, k
integer, value:: x1, x2, y1, y2, z1, z2,
Bx, By, Bz
i = threadidx%x + (blockidx%x - 1) *
blockdim%x
j = threadidx%y + (blockidx%y - 1) *
blockdim%y - ((blockidx%y - 1)/By) *
By * blockdim%y
k = threadidx%z + ((blockidx%y -
1)/By) * blockdim%z - ((blockidx%y
- 1)/(By * Bz)) * Bz * blockdim%z
if (i >= x1.and. I <= x2) then
if (j >= y1.and. j <= y2) then
if (k >= z1.and. k <= z2) then
Hx(i, j, k) = dax(i, j, k) * Hx(i, j,
k) + dbx(i, j, k) * (Ey(i, j, k + 1) -
Ey(i, j, k) - Ez(i, j + 1, k) + Ez(i, j,
k))
end if
end if
end if
end subroutine Yee_kernel
```

where x_1, x_2, y_1, y_2, z_1, and z_2 are the overall FDTD grid boundaries, B_x, B_y, B_z define the number of thread-blocks along the three dimensions of this grid, and i, j, k represent the H_x mapped position within it. The remaining algorithm's kernels have similar forms and differ only in terms of the actual update equation within the nested if statements and the requisite additional coefficients array declarations. A close look at the above kernel would reveal the absence of the requisite nested for loops in typical CPU codes. The reason behind that is the way GPU kernels work. In GPU kernels, scheduling is performed by the hardware which handles the division of the array elements among the GPU's many cores. All the user needs to do is specify the array bounds and the thread-to-data mapping structure.

The calling routine for the above kernel from the main program is given by

```
call Yee_kernel <<<dimGridYee,
dimBlockYee>>>
(Hxdev, Eydev, Ezdev, daxdev, dbxdev,
P1, IP, P1, JP, P1, KP, BYeex, BYeey,
BYeez, II, JJ, KK)
```

where dimGridHx and dimBlockHx are special variables that define the configuration of the kernel launch which decides how the overall grid is dissected into optimum subdomain sizes for best GPU performance. In particular, dimBlockHx decides the (up to) 3-dimensional size of the kernel's thread-block which directly affects its efficiency. Many factors influence the thread-block dimensions choice. Some are related to the available GPU resources such as number of computing cores and their groupings as well as memory architecture and bandwidth. Other factors are related to the algorithm being computed and its requirements of register space and shared memory. Other factors still are related to the programming language being used as CUDA C and CUDA Fortran follow their parent languages in the manner of one being row major (C) and the other being column major (Fortran).

Table 2: GPU kernel throughput for the standard FDTD algorithm at several thread-block size configurations.

$B_x \times B_y \times B_z$	MCells/s
$4 \times 4 \times 4$	177
$8 \times 8 \times 8$	221
$16 \times 8 \times 4$	301
$16 \times 8 \times 8$	249
$32 \times 8 \times 4$	350

Table 2 demonstrates the standard FDTD throughput in MCells/s at different thread-block size configurations. This metric represents the number of millions of FDTD cells that are updated per second which are computed as follows:

$$\frac{1}{6} \times \frac{\text{No of FDTD cells in millions}}{\text{Simulation time in seconds}} \times \text{No of time steps}. \tag{5}$$

The 1/6 factor is there to highlight the fact that this and subsequent simulation results are based on single and independent field-component kernel timings and not combined components FDTD simulations. With a few extra trials it was found that the optimum thread-block size for this particular algorithm is $32 \times 2 \times 3$. The 32 number is significant in that it fits the GPU access pattern of its global memory (The latest generation GPUs (compute capability 2.0 or later) access their global memory in groups of 32 4-byte floats instead of 16 [24]. For optimum results, each thread should read 32 consecutively stored numbers at a time from memory—which is along the first dimension of a Fortran array. Earlier generation GPUs allowed group accesses of multiples of 16 4-byte floats). As can be seen from the table, deciding the optimum thread-block size is critical to the kernel performance. Table 3 lists the optimum thread-block dimensions for each of the tested algorithms.

Table 3: Optimum thread-block configurations for the various FDTD algorithms.

Algorithm	B_x	B_y	B_z
FDTD	32	2	3
FV24	32	3	2
FV24-WB	32	2	2
S{2, 2}	32	2	2
S44	32	2	3

4. CUDA C Implementation

For those not familiar with Fortran, the following is the CUDA C version of the standard FDTD GPU kernel. The only difference from the previously listed CUDA Fortran kernel is the fact that the thread-block used in this code is one-dimensional

```
__global__ void Yee_kernel(float * dHx,
float * dEy,
float * dEz, float * ddax, float *
ddbx, int x1, int x2, int y1, int y2,
int z1, int z2, int sizeOfData) {
int t1, xIndex, yIndex, zIndex, EyIdx,
EzIdx;
int idx = blockDim.x * (gridDim.x *
blockIdx.y + blockIdx.x) + threadIdx.x;
if (idx < sizeOfData) {
t1 = floorf(idx/K1);
zIndex = idx-t1 * K1;
xIndex = floorf(t1/J1);
yIndex = t1-xIndex * J1;
EyIdx = zIndex + yIndex * (K + 2) +
xIndex * J1 * (K + 2);
EzIdx = zIndex + yIndex * (K + 1) +
xIndex * (J + 2) * (K + 1);
if((xIndex >= x1)&&(xIndex <= x2)) {
if((yIndex >= y1)&&(yIndex <= y2)) {
if((zIndex >= z1)&&(zIndex <= z2)) {
dHx [idx]= ddax [idx] * dHx [idx]
+ ddb1x [idx] * (dEy [EyIdx + 1] -
dEy [EyIdx] - dEz [EzIdx + K1] + dEz
[EzIdx]);
}}}}}
```

This code is only listed for direct comparison's sake between CUDA Fortran and CUDA C. All the results shown in the next section are based on the CUDA Fortran kernels.

5. Runtime Comparisons

Each of the five developed kernels is tasked with updating every H_z node in a 300 × 300 × 300 FDTD grid. The kernels were run 1,000 times each and the corresponding runtimes were averaged out. CUDA Fortran's `cudaEventElapsedTime` and related runtime API's were used for collecting run times and data were converted to units of MCells/s as explained earlier. The GPU used for the comparison is an nVidia Tesla C2050.

Table 4: Comparison of GPU kernels' throughputs for the various FDTD algorithms.

Algorithm	Throughput (MCells/s)	FDTD/Algorithm throughput
FDTD	383	1
FV24	159	2.4
FV24-WB	198	1.9
S{2, 2}	144	2.7
S44	215	1.8

Table 5: Comparison of CPU codes' throughputs for the various FDTD algorithms.

Algorithm	Throughput (MCells/s)	FDTD/Algorithm throughput
FDTD	75	1
FV24	32	2.3
FV24-WB	35	2.1
S{2, 2}	26	2.9
S44	26	2.9

Table 4 compares the GPU throughput for the five tested FDTD algorithms. As expected, the standard FDTD algorithm has the highest throughput at 383 MCells/s. For the record, the CPU-only version of the code managed a 75 MCells/s which translates into a GPU speed-up of nearly 5 : 1. What is important, however, is the fact that an algorithm such as the FV24 which has 7.5 times the computational density of the standard algorithm managed a 198 MCells/s throughput which is only 198/383 ≃ 1/2 as slow. This translates into the FV24 algorithm being nearly 4 times faster than predicted by a direct floating-point operations count. This result is even more impressive considering that the FV24 algorithm requires in theory nearly 6 times more memory accesses than the standard FDTD algorithm (see Table 1).

This behavior is by no means limited to GPU computing. Today's CPUs are also guilty of outpacing memory bandwidth. Running the CPU versions of the above discussed algorithms on a single core of a Xeon W5580 processor resulted in the throughputs listed in Table 5. Based on these results, and whether CPUs or GPUs are used, it can be argued that using high-order FDTD algorithms that run only twice as slow as the standard method can be worth the main advantage of such algorithms. For example, it has been reported in [20] that the FV24 algorithm is capable of modeling wave propagation for distances that are 3,000 times longer than in the standard method while accumulating the same overall phase error at the grid resolution of 20 cells per wavelength.

6. Conclusion

The main objective of this paper is to drive the point that there might be a need to reevaluate numerical algorithms that have long been considered to be computationally inefficient, in light of the unique nature of today's CPU and GPU computing architectures and programming models. It is possible, especially for memory bandwidth-bound algorithms, that memory latency can hide much of the computational inefficiency within the framework of modern processors. In such situations, such seemingly inefficient algorithms, if they are otherwise of value, might become viable again for certain applications. This possibility is by no means limited to FDTD or to computational electromagnetics. Virtually any memory bandwidth-bound numerical method can potentially benefit from these findings.

Acknowledgment

This work was supported by Kuwait University Research Grant no. EE02/07.

References

[1] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: a unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008.

[2] http://download.intel.com/support/processors/xeon/sb/xeon_E7-8800.pdf.

[3] V. Demir and A. Z. Elsherbeni, "Compute unified device architecture (CUDA) based finite-difference time-domain (FDTD) implementation," *Applied Computational Electromagnetics Society Journal*, vol. 25, no. 4, pp. 303–314, 2010.

[4] M. R. Zunoubi, J. Payne, and W. P. Roach, "CUDA implementation of TEZ-FDTD solution of Maxwell's equations in dispersive media," *IEEE Antennas and Wireless Propagation Letters*, vol. 9, Article ID 5512659, pp. 756–759, 2010.

[5] K. Xu, Z. H. Fan, D. Z. Ding, and R. S. Chen, "GPU accelerated unconditionally stable Crank-Nicolson FDTD method for the analysis of three-dimensional microwave circuits," *Progress in Electromagnetics Research*, vol. 102, pp. 381–395, 2010.

[6] M. Weldon, L. Maxwell, D. Cyca, M. Hughes, C. Whelan, and M. Okoniewski, "A practical look at GPU-accelerated FDTD performance," *Applied Computational Electromagnetics Society Journal*, vol. 25, no. 4, pp. 315–322, 2010.

[7] S. Adams, J. Payne, and R. Boppana, "Finite difference time domain (FDTD) simulations using graphics processors," in *Proceedings of the HPCMP Users Group Conference, High Performance Computing Modernization Program: A Bridge to Future Defense (DoD HPCMP UGC '07)*, pp. 334–338, Pittsburgh, Pa, USA, June 2007.

[8] A. Valcarce, G. De La Roche, and J. Zhang, "A GPU approach to FDTD for radio coverage prediction," in *Proceedings of the 11th IEEE Singapore International Conference on Communication Systems (ICCS '08)*, pp. 1585–1590, Guangzhou, Singapore, November 2008.

[9] P. Sypek, A. Dziekonski, and M. Mrozowski, "How to render FDTD computations more effective using a graphics accelerator," *IEEE Transactions on Magnetics*, vol. 45, no. 3, Article ID 4787427, pp. 1324–1327, 2009.

[10] A. Balevic, L. Rockstroh, W. Li et al., "Acceleration of a finite-difference method with general purpose GPUs—lesson learned," in *Proceedings of the 8th IEEE International Conference on Computer and Information Technology (CIT '08)*, pp. 291–294, Sidney, Australia, July 2008.

[11] A. Balevic, L. Rockstroh, A. Tausendfreund, S. Patzelt, G. Goch, and S. Simon, "Accelerating simulations of light scattering based on finite-difference time-domain method with general purpose GPUs," in *Proceedings of the 11th International Conference on Computational Science and Engineering (CSE '08)*, pp. 327–334, 2008.

[12] V. Demir, "A stacking scheme to improve the efficiency of finite-difference time-domain solutions on graphics processing units," *Applied Computational Electromagnetics Society Journal*, vol. 25, no. 4, pp. 323–330, 2010.

[13] N. Takada, T. Shimobaba, N. Masuda, and T. Ito, "Improved performance of FDTD computation using a thread block constructed as a two-dimensional array with CUDA," *Applied Computational Electromagnetics Society Journal*, vol. 25, no. 12, pp. 1061–1069, 2010.

[14] E. Turkel and A. Yefet, "Fourth order method for Maxwell equations on a staggered mesh," in *Proceedings of the International Symposium of IEEE Antennas and Propagation Society*, pp. 2156–2159, July 1997.

[15] J. B. Cole, "A high-accuracy realization of the yee algorithm using non-standard finite differences," *IEEE Transactions on Microwave Theory and Techniques*, vol. 45, no. 6, pp. 991–996, 1997.

[16] Y. W. Cheong, Y. M. Lee, K. H. Ra, J. G. Kang, and C. C. Shin, "Wavelet-Galerkin scheme of time-dependent inhomogeneous electromagnetic problems," *IEEE Microwave and Wireless Components Letters*, vol. 9, no. 8, pp. 297–299, 1999.

[17] E. A. Forgy and W. C. Chew, "A time-domain method with isotropic dispersion and increased stability on an overlapped lattice," *IEEE Transactions on Antennas and Propagation*, vol. 50, no. 7, pp. 983–996, 2002.

[18] K. P. Hwang and J. Y. Ihm, "A stable fourth-order FDTD method for modeling electrically long dielectric waveguides," *Journal of Lightwave Technology*, vol. 24, no. 2, pp. 1048–1056, 2006.

[19] T. T. Zygiridis and T. D. Tsiboukis, "Optimized three-dimensional FDTD discretizations of Maxwell's equations on Cartesian grids," *Journal of Computational Physics*, vol. 226, no. 2, pp. 2372–2388, 2007.

[20] M. F. Hadi, "A finite volumes-based 3-D low dispersion FDTD algorithm," *IEEE Transactions on Antennas and Propagation*, vol. 55, no. 8, pp. 2287–2293, 2007.

[21] PGI Group, "CUDA Fortran Programming Guide and Reference," 2011.

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

[23] M. F. Hadi, "Near-field PML optimization for low and high order FDTD algorithms using closed-form predictive equations," *IEEE Transactions on Antennas and Propagation*, vol. 59, no. 8, pp. 2933–2942, 2011.

[24] Nvidia Corporation, CUDA C Best Practices Guide, 2011.