

Implementing a 2D FDTD scheme with CPML on a GPU using CUDA

Alvaro Valcarce and Jie Zhang

Centre for Wireless Network Design (CWIND)
University of Bedfordshire, Luton, Bedfordshire, UK
alvaro.valcarce@beds.ac.uk

Abstract: This paper describes the implementation of a 2D Finite-Difference Time-Domain (FDTD) scheme on a Graphics Processing Unit (GPU). The architecture used is the Compute Unified Device Architecture (CUDA) from NVIDIA, which provides a shared-memory parallel computing paradigm. Hence, the use of different memory access patterns in the FDTD simulation is explained. For computational efficiency, separate kernels are designed for the evaluation of different regions. Finally, simulation results are presented in the context of radio coverage prediction for mobile communications systems.

1 Introduction

FDTD algorithms have been traditionally used for the performance evaluation of diverse systems in small-scale scenarios (e.g. antennas, microwave circuits, etc). However, interest has grown recently on the characterization of large-scale radio channels. This includes those channels used by mobile telecommunications systems, such as indoor [1] and urban environments.

Traditional channel modeling methods such as empirical or semi-empirical are still used for its quick implementability and short running time. However, these suffer from a lack of precision in complex environments such as urban scenarios, where the different obstacles should be more accurately modeled. FDTD, which solves the Maxwell equations on a discrete grid, takes all wave interactions implicitly into account thanks to its Maxwellian formulation. However, FDTD is also a computationally intensive method, so it has not yet been widely applied to the modeling of radio channels.

In recent years, two hardware acceleration techniques have received most of the attention from the FDTD community. These are Field-Programmable Gate-Arrays (FPGAs) [2] and GPUs [3]. For instance, [4] claims speeds of approximately 75 Mcps¹ for a 2D implementation in an FPGA. However, due to market needs, these devices tend to be more costly than GPUs. Furthermore, GPUs are nowadays present in every personal computer, thus being more accessible for testing than FPGAs. Regarding computation speed, [5] achieves speeds of up to 450 Mcps with a graphics card similar to the one used in this paper.

FDTD is an iterative model, i.e. the information within each cell must be updated at each time step. This makes it very suitable for implementation on a parallel architecture. Therefore, FDTD is starting to appeal to the propagation models community as a suitable fast method to perform radio coverage prediction and channel modeling. For example, [6] makes use of a multi-resolution FDTD implementation on a GPU as a tool to model a microwave wireless channel. The objective of the current work is the implementation of a propagation model for its integration into a wireless networks system-level simulator. Thus, GPU computing using CUDA seemed the quickest and most flexible solution.

¹Mega cells per second

2 Tools

The implementation presented here is formulated for 2D scenarios, being thus suitable for coverage simulations in flat or near-flat urban and indoor environments. GSM, UMTS and WiMAX base stations use typically vertically polarized antennas. In order to predict the propagation of electromagnetic waves with vertical polarization in a 2D simulator, it is necessary to restrict the formulation of the Maxwell equations to the TM_z mode (electrical field polarized in the vertical direction). Such equations are well-known and can be found in [7].

Just as with other traditional parallel programming paradigms, the GPU is capable of executing several *threads* running simultaneously. With CUDA, the developer writes the *kernels*, which are the pieces of C code that each thread executes. The compiler converts then the kernels into the instruction set of the device. Different threads are always bundled together into a thread *block* in a way that allows them to cooperate by exchanging data through shared memory. Since a block contains only a limited number of threads, blocks can also be grouped together into a *grid*, thus increasing the total number of threads to be executed.

The multiprocessors within the GPU are then responsible for executing the different thread blocks. This is achieved by splitting each block into Single Instruction, Multiple Data (SIMD) groups named *warps*. It is important to balance the amount of resources that a block will need, because it might restrict the number of warps that a multiprocessor is able to execute simultaneously. The ratio of the number of active warps to the maximum amount of warps that can be active simultaneously is called the *multiprocessor occupancy* and it is the responsibility of the programmer to maximize it in order to exploit the hardware resources.

3 CUDA Implementation

A. The kernels

One of the advantages of the Convolutional Perfectly Matched Layer (CPML) is that it can be computed independently from the computational domain. This facilitates the implementation on an SIMD architecture because it eliminates the need for having to decide in each kernel, whether the current cell belongs to the Absorbing Boundary Condition (ABC) or to the Computational Domain. Kernels are executed faster when all of the threads within a warp follow the same execution path. If the deciding condition evaluates differently for two or more threads within the same warp, the execution of the threads will have to be serialized, losing therefore the advantage of parallelization.

This feature can be exploited by dividing the computation into different kernels for different parts of the scenario. According to this, five different regions have been identified: top and bottom along the x dimension, top and bottom along the y dimension, and the central computational domain. The current design has included the computation of the corners in the kernels of the x dimension, although other approaches are also valid. Furthermore, there are two types of fields (E and H) to be updated in each region, apart from the CPML auxiliary variables. Hence, the different kernels are specialized on the computation of specific regions. Such kernels are listed on Table 1 with $\Psi_{H_{x,y}}$, $\Psi_{H_{y,x}}$, $\Psi_{E_{z,x}}$ and $\Psi_{E_{z,y}}$ being discrete variables with nonzero values only in certain CPML regions (see [7] for further information). Then, the tasks that each kernel must perform in order to update the FDTD fields are: First, load the field values from the previous time iteration, then compute the new field values and finally, store the result in the memory.

In order to exploit the SIMD architecture of the GPU, the workload is divided between all of the threads within each block. This way, each thread updates a single cell using the content of its neighbors. As an example, the pseudocode² of the kernel that implements the electrical field update is shown in Algorithm 1.

²Note that in Algorithm 1, i and j refer to the matrix coordinates in the x and y dimensions, not the physical Yee coordinates.

Table 1: Computational objective of each kernel type.

| | |
|--|--|
| $\Psi_{H_{y,x}}$ and H_x in X-bottom including corners | $\Psi_{E_{z,x}}$ and E_z in X-bottom including corners |
| $\Psi_{H_{y,x}}$ and H_x in X-top including corners | $\Psi_{E_{z,x}}$ and E_z in X-top including corners |
| $\Psi_{H_{x,y}}$ and H_y CPML in Y-bottom | $\Psi_{E_{z,y}}$ and E_z in Y bottom |
| $\Psi_{H_{x,y}}$ and H_y CPML in Y-top | $\Psi_{E_{z,y}}$ and E_z update in Y top |
| H_x and H_y field in the computational domain | E_z field in the computational domain |

```

Calculate  $i$  and  $j$  for the current thread;
Load  $H_x|_{i,j}^n$  into shared memory;
if  $threadIndex.y = 0$  then
    | Load  $H_x|_{i,j-1}^n$  into shared memory;
end
Load  $H_y|_{i,j}^n$  into shared memory;
if  $threadIndex.x = 0$  then
    | Load  $H_y|_{i-1,j}^n$  into shared memory;
end
Synchronize threads;
Load material index;
Compute  $E_z|_{i,j}^{n+\frac{1}{2}}$ ;
Store result to global memory;

```

Algorithm 1: Electrical field update

B. Block and scenario size

The block size is also an important design parameter since larger blocks contain more threads. In this implementation, each thread loads one field value (two (H_x and H_y) for the E_z update kernel) into shared memory. There is thus a direct relationship between the block size and the amount of shared memory a kernel needs. However the amount of shared memory that each block can use is finite (see Table 2), so this imposes an upper limit to the block size. Furthermore, shared memory must be divided between all the blocks that are executed on a multiprocessor. Considering these limitations, several numerical experiments indicated that a block size of $(B_x, B_y) = (16, 8)$ is optimal for this implementation, thus yielding 128 threads per block and a *multiprocessor occupancy* of 100% for all kernels.

In order to maximize the instruction throughput by reducing divergence within warps, an efficient design is to match the block size to the geometry of the matrices. Ten cells are usually enough for the CPML to avoid waves reflecting back into the simulation scenario. However, since the maximum dimension of the block is 16, a 16-cell deep CPML has been used, thus reducing instruction branching within warps and providing lower reflection coefficients. Another important parameter in an FDTD simulation is the spatial step (Δ), which must be smaller than the smallest obstacle within the computational domain. Since one of the objectives of this work is to perform radio coverage simulations, the spatial step will be determined by the resolution with which the simulation scenario (a residential area, in this case) can be modelled.

The grid and block sizes must be defined accordingly so that the total number of threads matches the total number of FDTD cells. In the cases where the size of the scenario is not a multiple of the block size, there will inevitably be some threads that do not need to compute anything, thus decreasing the total instructions throughput. That is why it is recommended that the dimensions (N_x, N_y) in number of cells of the simulation scenario be as close as possible to a multiple of the block size. This can be easily done by adjusting Δ .

Table 2: Cards used for computation

| | GF 8600M GT | TESLA C870 |
|---------------------------|-------------|------------|
| Global Memory | 256 MB | 1.5 GB |
| Constant Memory | 64 KB | 64 KB |
| Clock Rate | 337.5 MHz | 1.35 GHz |
| Memory Bandwidth | 9.6 GB/s | 76.8 GB/s |
| Number of Multiprocessors | 4 | 32 |

CUDA uses the Floating-Point Standard and the *float* type for the representation of real numbers. Since the electrical and magnetic fields are real physical magnitudes, the value of a field corresponding to one cell can be stored as a float type in 4 bytes of memory. Therefore, this and the spatial step determine the memory requirements for the simulation of a specific scenario.

C. Memory access scheme

One of the main tasks that kernels need to perform is the reading and writing of large matrices from/to the *global memory* of the graphics card. This is the main bottleneck of the FDTD implementation so it is crucial to use an appropriate memory access scheme to obtain the maximum memory bandwidth. The graphics cards from NVIDIA have four different types of memory spaces [8], thus offering a versatile range of memory access patterns.

Since it is not possible with CUDA to synchronize the execution of different thread blocks or to pass information between them, each individual block has to load the update coefficients (material properties) from memory. Since there are usually 5 to 10 different materials present in a simulation, one block should be enough to load this information. This redundancy in the memory access can be reduced by making use of the constant memory space. *Constant memory* is backed by an on-chip cache. Thanks to this, a reading from constant memory accesses the global memory only on a cache miss. Furthermore, because this cache is physically located on each multiprocessor, almost all thread blocks will read from the cache once it is loaded.

Another fast memory space is the *shared memory*, which also resides on-chip. All threads within the same block exchange information by doing synchronized reads and writes onto this memory. Since the update of each cell in an FDTD algorithm is based on the previous values of the fields in its neighborhood, each cell (i, j) needs to be accessed at least twice in each iteration. In order to minimize access to global memory, the threads of a block first load the corresponding submatrix into shared memory (see algorithm 1), and later use these values to perform the computation. This way, the access to each cell in the global memory space is performed only once.

4 Results

To illustrate the above described implementation, coverage predictions of femtocells [9] in the 3.5 GHz frequency band have been performed. However, these predictions use the so-called *lower frequency* approximation [10], in which the simulation frequency is several times lower than the propagation frequency. Accuracy in the prediction is guaranteed by a measurements-based calibration of the material properties. In this case $f_{sim} \approx 400$ MHz and $N_\lambda = \lambda_{sim}/\Delta = 6$. An important objective is to perform fast simulations on standard PCs. Therefore and for benchmarking reasons, two different graphics cards have been used to perform the computation: an off-the-shelf and non-expensive laptop graphics card and a high-performance computing card. Their features are summarized on Table 2.

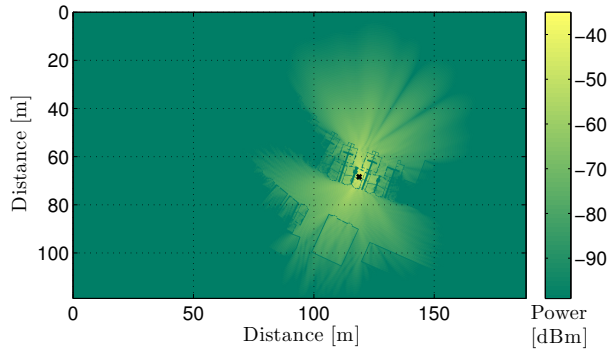


Figure 1: Femtocell coverage at $f = 3.5$ GHz and $P_{tx} = 15$ dBm.

The scenario is a residential area in a medium-size British town and with a size of approximately $(d_x, d_y) = (188, 199)$ meters. Using $\Delta = 12$ cm the scenario can be sampled into a matrix of size $(n_x, n_y) = (992, 1568)$ (roughly 1.5 million cells). Since the width and height of a block of threads are 16 and 8, the environment was extended by adding some extra cells containing air in both dimensions. After adding the CPML cells, the total size of the matrix to simulate is $(1056, 1632)$. Thus, since each cell is stored as a floating point number in 4 bytes, each matrix occupies $4 \cdot N_x \cdot N_y \approx 6.7$ MB. Since FDTD is a memory-intensive algorithm, the memory bandwidth is the most important feature to be sought in a GPU for this purpose. Note that four large matrices (E_z, H_x, H_y , index of materials) must be read, and three matrices (E_z, H_x and H_y) must be written into global memory in each iteration. This amounts, in this simulation, to around 46 MB of data to be read/written. It is easily tested that this number grows quickly for larger scenarios and/or grid resolutions.

For the matrix size given above, a total of 4000 FDTD iterations have been performed. Using the first GPU shown in Table 2, the total computation time has been of 46 seconds, which compares favorably with other existing propagation models. This algorithm has also been implemented in Matlab, which makes use of the AMD Core Math Library (ACML) for efficient matrix operations. The Matlab code was run on a computer equipped with an AMD Athlon 64 X2 Dual Core Processor 4600+ at 2.41 GHz and 3.25 GB of RAM. Applying the same configuration parameters as with the GPU simulation, the total computation time in Matlab was of approximately 78 minutes. This indicates that for the implementation of FDTD-based algorithms a standard off-the-shelf graphics card for laptops like the GeForce 8600M GT provides an speedup of around 100 times over optimized Linear Algebra libraries. Furthermore, when the computation is run on the high-performance computing card (TESLA C870), the simulation time is reduced to 8.6 seconds and the achieved computation speed is 760 Mcps, which is superior to the 450 Mcps achieved by [5]. With respect to the Matlab computation the obtained speedup is of 540. Figure 2 compares the measured running times as well as the performance of this FDTD implementation.

5 Conclusions

This study has presented design considerations to be considered when using CUDA to implement Finite-Difference algorithms. An efficient memory access scheme has been introduced that maps the execution threads to the different cells within the Yee lattice. Furthermore, the applicability of each memory space to the implementation of an FDTD algorithm for the minimization of memory accesses and the increase of simulation speed has been introduced. Besides and in order to minimize threads serialization and exploit the parallel capabilities of the GPU, an efficient way of subdividing the absorbing boundaries to minimize instructions branching within threads of the same warp has been presented.

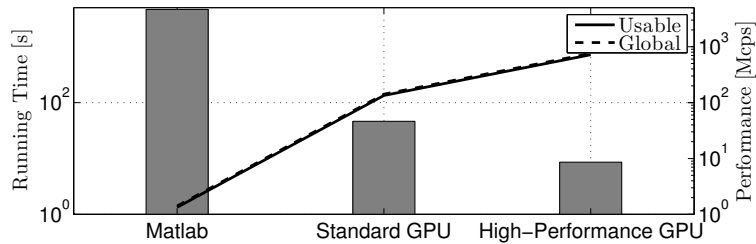


Figure 2: The bar graph represents the running time (left axis) of the algorithm. The line graph (right axis) represents the simulation speed in terms of Megacells per second for the overall scenario (including the ABC) and for only the computational domain (usable part without the ABC).

Finally, simulation results for a residential area have been presented, in which the accuracy of FDTD for coverage prediction is evinced. Since this method is a direct approximation to the Maxwell equations, the only limit to the achievable precision is in the proper calibration and configuration of the different FDTD parameters. It has been shown that in very short running times, it is possible to perform accurate predictions that implicitly consider all the distortions suffered by the electromagnetic propagation.

6 References

- [1] Michel Thiel and Kamal Sarabandi. A Hybrid Method for Indoor Wave Propagation Modeling. *IEEE Transactions on Antennas and Propagation*, 56(8):2703–2709, August 2008.
- [2] EM Photonics. <http://www.emphotonics.com/>, 2001.
- [3] So Poman. Time-Domain Computational Electromagnetics Algorithms for GPU Based Computers. In *EUROCON*, pages 1–4, Warsaw, Poland, Sep. 2007.
- [4] R. N. Schneider, L. E. Turner, and M. Okoniewski. A software-coupled 2D FDTD hardware accelerator. In *IEEE Antennas Propagation Society International Microwave Symposium*, volume 2, pages 1692–1695, 2004.
- [5] Piotr Sypek, Adam Dziekonski, and Michal Mrozowski. How to Render FDTD Computations More Effective Using a Graphics Accelerator. *IEEE Transactions on Magnetics*, 45(3):1324–1327, March 2009.
- [6] G. S. Baron, E. Fiume, and C. D. Sarris. Graphics hardware accelerated multiresolution time-domain technique: development, evaluation and applications. *IET Microwaves, Antennas & Propagation*, 2(3):288–301, April 2008.
- [7] Allen Taflove and Susan C. Hagness. *Computational Electrodynamics: The Finite-Difference Time-Domain Method*. Artech House, 3rd edition, 2005.
- [8] NVIDIA Corporation. *CUDA Programming Guide*, August 2009.
- [9] David Lopez-Perez, Alvaro Valcarce, Guillaume De La Roche, and Jie Zhang. OFDMA Femtocells: A Roadmap on Interference Avoidance. *IEEE Communications Magazine*, 47(9):41–48, September 2009.
- [10] Alvaro Valcarce et al. Applying FDTD to the coverage prediction of WiMAX femtocells. *EURASIP Journal on Wireless Communications and Networking*, March 2009.