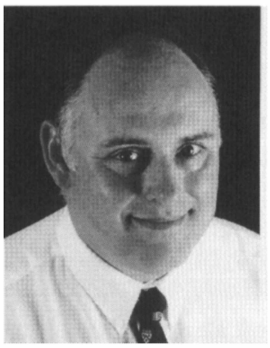


David B. Davidson
Dept. E&E Engineering
University of Stellenbosch
Stellenbosch 7600, South Africa
Tel: +27 21 808 4458
Fax: +27 21 808 4981
E-mail: davidson@sun.ac.za

Foreword by the Editor

The stagnation of CPU clock speeds has brought parallel computing, multi-core architectures, and hardware-acceleration techniques back to the forefront of computational science and engineering. In particular, there has been an explosion of interest in general-purpose graphical-processing units (GPGPUs). In the latest contribution from this group on computational and IT issues impacting on computational electromagnetics (CEM), the use of GPGPUs for FDTD computations is discussed. (In the April 2003 column, they introduced grid computing to our readers; the April 2006 contribution extended many of the ideas, focusing on semantic grids; the October 2007 column considered the use of ontologies in the field of knowledge modeling; and in the August 2009 column, software agents in the context of grid computing were described). We thank the authors for their continuing contributions. Papers discussing other applications in CEM of GPGPUs are scheduled for future columns.

Introduction to GPU Computing and CUDA Programming: A Case Study on FDTD

Danilo De Donno, Alessandra Esposito, Luciano Tarricone, and Luca Catarinucci

University of Salento
Via Monteroni, 73100 – Lecce, Italy
Tel: 0039 0832 29 - 7261, 7226, 7278; Fax: 0039 0832 1830127
E-mail: {alessandra.esposito, danilo.dedonno, luciano.tarricone, luca.catarinucci}@unisalento.it

Abstract

The recent advent of general-purpose graphics-processing units (GPGPUs) as inexpensive arithmetic-processing units brings a relevant amount of computing power to modern desktop PCs. This thus providing an interesting pathway to the acceleration of several numerical electromagnetic methods. In this paper, we explain how to exploit GPGPU features by examining how the computational time of the Finite-Difference Time-Domain Method can be reduced. The attainable efficiency is demonstrated by providing numerical results achieved on a two-dimensional study of a human-antenna interaction problem.

Keywords: Parallel processing; parallel programming; FDTD methods; general purpose graphics processing unit (GPU); Compute Unified Device Architecture (CUDA).

1. Introduction

The solution of large and complex electromagnetic (EM) problems often leads to a substantial demand for high-performance computing (HPC) resources and strategies. In many cases, the adoption of high-performance computing tools guarantees an impressive reduction of solution times. In several other cases, it paves the way to the investigation of problems otherwise unaffordable. These observations are valid for a wide variety of numerical methods and applications, ranging from EM compatibility to radio coverage [1, 2].

Until now, two basic answers have been given to this demand for peak performance. The first is represented by *architectural* solutions, i.e., the migration of EM codes towards parallel and high-performance computers. Many different platforms have been developed in the last decades, reaching teraFLOPs (or higher) performance. The most relevant limitation of this architectural solution is represented by its cost: supercomputers are expensive.

The second solution is represented by low-cost *distributed platforms*, combined with suitable *software* tools, such as workstation clusters plus *PVM* [3] (late 1980s and early 1990s), or the very recent grid-computing technologies [4-6]. This latter solution, which is opening very challenging perspectives [7], suffers from a major drawback. In order to be effective, it needs very-large-bandwidth connections. Although this requirement is today more and more fulfilled, in some cases bandwidth still represents an unpleasant bottleneck.

Recently, a third approach has been proposed to overcome the aforementioned limitations (costs and wideband connectivity): the exploitation of graphical-processing units (GPUs) to speed up computation. The adoption of GPU computing for EM numerical methods is a rather new issue, even though it is featuring a positive trend. This is also due to the recent publication of the Compute Unified Device Architecture (CUDA) [8], a powerful and simple-to-use programmer environment available for Nvidia cards, which renders GPU computing accessible to developers not expert in computer graphics. Several EM numerical methods are amenable to benefit from appealing GPU characteristics [2, 9, 10]. Among them, the Finite-Difference Time-Domain (FDTD) algorithm is perhaps the best suited to be implemented in a GPU-enabled fashion. It is the simplest to describe, as it is an inherently data-parallel algorithm. It was therefore chosen in this paper as a case study for demonstrating GPU capabilities, and for illustrating the software design and implementation efforts required for developing efficient parallel codes on a GPU.

Accordingly, in this paper we explain how to implement a CUDA version of the FDTD algorithm so as to achieve peak performance. The attainable efficiency is demonstrated by discussing a two-dimensional study of a human-antenna interaction problem, implementing the FDTD algorithm on an off-the-shelf, inexpensive GPU.

The paper is structured as follows. Section 2 summarizes the main concepts of GPU computing, and of the CUDA programming model. Section 3 describes how to implement a GPU-enabled version of the FDTD. Section 4 focuses on software strategies for optimizing performance, and provides numerical results obtained on an off-the-shelf, economical GPU. Finally, conclusions are drawn.

2. GPU Programming and CUDA Overview

In recent years, GPUs have become increasingly attractive for general-purpose parallel computation. Parallel codes exploiting GPU hardware may yield results equivalent to tens of traditional CPUs, at a fraction of the cost. This gap is going to increase, because the ever-increasing requirements of the video-game market make the GPU rate of growth much higher than that of traditional microprocessors. Moreover, CPU hardware is optimized for generic sequential codes, being mostly dedicated to non-computational tasks, such as branching. Instead, GPU design is totally devoted to the optimization of graphics operations, which are parallel by nature. As a result, GPUs dedicate a greater number of transistors to computation, thus achieving an enormous arithmetical intensity.

In order to provide an idea of available resources, we observe that recent GPUs may embed up to 60 streaming multiprocessors (SM), each being able to execute up to 1024 concurrent threads. The price of GPUs can be very low (a few hundreds of Euros) and still provide huge processing power. For example, the inexpensive off-the-shelf GPU we used for our tests contained four streaming multiprocessors, each equipped with eight processors, and supported up to 768 concurrent threads.

The huge computing capability of the GPU is attracting more and more software developers involved in diverse scientific areas. However, until a few years ago, GPU programming was a very hard task, as it required a deep knowledge of GPU architecture and of graphics terms and models. Recently, Nvidia has greatly reduced the programmer's burden by launching CUDA [8]. CUDA includes *C/C++* software-development tools, function libraries, and a hardware-abstraction mechanism that hides the GPU hardware from developers.

Even though CUDA has greatly simplified the implementation of GPU-enabled applications, a number of hardware and software constraints must still be addressed in order to achieve high performance. In order to understand how to deal with such limitations, the comprehension of some simple basic concepts of CUDA is fundamental.

According to CUDA, data-parallel portions of an application are implemented as *kernels*. The main CPU, acting as the *host*, can initiate one kernel at a time. Each kernel is executed in parallel by several threads. The programmer groups threads into blocks, which are logically aggregated into a *grid*. When a CUDA application is executed, threads are scheduled in groups of *warps*. A warp executes one common instruction at a time, so full efficiency is realized when all threads of a warp share the same execution path: if threads of a warp diverge via a conditional branch, the warp serially executes each branch path, thus canceling the advantages of parallelization. It is therefore good sense to design kernels to avoid branch paths. Other software strategies concern the optimization of memory-access patterns.

The host and the device (i.e., the GPU) communicate by copying data from/to the CPU's memory to/from the so-called GPU *global* memory. Therefore, by default, threads operate on data stored in the GPU's global memory. This kind of memory is off-chip, and features huge latency times. Other memory devices – faster than global memory – are available, and CUDA provides the software constructs to transfer data to/from the global memory

from/to these devices. It is a programmer's responsibility to carefully design and implement kernel codes in order to minimize the number of global-memory reads, by making use of the other available kinds of memory (see Section 4 for more details). Moreover, when global memory must be used, it is essential that threads access consecutive array elements, so that memory reads are automatically combined (*coalesced*) into larger reads, thus reducing the effective number of memory transactions.

These and other programming strategies are described in the next sections, by illustrating a simple CUDA implementation of the well-known FDTD algorithm.

3. Implementing a CUDA-Enabled FDTD

3.1 The Flowchart

The FDTD approach [11] partitions space into elementary cells and, assuming that rectangular coordinates are used, the generic space point, P, is identified with notation (i,j,k). Any space and time function, F, is identified with notation $F|_{i,j,k}^{n}$, this meaning that function F is computed at time $n\Delta t$ (where Δt is the time step) at point (i,j,k).

The temporal and spatial discretizations adopted in the FDTD algorithm are implemented at their best to solve Maxwell's equations by using a leap-frog integration scheme. In the leap-frog scheme, at time $t = n\Delta t + 1/2$ at each mesh point (i,j,k), each $H^{n+1/2}$ component is computed as a function of the previous value, $H^{n-1/2}$, at the same point, plus a function of E components at time $t = n\Delta t$ at the mesh points belonging to the neighborhood of (i,j,k). Similar computations are performed for computing the E components.

To simplify, the typical FDTD implementation may be described by the following pseudocode:

```
for each timeIteration
{
    for each cell {
        UpdateFieldValues;
    }
}
```

In the CUDA-enabled implementation, the inner loop disappears, as the calculations over the cells are operated in parallel by CUDA threads. Figure 1 depicts the mapping between CUDA blocks and a two-dimensional domain. As shown in the figure, thread blocks are associated with sets of contiguous cells, one thread being allocated to each Yee cell.

Figure 2 shows the flowchart of a typical FDTD CUDA-enabled implementation. According to such a flowchart, we have two kernels for the calculation of the field components: the former calculates the H components, while the latter calculates the E components. A transfer of data between host and device is performed both before and after kernel invocation. First, the main CPU allocates space over the GPU's *global* memory, and transfers therein data needed by kernels. At the end of computations, results are copied to the CPU's main memory.

A kernel call has the following form:

```
kernelName <<< gridDim, blockDim >>> (..parameter
list..);
```

where `gridDim` and `blockDim` respectively contain the dimensions of the CUDA grid and of the blocks. Execution of the kernel creates `gridDim.x*gridDim.y` thread blocks, and `blockDim.x*blockDim.y` threads within each block. Each thread has its own *id*, thus making it possible to assign a specific cell to each thread.

In the following subsections, we give some more details about FDTD kernel implementation, with the hypothesis of having the same step of discretization, $\Delta = \Delta x = \Delta y$, along the x and y axes.

3.2 Kernel Implementation

The following listing shows the H_z kernel code, providing the computation of the z-axis component of the magnetic field:

```
__global__ void HZ_kernel(float *d_EX, float
    *d_EY, float *d_HZ, float coH)
{
    int th_x = threadIdx.x;
    int th_y = threadIdx.y;
    int x = blockIdx.x * blockDim.x +
            threadIdx.x;
    int y = blockIdx.y * blockDim.y +
            threadIdx.y;
    if (x < (NX-1) && y <(NY-1)) {
        d_HZ[x+y*NX] = d_HZ[x+y*NX] + coH
            *(d_EY[(x+1)+y*NX] -
            d_EY[x+y*NX] - d_EX[x+(y+1)*NX] +
            d_EX[x+y*NX]);
 }
}
```

The function type qualifier `__global__` declares that the function is a kernel executable on the CUDA device. Each thread and each block are provided with a unique ID, respectively given by the

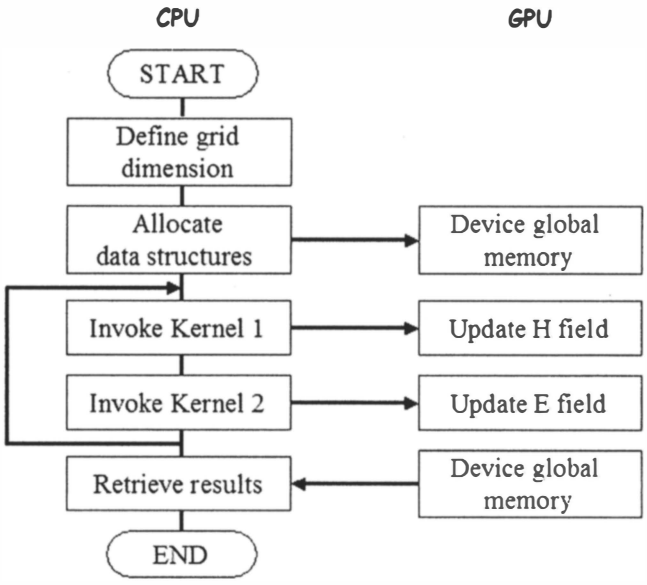


Figure 2. A flowchart of the GPU-enabled FDTD code.

couples (`threadIdx.x`, `threadIdx.y`) and (`blockIdx.x`, `blockId.y`). Such IDs are used to compute the indices of the associated cell (x and y), and to calculate H_z according to Yee's formula. Note that in order to best exploit the linearity of GPU global memory, field values are stored in one-dimensional arrays in row-major order.

This is a very simple example of kernel implementation, which does not take into account the architecture of CUDA cards, nor does it exploit available CUDA software mechanisms. A number of useful software enhancements to the above kernel are proposed in the following section, together with numerical results obtained in a real-life FDTD application.

4. Performance Optimization

First, much higher-performing kernels can be implemented when the CUDA memory hierarchy is considered. As explained before (Section 2), the CUDA memory model comprises various memory spaces, which differ enormously in latency times, availability of caches, etc. Particularly important are global and shared memories, the former being an off-chip memory and thus featuring high-access latency; the latter being an on-chip memory and thus having reduced latency. Global memory has the lifetime of the application and is accessible from all running threads. The low-latency on-chip shared memory instead has the lifetime of a block, and is accessible from all the threads within the same block. The high-latency global memory is used by the CPU to share data with the GPU. In order to minimize the performance loss due to global-memory access, programmers are therefore in charge of designing and implementing the CUDA code so that data accessed at the block level reside on the shared memory.

In the FDTD algorithm, the calculation of a field component depends at each step on the value of the same component at the previous time step, and on some field components at neighboring cells (Figure 3). One way to exploit the availability of the on-chip shared memory is therefore to enable kernels to load therein all the needed field components, including the components corresponding to adjacent blocks (Figure 4). This requires the allocation of space in the shared memory, and the execution of data transfers from global to shared memory. The following lines of code show how to allocate a data structure in the shared memory (`__shared__` keyword), and how data can be copied from the global memory (`d_EX` array) to the shared memory (`EX_S` array):

```
__shared__ float EX_s[(BLOCKSIZE)*(BLOCKSIZE)];

EX_s[th_x+(BLOCKSIZE)*th_y] = d_EX[x+NX*y];
```

According to row-major indexing, the `th_x+(BLOCKSIZE)*th_y` index identifies the current thread, whilst the `x+NX*y` index refers to the associated cell. In this way, each thread loads the value of the E_x component calculated at the associated cell. Similar operations can be performed for the other field values required by Yee's formulation. Once all needed data have been loaded into the shared memory, Yee's formulation is efficiently calculated on top of shared-memory data.

Figures 5 and 6 show the output of the "CUDA Visual Profiler," a tool that enables the profiling of CUDA kernels. Shown are the outputs of two implementations of the GPU-enabled FDTD algorithm: thee former exploited the shared memory, while the latter used only the global memory. As the figures show, shared-memory exploitation provides huge performance gains in terms of GPU and CPU execution time. Such gains are due both to the high efficiency of shared-memory access, and to the limited number of *uncoalesced* accesses, i.e., situations where a separate memory transaction is issued for each thread. Uncoalesced accesses have a negative impact on performance, and occur when addresses accessed simultaneously by multiple threads are not correctly aligned, and cannot be aggregated into a single transaction. As shown in the screenshots, uncoalesced accesses have been eliminated by implementing kernels that make use of shared memory.

A further performance improvement can be obtained by using the so called *built-in* arrays (i.e., arrays having two, three, or four components accessible through the fields *.x*, *.y*, *.z*, and *.w*), which allow the programmer to best exploit global-memory bandwidth. Indeed, built-in arrays minimize the number of access operations by maximizing the number of bytes being simultaneously transferred. As an example, in a single built-in array, the E_x and E_y components can be declared (`float2` keyword), the former being accessible through the *.x* field, the latter through the *.y* field:

```
__shared__ float2 EX_EY[(BLOCKSIZE)*(BLOCKSIZE)];
```

In this way, the memory-transfer formulas of the E_x and E_y components become

```
EX_EY[th_x+(BLOCKSIZE)*th_y].x = d_EX[x+NX*y];
EX_EY[th_x+(BLOCKSIZE)*th_y].y = d_EY[x+NX*y];
```

Finally, a further improvement can be obtained by exploiting the so called *texture* memory, which is used by GPU chipsets to accelerate frequently performed graphics operations. Texture memory buffers data in a suited cache, optimized for two-dimensional spatial locality, thus providing huge performance gains when threads read locations that are spatially close together (as happens in the FDTD case). Texture memory is read from kernels by using device functions called *texture fetches*, which bind texture-memory areas to global-memory regions. For instance, the following piece of code declares a texture reference for the E_x field component (namely, `EX_texRef`), binds it to a global-memory structure (`cudaBindTexture` function), and performs a texture fetch (`tex1Dfetch` function):

```
texture <float, 1, cudaReadModeElementType>
    EX_texRef;
cudaBindTexture(0, EX_texRef, d_EX, size_d_EX);
tex1Dfetch(EX_texRef, x+NX*y);
```

When texture memory is also used, the memory-transfer formulas of the E_x and E_y components are

```
EX_EY[th_x+BLOCKSIZE*th_y].x =
    tex1Dfetch(EX_texRef, x+NX*y);
EX_EY[th_x+BLOCKSIZE*th_y].y =
    tex1Dfetch(EY_texRef, x+NX*y);
```

Once that CUDA code is ready to run, an important aspect to take into account is the so-called *multiprocessor occupancy*. Threads are scheduled for parallel execution in scheduling units called "*warps*." The multiprocessor occupancy indicates the ratio of active warps to the maximum number of warps supported on a multiprocessor of the GPU. A 100% occupancy indicates that the application fully exploits available processing units. Unfortunately, the amount of shared memory and registers used by each thread block limits the occupancy value. The size of thread blocks and/or shared-memory and registers usage must therefore be designed

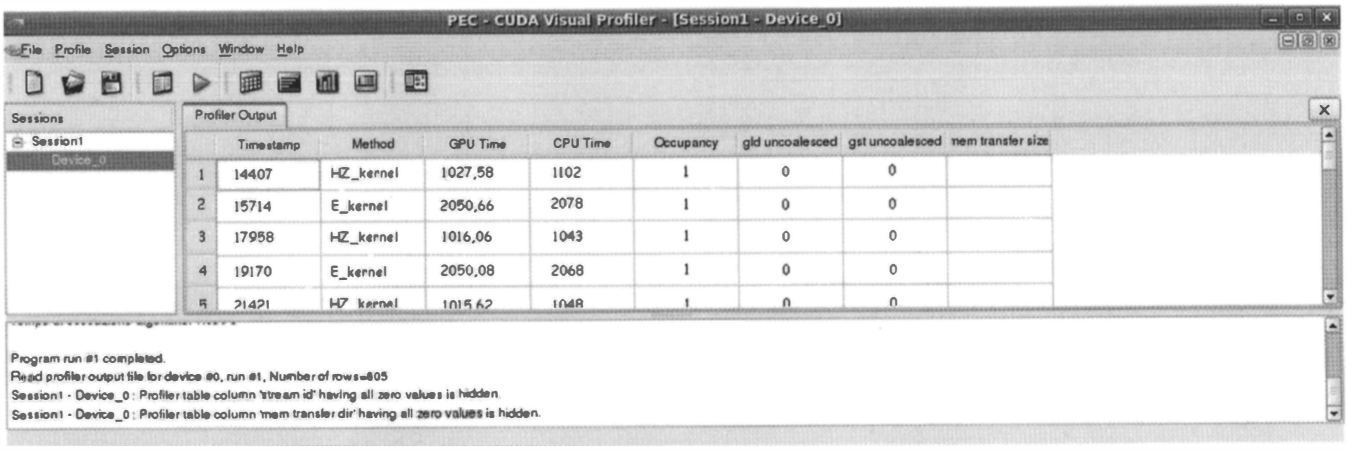


Figure 5. A screen shot of the CUDA Profiler, showing the performance data for FDTD kernels when the shared memory was exploited.

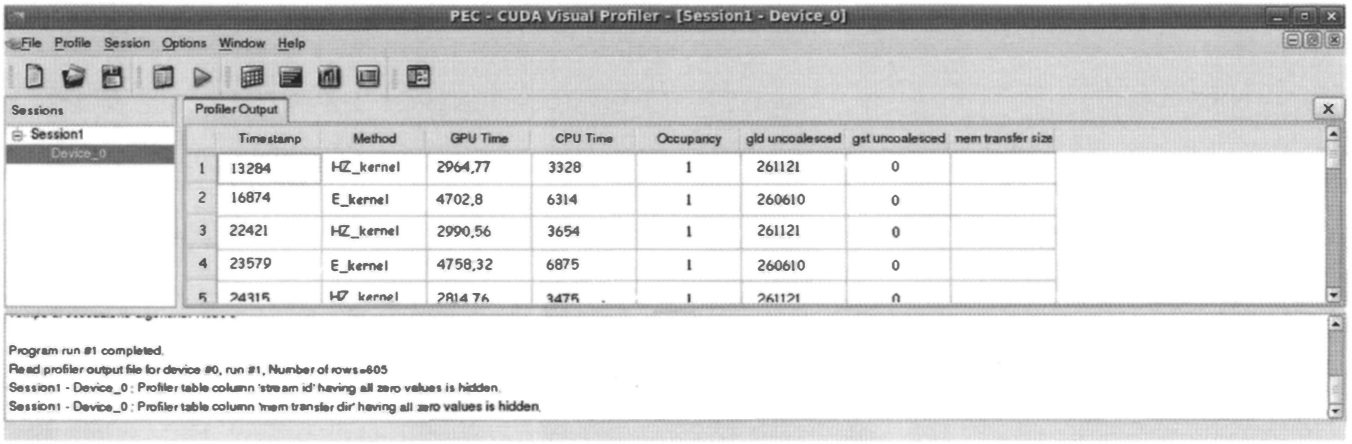


Figure 6. A screenshot of the CUDA Profiler, showing the performance data for FDTD kernels when the shared memory was not exploited.

with care in order to maximize occupancy. The CUDA software environment provides the developer with a useful tool, the so-called "Occupancy Calculator," which assists her/him in choosing the thread-block size, based on shared-memory and register requirements. Consider, for instance, the kernel that computes *E*-field components. Figure 7 shows the effect on multiprocessor occupancy of using different block sizes and registers per thread, on the card described by Table 1. As shown in the figure, a 100% multiprocessor occupancy (the triangle) is obtained when defining blocks of size 384.

As a final assessment of benefits derived from the previously described software strategies, we report in Figure 8 the performance gain obtained by using shared (rather than global) memory, and by applying the previously described software strategies (built-in arrays, texture-memory exploitation, and maximization of multiprocessor occupancy).

Table 2 shows the performance obtained when the practical problem of human interaction with a radio base-station antenna [1] was solved by varying the problem sizes (up to 5100 × 5100 cells), and for 1000 time iterations, with the optimized code. The table reports the speed-up and the speed, evaluated according to the following formula [12]:

$$Speed\ [Mcells/s] = \frac{NX\ NY\ n_{it}}{10^6\left(runtime\ [s]\right)}.$$

Table 1. A list of card resources for the Nvidia GeForce 9500 GT.

Global Memory	1 GB
Constant Memory	64 KB
Shared Memory per Block	16 KB
Processor Clock Rate	550 MHz
Memory Bandwidth	16 GB/sec
Number of Multiprocessors	4 (32 processor cores)

Table 2. The performance obtained for different problem sizes.

Problem Size	CPU Runtime [s]	GPU Runtime [s]	Speed-Up	Speed [Mcells/s]
300×300	4.65	0.48	9.69	187.50
1500×1500	211.04	10.4	20.29	216.34
2100×2100	581.80	21.71	26.79	203.13
2700×2700	1149.55	35.88	32.03	203.17
3300×3300	1877.03	53.77	34.90	202.52
3900×3900	2657.30	76.48	34.74	198.87
4500×4500	3682.66	99.16	37.13	204.21
5100×5100	5412.40	121.5	42.48	214.07

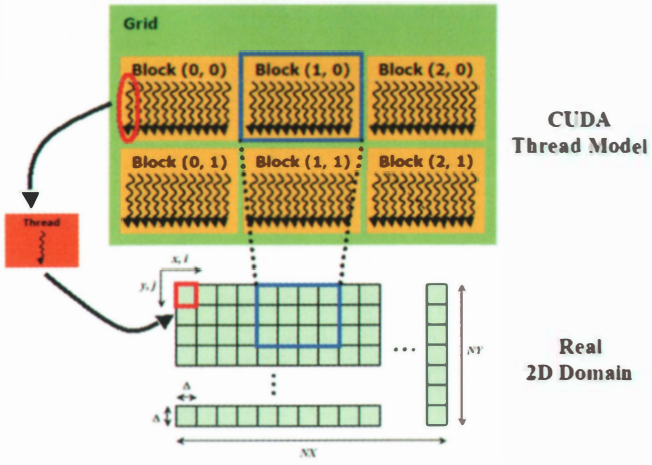
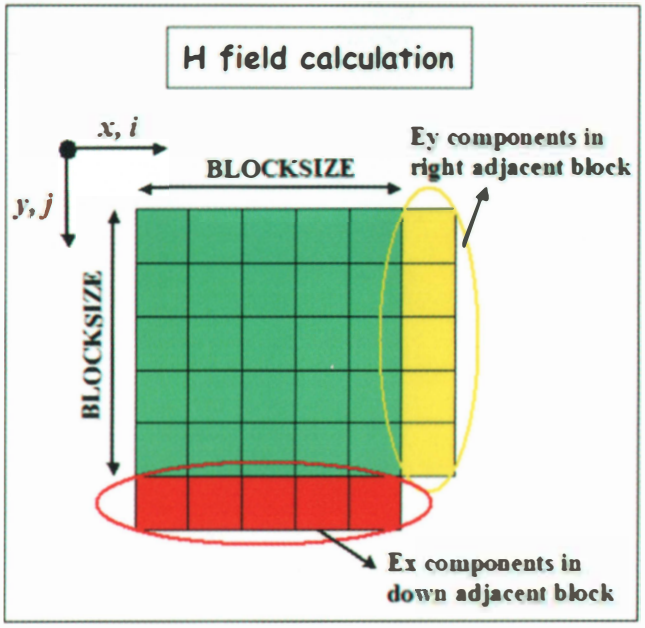


Figure 1. The mapping between CUDA blocks and the two-dimensional domain.



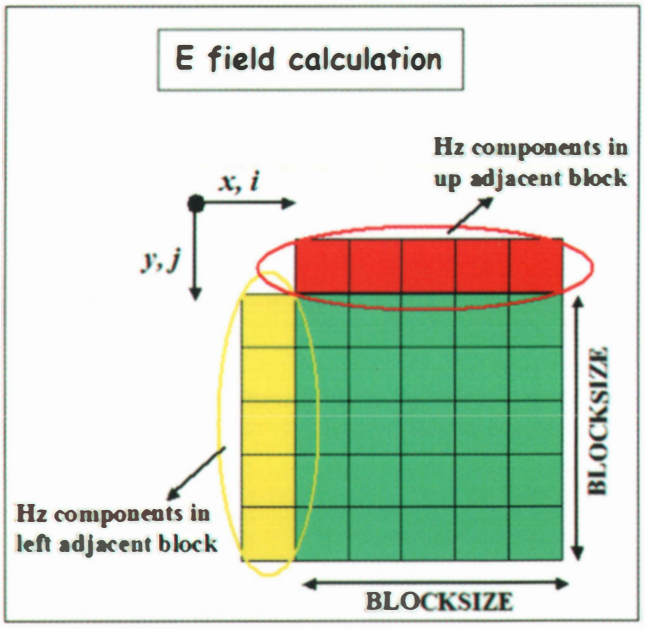


Figure 3. The field-component calculation depends on field components at neighboring cells. Therefore, cells situated at the border of CUDA blocks need to access the values of field components belonging to adjacent blocks.

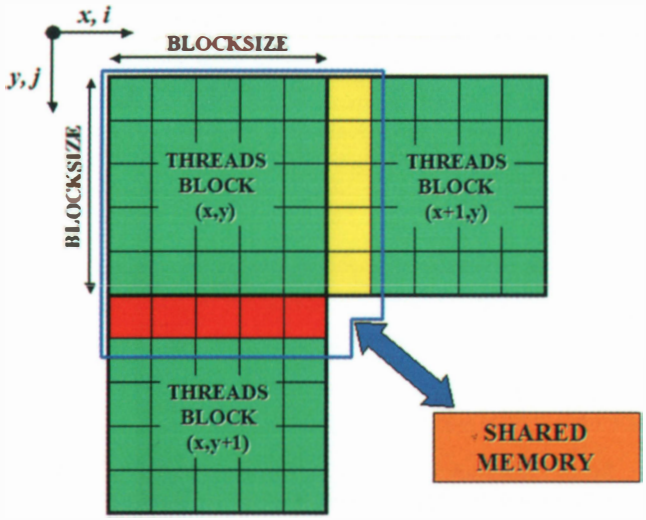


Figure 4. Each block allocates a space in the shared memory suited to store all the field values needed to perform the calculations at the current iteration.

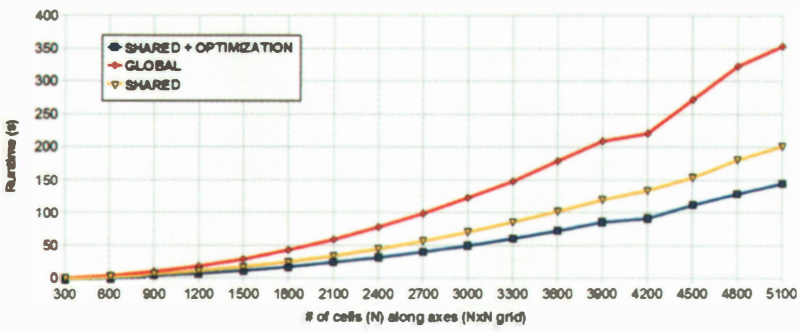
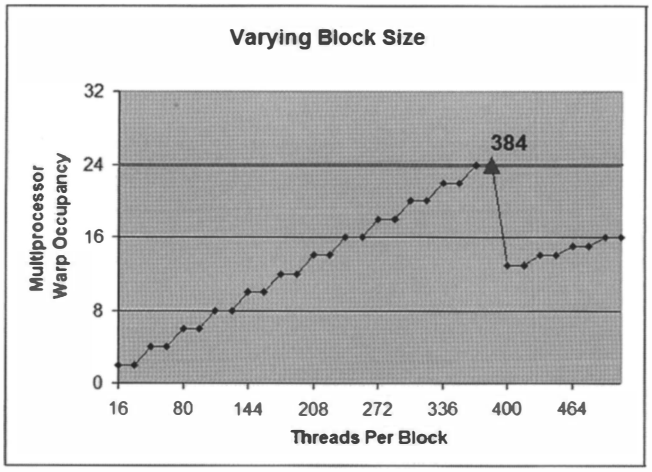


Figure 8. A comparison of three FDTD implementations.



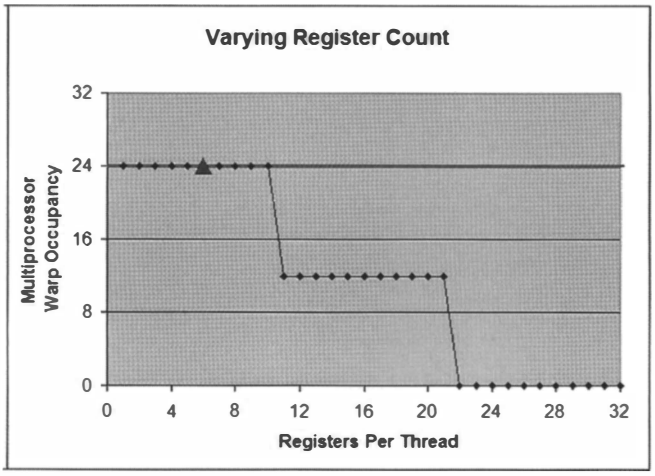


Figure 7. The output from the CUDA Occupancy Calculator, showing the effect on multiprocessor occupancy of using different block sizes and registers per thread.

The reported results were obtained by using an inexpensive off-the-shelf GPU, i.e., the CUDA-compatible Nvidia GeForce 9500 GT GPGPU, featuring 32 streaming processors and a global memory of 1 GB (Table 1), and a PC equipped with an AMD Semprom 3400+ @ processor (1.8 GHz) with 1.0 GB memory (DDR2-533). As shown by such data, relevant speed-ups when compared to sequential execution on a host PC and huge speeds can be obtained, even with inexpensive off-the-shelf GPUs.

5. Conclusions

In this paper, we introduced GPU computing and the CUDA programming environment by explaining, step-by-step, how to implement an efficient GPU-enabled FDTD application. The FDTD was chosen as an example algorithm, but many other EM methods are suitable to exploiting GPU benefits. We also provided results obtained for an application running on an inexpensive off-the-shelf GPU. As shown in the paper, even a non-optimized parallel implementation of the FDTD on a GPU can lead to a significant decrease in computation time with respect to a CPU implementation. However, in order to fully exploit the available hardware, several software strategies have to be carefully designed and implemented. Using the techniques described in the paper, a peak speed-up of 42.5 was obtained, this demonstrating the huge potential of GPU processing for scientific computing.

6. References

1. L. Catarinucci, P. Palazzari, and L. Tarricone, "Human Exposure to the Near-Field of Radio Base Antennas: a Full-wave Solution Using Parallel FDTD," *IEEE Transactions on Microwave Theory and Techniques*, **51**, 3, March 2003, pp. 935-941.

2. A. Valcarce et al., "Applying FDTD to the Coverage Prediction of WiMAX Femtocells," *EURASIP Journal on Wireless Communication Networks*, **2009**, February 2009, 308606.

3. PVM Web site: http://www.epm.ornl.gov/pvm/.

4. L. Tarricone and A. Esposito, *Grid Computing for Electromagnetics*, Norwood, MA, Artech House Publishers, 2004.

5. P. Lorenz, J. V. Vital, B. Biscontini and P. Russer, "TLM-G: A Grid-Enabled Time-Domain Transmission-Line-Matrix System for the Analysis of Complex Electromagnetic Structures," *IEEE Transactions on Microwave Theory and Techniques*, **53**, 11, November 2005, pp. 3631-3637.

6. F. Khalil et al., "Electromagnetic Simulations via Parallel Computing: an Application Using Scale Changing Technique for Modeling of Passive Planar Reflectarrays in Grid Environment," IEEE International Symposium on Antennas and Propagation and USNC/URSI National Radio Science meeting, July 2008, San Diego, California.

7. A. Esposito and L. Tarricone, *Advances in Information Technologies for Electromagnetics*, Berlin, Springer, 2006.

8. NVIDIA Corporation Technical Staff, *NVIDIA CUDA Programming Guide 2.2*, NVIDIA Corporation, 2009.

9. 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.

10. S. Peng and Z. Nie, "Acceleration of the Method of Moments Calculations by Using Graphics Processing Units," *IEEE Transactions on Antennas and Propagation*, **AP-56**, 7, July 2008, pp. 2130-2133.

11. K. S. Yee, "Numerical Solution of Initial Boundary Value Problems Involving Maxwell's Equation in Isotropic Media," *IEEE Transactions on Antennas and Propagation*, **AP-14**, 4, May 1966, pp. 302-307.

12. A. Balevic et al., "Accelerating Simulations of Light Scattering Based on Finite-Difference Time-Domain Method with General Purpose GPUs," IEEE International Conference on Computational Science and Engineering, July 2008, pp. 327-334.