David B. Davidson
Dept. E&E Engineering
University of Stellenbosch
Stellenbosch 7600, South Africa
Tel: +27 21 808 4458;
Fax: +27 21 808 4981
E-mail: davidson@sun.ac.za

Foreword by the Editor

Two recent columns have considered the use of general-purpose graphical processing units (GPUs) for computational electromagnetics: the June 2010 column looked specifically at the FDTD method, and the December 2010 column addressed acceleration of the MoM. Progress in this field is rapid as new hardware becomes available. This month's column revisits the FDTD method, providing a comprehensive review of various current approaches to an FDTD *CUDA* implementation. Two of this month's co-authors also contributed to the June 2012 column on coding the FDTD on more conventional x86 CPUs. As always, we thank the authors for their contributions.

Development of a *CUDA* Implementation of the 3D FDTD Method

Matthew Livesey[1], James Francis Stack, Jr.[2], Fumie Costen[3], Takeshi Nanri[4], Norimasa Nakashima[5], and Seiji Fujino[6]

[1]Accenture
Kingsley Hall, 20 Bailey Lane, Manchester Airport, Manchester, M90 4AN, UK
Tel: +44 161 435 5865; E-mail: matthew.livesey@accenture.com

[2]Remcom, Inc.
315 S. Allen Street, Suite #416
State College, PA, 16823, USA
Tel: +1-814-861-1299; E-mail: James.Stack@remcom.com

[3]School of Electrical and Electronic Engineering
The University of Manchester
Sackville Street Building, Sackville Street, Manchester, M13 9PL UK
Bio-research Infrastructure Construction Team, Advanced Technology Support Division
Advanced Science Institute
RIKEN 2-1 Hirosawa, Wako, Saitama 351-0198, Japan
Tel: +44-161-306-4717; E-mail: fumie.costen@manchester.ac.uk

[4]Research Institute for Information Technology
Kyushu University
6-10-1 Hakozaki, Higashi-ku, Fukuoka, 812-8581, Japan
Tel: +81-92-642-2298; E-mail:nanri@cc.kyushu-u.ac.jp

[5]Faculty of Information Engineering
Fukuoka Institute of Technology
3-30-1 Wajiro-higashi, Higashi-ku, Fukuoka, 811-0295, Japan
Tel: +81-92-606-4968; E-mail: n-nakashima@fit.ac.jp

[6]Research Institute for Information Technology
Kyushu University
6-10-1 Hakozaki, Higashi-ku, Fukuoka, 812-8581, Japan
Tel: +81-92-642-3837; E-mail: fujino@cc.kyushu-u.ac.jp

Abstract

The use of general-purpose computing on a GPU is an effective way to accelerate the FDTD method. This paper introduces flexibility to the theoretically best available approach. It examines the performance on both Tesla- and Fermi-architecture GPUs, and identifies the best way to determine the GPU parameters for the proposed method.

Keywords: Finite difference methods; time domain analysis; hardware; acceleration; high performance computing; parallel programming; parallel architectures; GPU; graphical processing unit

1. Introduction

The use of general-purpose computing on graphics processing units (GPGPU) to execute scientific computations is becoming increasing prevalent. GPGPU is particularly suitable for executing problems with a high degree of data parallelism, in order to make use of the many processing units present on a typical GPU. The Finite-Difference Time-Domain (FDTD) method is popular because it directly solves Maxwell's curl equations with a minimal set of assumptions, thus providing a robust, straightforward method. The FDTD method is characterized by tens or hundreds of thousands of time-step iterations over large amounts of data, organized into multidimensional arrays. Due to significant data independence among the calculations performed for each element in each array at each time step, the FDTD method exhibits a large degree of parallelism. Several GPGPU implementations of the FDTD method have been presented in research to date using NVIDIA's *CUDA* technology. Here, we demonstrate how differences in configuration affect the performance of a *CUDA*-based FDTD implementation. Differences in memory-access patterns, single versus double-precision arithmetic, and differences in hardware generation are considered. Section 2 briefly introduces the characteristics of the FDTD method. Section 3 presents the architecture of GPGPU technology, and introduces its key aspects that may affect the speed of computation, in general. Section 4 summarizes the currently available methods for implementing the FDTD method on GPU hardware. Since this paper is concerned only with single GPU boards, work which focuses on the utilization of multiple GPU boards for the FDTD computation are not included. Section 5 details our approach to implementing the FDTD method on a GPU board. Section 6 presents our various numerical experiments, and analyses the results to understand the optimum condition under which our approach should be used. Section 7 concludes the paper, summarizing the key outcomes of our work.

2. Overview of the FDTD Method

Maxwell's curl equations for free space without sources are written as

$$\nabla \times \mathbf{H} = \varepsilon \frac{\partial \mathbf{E}}{\partial t}, \qquad \nabla \times \mathbf{E} = -\mu \frac{\partial \mathbf{H}}{\partial t}, \tag{1}$$

where $\mathbf{E}$, $\mathbf{H}$, ε, and μ are the electric and magnetic fields, and the permittivity, and permeability of free space, respectively. Application of the central finite-difference approximation to both the space and time derivatives of Equation (1) yields the discretized equations:

$$\begin{aligned} H_y^{n+\frac{1}{2}}(i,j,k) = {} & H_y^{n-\frac{1}{2}}(i,j,k) \\ & + \frac{\Delta t}{\mu \Delta x}\left[E_z^n\left(i+\frac{1}{2},j,k\right) - E_z^n\left(i-\frac{1}{2},j,k\right)\right] \\ & - \frac{\Delta t}{\mu \Delta z}\left[E_x^n\left(i,j,k+\frac{1}{2}\right) - E_x^n\left(i,j,k-\frac{1}{2}\right)\right], \end{aligned} \tag{2}$$

where Δt, Δx, and Δz are the temporal step, the spatial step in the x direction, and the spatial step in the z direction, respectively. [1] presents the rest of the core equations. In a three-dimensional implementation of the FDTD method, **E** and **H** are vectorized into six three-dimensional arrays, denoted E_x, E_y, E_z, H_x, H_y, and H_z. For a single time step, each element in each array must be calculated using the core equations. Each calculation in E_x, E_y, and E_z is independent from the others, and the same is true of those in H_x, H_y, and H_z. Also, as is seen in Equation (2), the computation is spatially localized to the one-cell neighbors. The FDTD method is thus well known to be suitable for parallel computing.

3. General-Purpose GPU Computing with *CUDA*

NVIDIA's Tesla Architecture introduced a GPU hardware design consisting of an array of general-purpose streaming multiprocessors. The architecture of a Tesla GPU is shown in Figure 1 [2]. In conjunction, NVIDIA introduced the Compute Unified Device Architecture programming model (*CUDA*) [2-4]. *CUDA* allows a programmer to specify which sections of a computation (described as "kernels") should be executed on the GPU [2]. Computations suitable for execution on a GPU are characterized by massive parallelism. A kernel operates over a grid of threads, where a grid is divided into blocks [5] in either one or two dimensions. Each block is further divided into threads in one, two, or three dimensions [3]. Figure 2 shows the hierarchical organization of grids, blocks, and threads. Within one kernel, a thread is identified by its indices at both the block and thread level. The built-in parameters `blockIdx.x` and `blockIdx.y` (and `blockIdx.z` for Fermi) determine to which block a thread belongs, while `threadIdx.x`, `threadIdx.y`, and `threadIdx.z` determine the thread's position within a block. The threads within one block are executed simultaneously on a single streaming multiprocessor (labeled SM in Figure 1). Threads within a block can synchronize with each other and share data via the shared memory on each streaming multiprocessor. However, there is no guarantee of the execution order of each block within a grid, so communication and synchronization between threads in different blocks is not permitted, although all threads can be synchronized outside kernel execution. Making effective use of the GPU hardware therefore requires identification of a large number of independent calculations within a computation that can be executed as a kernel on the GPU. The global memory (shown as DRAM in Figure 1) of the Tesla architecture is large, and is accessible to all threads in all blocks in a grid. The memory bandwidth in Tesla T10 and Tesla M2050 are 32 GB/s and 148 GB/s, respectively. On the other hand, the bandwidth in the AMD Athlon 64×2 5600+ is about 9 GB/s. The global memory thus has more than four times wider bandwidth than a modern PC, but will limit performance if used too extensively [4]. It is possible to optimize the performance of access to global memory through "memory coalescing" [4]. If simultaneously executing threads within a

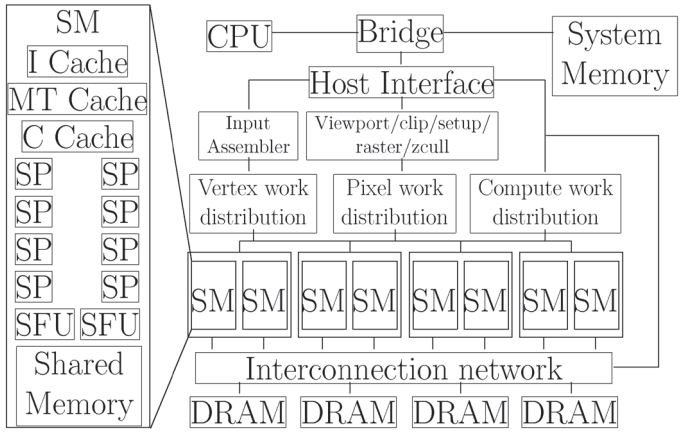


Figure 1. The Tesla unified graphics and computing GPU architecture [2].

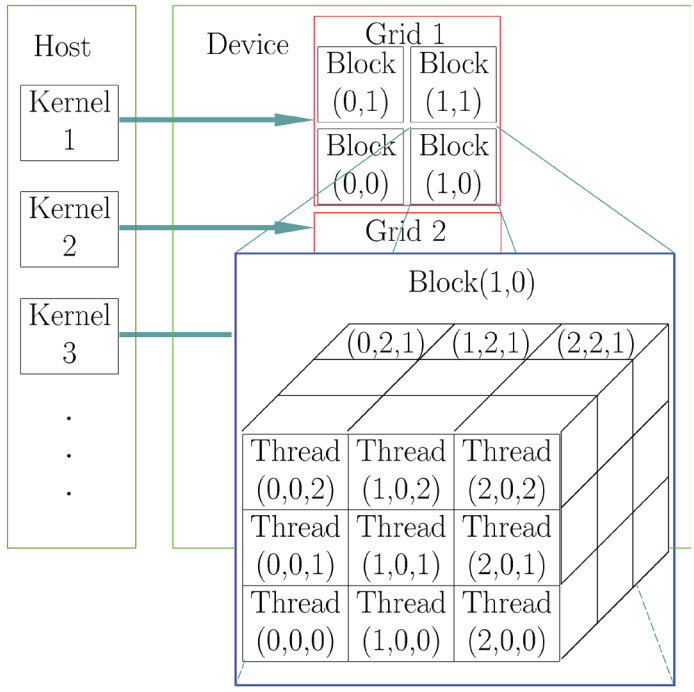


Figure 2. The *CUDA* grid organization in Tesla [4].

block access consecutive memory locations, the requests are combined into a single-larger memory fetch. Fetching a large consecutive block of memory results in more efficient use of the memory's bandwidth [4]. The streaming processor cores (labeled SP in Figure 1) within each streaming multiprocessor in the earliest Tesla GPUs only supported single-precision arithmetic [6]. A revision to the architecture introduced double-precision support, but with far lower arithmetic performance [6]. A more-recent major revision to the architecture, named Fermi, introduced much more comprehensive double-precision support [6]. Fermi also introduced improvements in the memory system. The shared memory can be configured as partially user programmable and partially a level 1 cache [6]. A level 2 cache to global memory, unified across all streaming multiprocessors, was also introduced [6]. This paper examines

the most-efficient approach to the implementation of the three-dimensional FDTD method on the GPGPU. The paper studies the FDTD's performance on both Tesla- and Fermi-architecture hardware in both single and double precision, to understand the best way to run the proposed approach.

4. Existing Implementations of the FDTD Method on GPU Hardware

Several implementations of the three-dimensional FDTD method using *CUDA* have been published. Each of these exhibits one of three domain-decomposition approaches to mapping the x, y, and z dimensions of the FDTD space to the allocation of blocks and threads.

4.1 First Approach

The three-dimensional FDTD space is broken down into two-dimensional planes. Figure 3 depicts the decomposition of a 6×4×5-dimensional FDTD space. The FDTD equations for a particular plane are solved in parallel using a kernel on the GPU, but each plane is computed by its own kernel invocation, sequentially to the others, within a single time step. Within each plane, the elements are divided into blocks in two dimensions, so that each block has the standard *CUDA* parameters of `blockIdx.x` and `blockIdx.y`. Each block is further broken down into threads in two dimensions, so that each thread has the standard *CUDA* parameters of `threadIdx.x` and `threadIdx.y`. Figure 4 presents this method of decomposition for a plane within the three-dimensional FDTD space shown in Figure 3. Each thread is responsible for a single cell of the FDTD space, and can calculate its i and j coordinates as `blockIdx.x * blockDim.x + threadIdx.x` and `blockIdx.y * blockDim.y + threadIdx.y`, respectively. While this method allows very fine-grained decomposition to expose all of the parallelism within a single plane, it does not exploit the full concurrency of the algorithm, since all the calculations in all planes are independent from each other. This may or may not limit performance, depending on whether the parallelism exposed is enough to keep the cores of the GPU fully occupied during execution. This approach was used in [7] and [8].

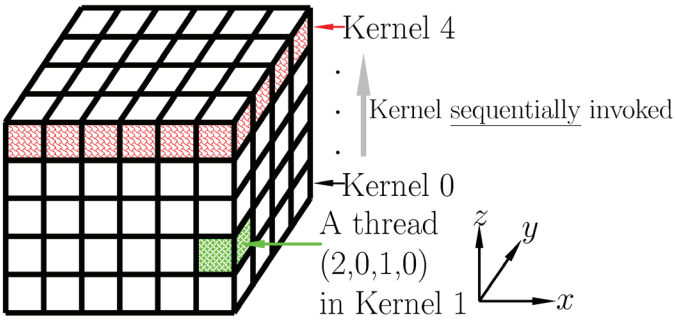


Figure 3. The domain decomposition of the three-dimensional FDTD space in the first approach. (a,b,c,d) means (`blockIdx.x`, `blockIdx.y`, `threadIdx.x`, `threadIdx.y`).

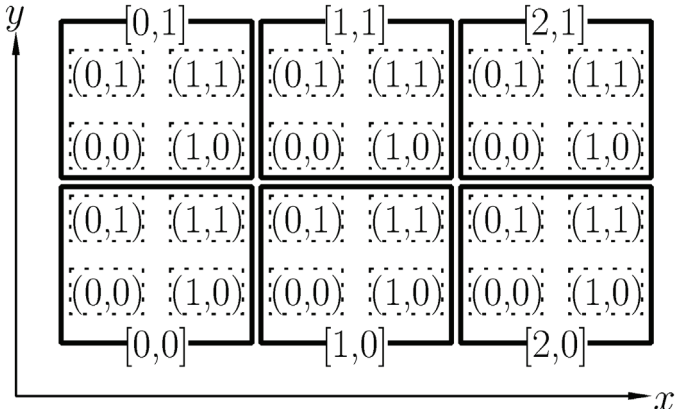


Figure 4. Two-dimensional decomposition using blocks and threads on a $k = constant$ plane in the first and second approaches. $[a,b]$ and (a,b) mean [`blockIdx.x`, `blockIdx.y`] and (`threadIdx.x`, `threadIdx.y`), respectively.

4.2 Second Approach

The second approach addresses the entire three-dimensional space within a single kernel invocation. As with the first approach, blocks and threads are used to produce a two-dimensional decomposition matched to the x and y dimensions of the FDTD space, as is shown in Figure 4. However, in this case, each thread executes all of the elements in the z direction in a `for` loop, as is shown in Figure 5. This means that each thread does much more work than in the first approach and many fewer kernel instances are required, but it does not alter the amount of parallelism exposed at any one point in time. Each approach has a maximum of $N_x \times N_y$ threads, where N_x and N_y are the size of the FDTD space in the x and y dimensions. This approach was demonstrated in [9], and was also used in [10].

4.3 Third Approach

The third approach applies a fundamentally different method to domain decomposition. As is the second approach, the entire three-dimensional space is addressed within a single kernel invocation. Blocks are allocated in two dimensions, and the x and y indices of each block are mapped to two of the x, y, and z dimensions of the FDTD space. Threads within each block are allocated in one dimension, as shown in Figure 6, and the x index of the thread is mapped to the remaining dimension of the FDTD space. One thread can now be allocated to every

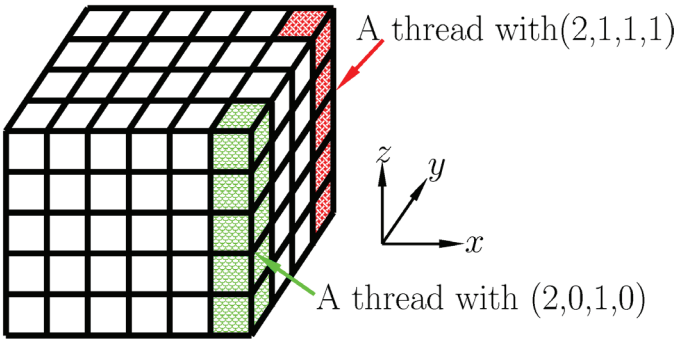


Figure 5. The domain decomposition of the three-dimensional FDTD space in the second approach. (a,b,c,d) means (`blockIdx.x`, `blockIdx.y`, `threadIdx.x`, `threadIdx.y`).

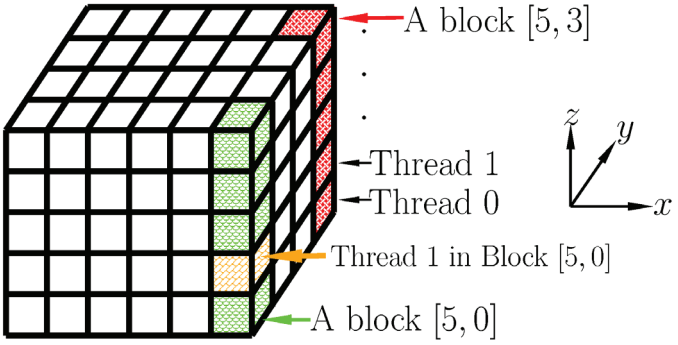


Figure 6. The domain decomposition of the three-dimensional FDTD space in the third approach. $[a,b]$ means [`blockIdx.x`, `blockIdx.y`].

```
dim3 blocks(DIMY,DIMZ);
dim3 threads(THREAD_COUNT);
//...
eKernel<<<blocks,threads>>>(dev_ex,
dev_ey,dev_ez,dev_hx,dev_hy,dev_hz,dx,dy,dz,pmt,dt);
//...
hKernel<<<blocks,threads>>>(dev_ex,
dev_ey,dev_ez,dev_hx,dev_hy,dev_hz,dx,dy,dz,pma,dt);
```

Figure 7. Launching kernels with multiple blocks and multiple threads.

cell in the FDTD space, exposing the maximum available parallelism in the algorithm. Whether or not all these threads actually execute simultaneously depends on the number of streaming multiprocessors in the hardware, and this will in part determine whether the exposure of additional parallelism provides any performance benefit over the other approaches. This method of decomposition was demonstrated in [11]. In [12], only one dimension was used for block allocation, but this was mapped to both *i* and *j* in such a way that the execution was equivalent to this approach. The two dimensions of *i* and *j* were effectively unrolled into a single dimension, so that the highest block index was $N_x \times N_y$ rather than N_x or N_y.

5. Our Implementations of the FDTD Method on GPU Hardware

We use a similar method to the third approach, but with flexibility in how much work is done by each thread. Figure 7 shows how block and thread dimensions are defined. The data type `dim3` allows one-, two-, or three-dimensional information to be represented. When a kernel is launched, two `dim3` variables are provided, one representing the organization of blocks within a grid, and one representing the organization of threads within each block. Figure 8 shows how the threads within one block are responsible for the elements in a single dimension, in this case, the *x* dimension. Each thread performs the calculations for one or more cells in the FDTD space. The number of cells executed by each thread is determined by the ratio of threads per block to N_x. If, as in Figure 8, N_x is six but the number of threads per block is three, each thread is responsible for two cells in the *x* dimension. The number of active threads is $N_y \times N_z \times$ the number of threads per block. If each thread executes a series of sequential cells as per Figure 8a, then memory access is un-coalesced, since the threads within a block simultaneously access memory locations that are not adjacent. However, if the allocation of cells to threads is interleaved as per Figure 8b, then threads simultaneously request adjacent memory locations, and memory access is coalesced. Figure 9 shows the iterative behavior of a thread in our un-coalesced implementation, while Figure 10 shows the same for our coalesced implementation. `gridDim.x` is set to N_x. There are many ways to improve the computational efficiency by using the shared memory. However, it was not used in our implementation, to see the influence from the methods to access the global memory without being obscured by using the shared memory. The experiments performed here investigated the performance impact of un-coalesced versus coalesced memory access, and also the performance impact of changing the number of cells to be executed by each thread. When the thread count is equal to N_x, both the coalesced and un-coalesced approaches assign one element per thread. This means the behavior at runtime of the un-coalesced and coalesced implementations should be almost identical, with only slight differences in the calculation used to determine the index for each thread. The similarity in execution time when the thread count equals N_x is therefore expected.

6. Numerical Experiments

6.1 Computational Environment

Our GPU implementation was executed on a Tesla T10 GPU and on a GPU with compute capability 2.0 (commonly known as a Fermi GPU), the specifications of which are shown in Table 1. On each GPU, the implementation was executed for 5000 time steps. The center of the 256^3 FDTD space with the PEC boundary condition was excited by a *z*-directed soft point source [13]. The calculation was performed in both single and double precision. The memory-access patters presented in

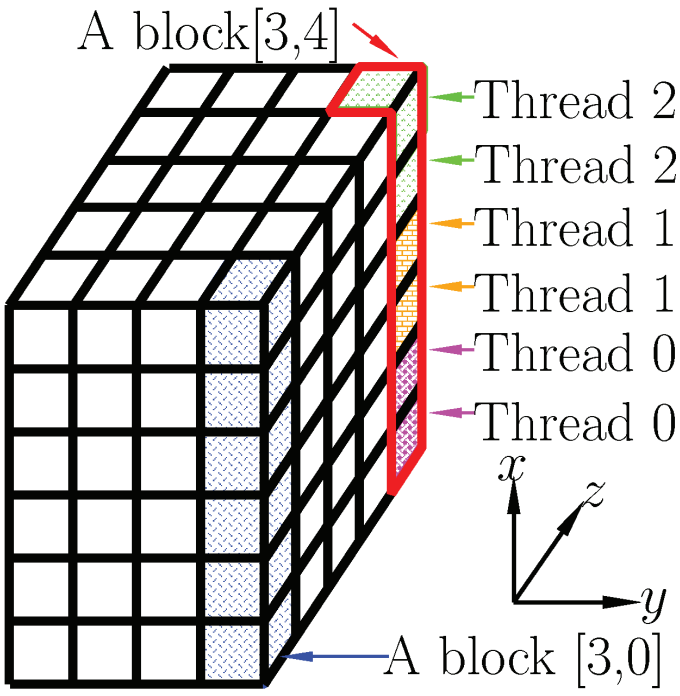


Figure 8a. Un-coalesced memory access for multiple threads in a block. $[a,b]$ means [`blockIdx.x`, `blockIdx.y`].

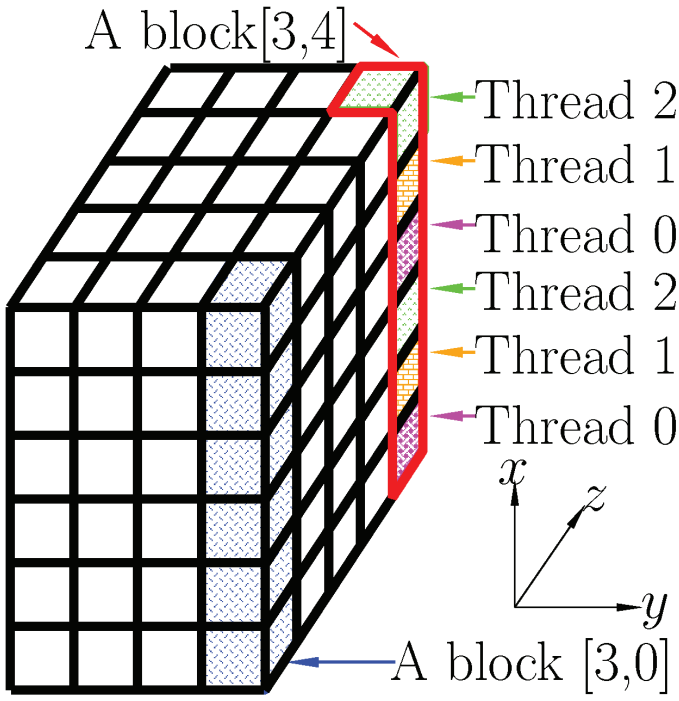


Figure 8b. Coalesced memory access for multiple threads in a block. $[a,b]$ means [`blockIdx.x`, `blockIdx.y`].

Table 1. The specifications of the GPUs used in this report. SM stands for streaming multiprocessor. 64 KB for the shared memory per streaming multiprocessor in the Tesla M2050 was configured as either 48 KB shared memory and 16 KB L1 cache, or 16 KB shared memory and 48 KB L1 cache.

GPU Name	Tesla T10	Tesla M2050
Number of SM	30	14
Number of SP cores	**240** (8 per SM)	**448** (32 per SM)
Clock speed per core	1.3 GHz	1.1 GHz
Single-precision performance	933 GFlops	1030 GFlops
Double-precision performance	**78** GFlops	515 GFlops
Registers per SM	16384	32768
Shared memory per SM	16 KB	64 KB
Constant memory size	64 KB	64 KB
Global memory size	4 GB	3 GB
Level 2 cache size	**None**	**768 KB**
Global memory bandwidth	**32 GB/s**	**148 GB/s**
Maximum threads per block	512	1024
Compute capacity	1.3	2.0

```
j=blockIdx.x+1;
k=blockIdx.y+1;
int iterations=gridDim.x/THREAD_COUNT;
i=threadIdx.x*iterations+1;
while(i<=threadIdx.x*iterations+iterations)
{
 //Calculate   Ex(i,j,k),  Ey(i,j,k) and   Ez(i,j,k)
 i++;
}
```

Figure 9. The structure of the un-coalesced kernel implementation.

```
j=blockIdx.x+1;
k=blockIdx.y+1;
i=threadIdx.x+1;
while(i<=gridDim.x)
{
  //Calculate  Ex(i,j,k),  Ey(i,j,k) and   Ez(i,j,k)
  i+=THREAD_COUNT;
}
```

Figure 10. The structure of the coalesced kernel implementation.

Figure 8a and Figure 8b were both tried. The NVIDIA M2050 Fermi GPU has 1.9 times more cores than the T10 GPU. The M2050 GPU can thus execute a larger number of threads in parallel. All these cores need to access the global memory. The global memory bandwidth of the M2050 GPU is 4.6 times wider than the bandwidth of the T10 GPU. Although the T10 GPU does not have a cache, the M2050 GPU does include cache between the processing units and the global memory. The M2050 GPU has a level 2 cache shared across all streaming multiprocessors, and the option to use 48 KB of the shared memory on each multiprocessor as a level 1 cache was used in this work. The level 1 cache and the level 2 cache reduce the number of requests made to global memory. The double-precision performance of compute capability 2.0 is 6.6 times higher than the performance of compute capability 1.3. However, as [10] pointed out, in the FDTD computation, memory operations can consume GPU clock cycles a couple of hundreds of times more than arithmetic operations. The double-precision performance in the M2050 GPU i.e., 515 GFlops (giga-floating-point operations per second), compared with 78 GFlops in the T10 GPU, would have a minimum impact on the overall performance of the M2050 GPU in our GPU implementation. On the other hand, there is not a large difference in the maximum single-precision performance of the Fermi GPU and the Tesla T10 GPU. Therefore, if there is any noticeable performance difference between these GPUs for the single-precision computation, that would be mainly due to the global memory bandwidth. Although arithmetic operations are very light in the FDTD calculation, large data access is required. For both global and shared memory access, the T10 GPU (compute capability 1.3) experiences bank conflicts for 64-bit access because "memory request is compiled into two separate 32-bit requests" [5]. For the M2050 GPU (compute capability 2.0), 64-bit access is handled to minimize bank conflicts. It is thus expected to that a significantly improved performance of calculation in double precision in the M2050 GPU relative to the T10 GPU would be seen. Given the fact that the FDTD algorithm is memory-throughput limited rather than limited by the arithmetic operations [10], the higher bandwidth will positively affect the performance of our implementation in the M2050 GPU.

6.2 Performance Measurement

Figure 11 shows the overall performance of the T10 (Figure 11a) and M2050 (Figure 11b) GPUs. The abscissa presents

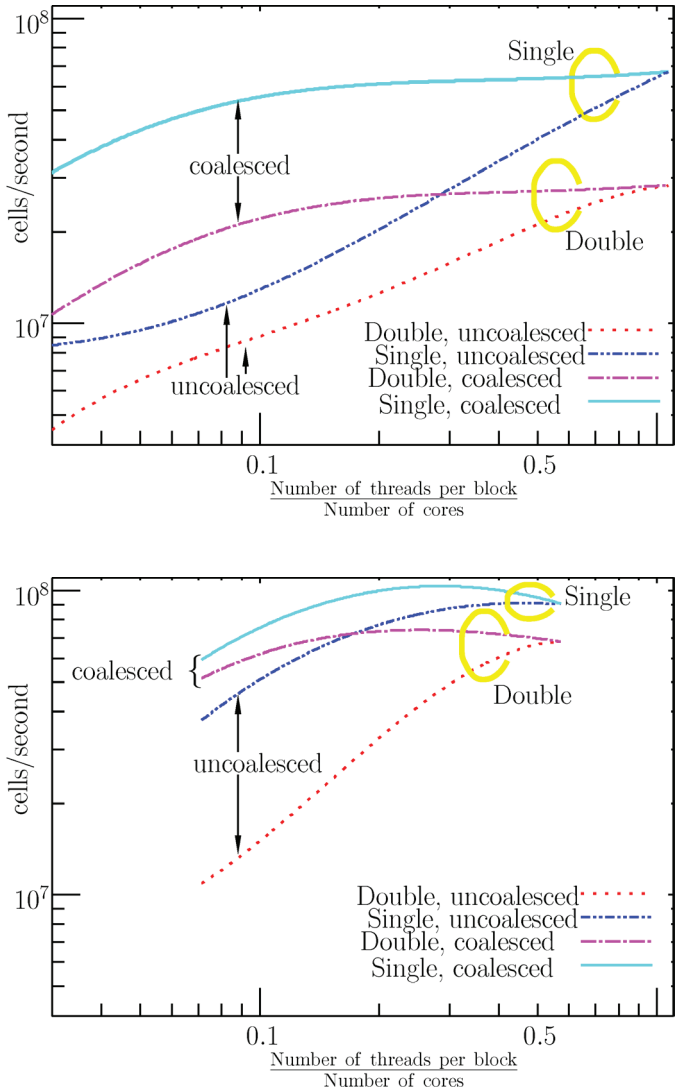


Figure 11. The performance of our implementation on the T10 GPU and the M2050 GPU. A solid line, a dotted line, a broken line with one dot, and a broken line with two dots means the coalesced method in single precision, the un-coalesced method in double precision, the coalesced method in double precision, and the un-coalesced method in single precision, respectively. (a, top) Results on the T10; (b, bottom) Results on the M2050.

$$\mathcal{R} = \frac{\text{Number of threads per block}}{\text{Number of GPU cores available in a GPU board}}, \quad (3)$$

and the ordinate presents the number of FDTD cells calculated per second. The number of the GPU cores in these GPU boards was different. However, by expressing the abscissa as the number of threads per block divided by the number of GPU cores, the result could be applicable to the GPU boards that are not handled in this paper. An equivalent, sequential, CPU implementation was also executed on an Intel Xeon E5620 CPU for comparison. Its operating system was 64-bit *Scientific Linux 5.5*, and the kernel version was 2.6.18. It contained eight cores, the core speed was 2.4 GHz, and the *gcc* version was 4.1.2. For the FDTD computation, only one core was used. The CPU computation in single (double) precision took 29 (33) times more time than the M2050 GPU computation with un-coalesced memory access when each thread was responsible for just one cell, i.e., the fastest case in the un-coalesced memory-access approach. When a thread was responsible for all cells in one block, the GPU computation took more than 1.3 times as much time as the CPU computation. It was clear in Figure 11 that the approach in Figure 8b outperformed the approach in Figure 8a. The coalesced approach performs best when R is around 0.15.

The same results were obtained when the different sizes of the FDTD space were tested. In the case of the coalesced approach in single precision, the T10 GPU took about 1.5 times more time than the M2050 GPU. This suggests the improvement of the performance by the M2050 GPU in single precision mainly results from the increase in memory bandwidth, because the bandwidth of the M2050 GPU is 4.6 times wider than that of the T10 GPU. In the case of double precision, our coalesced approach performed better on the M2050 GPU than on the T10 GPU, reducing 60% of the elapsed time of the T10 GPU. The M2050 GPU has both good double-precision performance and the level 1 and 2 caches. However, we find the contribution of these to the performance improvement to be negligible. First, the elapsed time of the FDTD method is mostly dominated by memory operations, rather than arithmetic operation. Second, the cache will be quickly overwritten by other threads, and the data from the global memory may not be reused to compute each of the neighboring field values; it is thus tricky to effectively make use of the cache for speedup with the FDTD method. We find the major contribution to be the reduction of bank conflicts that were occurring on the T10 GPU. 64-bit access on the T10 GPU architecture requires two memory instructions, while the M2050 GPU requires only one [5]. The bank conflicts occur in global memory, which causes the T10 GPU architecture to use twice the number of memory lookups. This means that memory fetches for 64-bit words happen twice as fast on the M2050 GPU, which probably accounted for 50% of the speedup. The required memory access for double-precision computation in our approach would exceed the available bandwidth in the T10 GPU. In the M2050 GPU, the special treatment of 64-bit access reduced the frequency of the global memory access. This may make the practical bandwidth required for double-precision computation comparable to one in single precision. In the M2050 GPU, the double-precision computation thus took only 1.3 times more time than the single precision. On the other hand, in the T10 GPU, the double-precision computation took 2.5 times more time than the single precision. When our approach was compared to implementations of methods 1 and 2 running on the same T10 GPU, our method showed 6.4 and 6.6 times higher performance than methods 1 and 2, respectively, for single precision. For double precision, our method showed 3.3 times higher performance than both methods 1 and 2.

7. Conclusion

This paper categorized the currently available approaches for implementing the three-dimensional FDTD method on GPGPUs, and explained the merits and demerits of these approaches. The most promising approach was taken as a base for our work, and a further improvement to the approach was proposed. Numerical experiments showed that when the ratio between the number of threads per block and the number of cores available in a GPU board was around 0.15, our coalesced approach had a significantly more efficient global-memory access, just enough for the bandwidth available in the T10 GPU in the case of single precision. In this case, the M2050 GPU gave the performance improvement only for the increase of the bandwidth compared with the T10 GPU. However, the double-precision computation doubles the amount of data requested. In our approach, the access in the double-precision computation doubles the global-memory access of the single-precision computation in the T10 GPU, and significantly exceeded the bandwidth of the T10 GPU. The performance was thus less than half of the single-precision performance. On the other hand, our approach significantly benefited from the specific treatment of 64-bit access in the M2050 GPU, and the calculation in double precision took only 1.3 times more time than the single-precision calculation.

8. References

1. A. Taflove and S. C. Hagness, *Computational Electrodynamics: The Finite-Difference Time-Domain Method, 3rd Edition*, Norwood, MA, Artech House Publishers, 2005.

2. E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, **28**, 2, March-April 2008, pp. 39-55.

3. J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, Reading, MA, Addison-Wesley Professional, 2010.

4. D. Kirk and W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach (Applications of GPU Computing Series)*, Burlington, MA, Morgan Kaufmann, 2010.

5. NVIDIA, "NVIDIA CUDA C Programming Guide 4.1 ," Technical Report, November 2011, available at http://

developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf.

6. NVIDIA Corporation, "NVIDIA's Next Generation CUDA Compute Architecture: Fermi," whitepaper, 2009, available at http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

7. J. Chi, F. Liu, E. Weber, Y. Li, and S. Crozier, "GPU-Accelerated FDTD Modeling of Radio-Frequency Field-Tissue Interactions in High-Field MRI," *IEEE Transactions on Biomedical Engineering*, **58**, 6, June 2011, pp. 1789-1796.

8. Z. Bo, X. Zheng-hui, R. Wu, L. Wei-ming, and S. Xin-qing, "Accelerating FDTD Algorithm Using GPU Computing," IEEE International Conference on Microwave Technology & Computational Electromagnetics, May 2011, pp. 410-413.

9. T. Nagaoka and S. Watanabe, "A GPU-Based Calculation Using the Three-Dimensional FDTD Method for Electromagnetic Field Analysis," in International Conference of the IEEE, Engineering in Medicine and Biology Society, 2010, pp. 327-330.

10. J. F. Stack, "Accelerating the Finite Difference Time Domain (FDTD) Method with CUDA," Applied Computational Electromagnetics Society Conference, 2011.

11. C. Y. Ong, M. Weldon, S. Quiring, L. Maxwell, M. Hughes, C. Whelan, and M. Okoniewski, "Speed it Up," *IEEE Microwave Magazine*, **11**, 2, April 2010, pp. 70-78.

12. P. Sypek, A. Dziekonski, and M. Mrozowski, "How to Render FDTD Computations More Effective Using a Graphics Accelerator," *IEEE Transactions on Magnetics*, **45**, 3, March 2009, pp. 1324-1327.

13. F. Costen, J.-P. Berenger, and A. Brown, "Comparison of FDTD Hard Source with FDTD Soft Source and Accuracy Assessment in Debye Media," *IEEE Transactions on Antennas and Propagation*, **AP-57**, 7, July 2009, pp. 2014-2022.

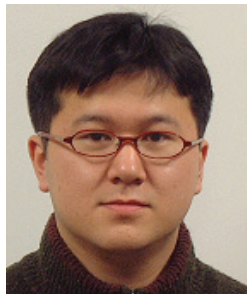
Introducing the Authors

Matthew Livesey graduated from University of Manchester in 2005 with BEng Software Engineering (1st class Hons). He subsequently joined Accenture, working as an IT consultant. In 2011, he returned to the University of Manchester, and received an MSc with distinction in Computer Science, and was awarded the Peter Jones prize as the highest achiever in the year. Matthew continues to work at Accenture as an IT Project Manager and Solution Architect. He is interested in parallel and distributed systems, and is currently working with Big Data in Hadoop.

James F. Stack, Jr. was born in Kettering, Ohio in 1979. He received a BS in Electrical Engineering and ME in Systems Engineering from the Pennsylvania State University in 2001 and 2010, respectively. He joined Remcom Inc. in 2000 as an intern, and began full-time work upon completion of his bachelor's degree. He served as a software engineer until 2005, and lead developer of *XFdtd version 6* from 2005 to 2008. In 2009, he transitioned to lead the Training, Applications and Consulting department. His current research interests include stream processing and optimization on both the CPU and GPU.

Fumie Costen received the BSc, the MSc in Electrical Engineering, and the PhD in Informatics, all from Kyoto University, Japan. From 1993 to 1997, she was with Advanced Telecommunication Research International, Kyoto, where she was engaged in research on direction-of-arrival estimation based on Multiple SIgnal Classification (MUSIC) algorithm for three-dimensional laser microvision. She received an academic invitation at Kiruna Division, Swedish Institute of Space Physics, Sweden in 1996, and gained three patents from the research in 1999. From 1998 to 2000, she was with Manchester Computing in the University of Manchester, UK, where she was engaged in research on metacomputing, and has been a Lecturer since 2000. Her research interests include computational electromagnetics in such topics as a variety of the finite-difference time-domain methods for the microwave frequency range, and high-spatial-resolution and FDTD subgridding and boundary conditions. Her work extends to the hardware acceleration of computation using general-purpose computing on graphics-processing units, streaming single instruction multiple data extension (SSE) and advanced vector extensions instructions. Dr. Costen received an ATR Excellence in Research Award in 1996, and a best paper award from the 8th International Conference on High Performance Computing and Networking Europe in 2000.

Seiji Fujino was born in Fukuoka, Japan, in 1950. He graduated from the Faculty of Science of Kyoto University in 1974. He received the DE in Computer Science from Tokyo University in 1993. Currently, he is a professor of the Research Institute for Information Technology of Kyushu University. His research interest is numerical analysis, in particular, iterative methods of the Krylov subspace method. He is a member of JSIAM of Japan.



Norimasa Nakashima received the BE, ME, and DE from Kyushu University, Fukuoka, Japan in 1999, 2001, and 2004, respectively. He joined the Faculty of Information Science and Electrical Engineering, Kyushu University, as a research associate in 2004, and became an Assistant Professor in 2007. He is currently an Associate Professor of the Department of Information and Communication Engineering, Fukuoka Institute of Technology, Japan. His research interest is in computational electromagnetics. He is a member of JSIAM, ACES, and IEICE.

Takeshi Nanri received the BE and ME from Kyushu University. After joining the Computer Center of the university, he received the PhD in Computer Science in 2000. Since 2001, he has been an Associate Professor at the Research Institute for Information Technology of Kyushu University. His major interest is the middleware for large-scale parallel computing. He is now working on the techniques of optimization on collective communications and rank allocations.