

# A GPU approach to FDTD for Radio Coverage Prediction

Alvaro Valcarce, Guillaume De La Roche, Jie Zhang  
Centre for Wireless Network Design (CWIND)  
University of Bedfordshire, Luton, Bedfordshire, UK  
alvaro.valcarce@beds.ac.uk

**Abstract**—The benefits of using Finite-Difference alike methods for coverage prediction comprise highly accurate electromagnetic simulations that serve as a reliable input for wireless networks planning and optimization algorithms. These algorithms usually require several thousands of iterations in order to find the optimal network configuration, so to obtain results within reasonable computation times, the applied propagation models must be as fast as possible. In this study an implementation-oriented analysis on the suitability of using Graphics Processing Units (GPU) to perform Finite-Difference Time-Domain simulations is carried out. We believe that the recently released Compute Unified Device Architecture (CUDA) technology has opened the door for computational intensive algorithms such as FDTD to be considered for the first time as a precise and fast propagation model to predict radio coverage.

## I. INTRODUCTION

With the very fast deployment of wireless networks during the last decades, the need for radio network planning tools that aid operators to design and optimize their wireless infrastructure is rising. In order to increase the reliability of these tools, accurate radio wave propagation models are necessary.

The currently used optimization tools make use of empirical or semi-empirical models [1] [2] due to its quick implementability and short running time. But these models suffer from a lack of precision in complex environments such as urban scenarios, where the different obstacles should be more accurately modeled. It is thus necessary to make use of deterministic models based on physical laws that try to compute the reflections, diffractions, transmissions and scattering on obstacles.

Ray-tracing and geometric-like models [3] [4] have been proposed. They have shown to be very efficient, except in too severe environments, where too many reflections need to be computed, and where the diffraction phenomena, even with the Uniform Theory of Diffraction (UTD), are difficult to simulate. Another well known approach to compute radio wave propagation is the Finite-Difference Time-Domain (FDTD) model [5], which solves the Maxwell equations on a discrete grid. This method is attractive because all the reflections/diffractions are implicitly taken into account by its formulation. However, since FDTD tries to simulate an unbounded region in a size-restricted area, an Absorbing Boundary Condition (ABC) such as the Convolutional Perfectly Matched Layer (CPML) [6] must be used in order to avoid reflections on the border.

On the other hand, FDTD is a very computationally intensive model, so it has not yet been widely applied for radio coverage predictions. In [7] an adaptation of a discrete model called Parflow has been proposed in the frequency domain, reducing a lot the complexity of the problem but bypassing the time related information such as the delays of the different rays. In [8] an hybridization of FDTD with a geometric model is proposed. In this approach FDTD is applied only in small complex areas and combined with ray-tracing for the more open space regions, but the running time of such an approach is still too large for the purpose of radio network planning.

## II. RELATED WORK

During the last years, two hardware acceleration techniques have received most of the attention from the FDTD community. These are the implementations on Field-Programmable Gate-Arrays (FPGAs) [9] and on Graphics Processing Units [10]. For instance, [11] claimed to have achieved speeds of around 75 Mcps<sup>1</sup> for a 2D implementation in an FPGA. However and due to market needs, these devices tend to be more costly than GPUs. On top of that, GPUs are nowadays present in every personal computer, being more accessible for testing than FPGAs. Furthermore, we must mention that other models such as ray-tracing have been also tested under GPU architectures [12].

With the development of new programmable graphics hardware, novel solutions to compute electromagnetics are being already implemented on GPUs [13]. Graphics chipsets are becoming cheap and powerful, and their architecture is very well adapted to parallel algorithms.

FDTD is however an iterative model, i.e. the information within each pixel must be properly updated at each iteration. The fact that all the pixels must be simultaneously recomputed makes it very suitable for implementation on a parallel architecture [14]. Thanks to this, FDTD is starting to appeal to the propagation models community as a suitable fast method to perform radio coverage and channel modeling. For example, [15] makes use of a multi-resolution FDTD implementation on a GPU as a tool to model a microwave wireless channel, and [16] presents a hybrid UTD-FDTD approach for this very purpose.

<sup>1</sup>Mega cells per second

Since our objective was to implement a propagation model for its integration into a generic wireless networks simulator, GPU computing using CUDA seemed to be the quickest and most flexible solution. We therefore present in this paper, a potential GPU implementation of an FDTD model. This approach, based on the recently released Compute Unified Device Architecture (CUDA) from NVIDIA is adapted to simulate radio coverage, and its performance in terms of speed and accuracy is evaluated taking the COST 231 Munich [17] city test environment as reference.

### III. FDTD, CPML AND CUDA

Our implementation is formulated for 2D scenarios, being thus very suitable to perform coverage simulations in flat or near-flat urban and indoor environments. Although the memory requirements will be substantially higher, a 3D implementation can be easily derived by porting the traditional Yee [5] formulae to the memory and execution scheme presented in this paper.

GSM, UMTS and WiMAX typically use vertically polarized antennae. 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). Following the terminology given in [18] it is straightforward to deduce such FDTD and CPML expressions. To update the magnetic field  $H$  and electrical field  $E$  in a discrete grid sampled with a spatial step of  $\Delta$ , the  $TM_z$  formulae are as follows.

$$H_x|_{i,j+\frac{1}{2}}^{n+1} = H_x|_{i,j+\frac{1}{2}}^n - D_b|_{i,j+\frac{1}{2}} \cdot \left[ \frac{E_z|_{i,j+1}^{n+\frac{1}{2}} - E_z|_{i,j}^{n+\frac{1}{2}}}{\Delta\kappa_{y_{j+\frac{1}{2}}}} + \Psi_{H_{x,y}}|_{i,j+\frac{1}{2}}^{n+\frac{1}{2}} \right] \quad (1)$$

$$H_y|_{i+\frac{1}{2},j}^{n+1} = H_y|_{i+\frac{1}{2},j}^n + D_b|_{i+\frac{1}{2},j} \cdot \left[ \frac{E_z|_{i+1,j}^{n+\frac{1}{2}} - E_z|_{i,j}^{n+\frac{1}{2}}}{\Delta\kappa_{x_{i+\frac{1}{2}}}} + \Psi_{H_{y,x}}|_{i+\frac{1}{2},j}^{n+\frac{1}{2}} \right] \quad (2)$$

$$E_z|_{i,j}^{n+\frac{1}{2}} = C_a|_{i,j} \cdot E_z|_{i,j}^{n-\frac{1}{2}} + C_b|_{i,j} \cdot \left[ \Psi_{E_{z,x}}|_{i,j}^n - \Psi_{E_{z,y}}|_{i,j}^n + \frac{H_y|_{i+\frac{1}{2},j}^n - H_y|_{i-\frac{1}{2},j}^n}{\Delta\kappa_{x_i}} - \frac{H_x|_{i,j+\frac{1}{2}}^n - H_x|_{i,j-\frac{1}{2}}^n}{\Delta\kappa_{y_j}} \right], \quad (3)$$

where  $D_b$ ,  $C_a$ , and  $C_b$  are the update coefficients that depend on the properties of the different materials, and  $\Psi_{H_{x,y}}$ ,  $\Psi_{H_{y,x}}$ ,  $\Psi_{E_{z,x}}$  and  $\Psi_{E_{z,y}}$  are discrete variables with nonzero values only in some CPML regions. For a more in-depth development of the variables used in these formulae the reader is referred to [18].

Just as with the traditional parallel programming paradigm, the GPU is a device capable of executing several *threads* running simultaneously. With CUDA, the developer is responsible for writing the *kernels*, which are the pieces of C code that each thread will execute. The compiler will then convert it

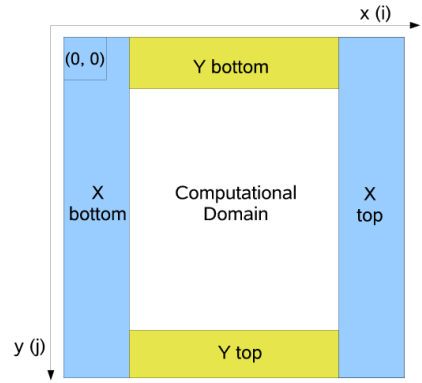


Fig. 1. Division of the scenario

to the instruction set of the device for execution. 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*, increasing therefore the total number of threads that will be executed. By looking at (3), it is easy to see that the same computation must be applied to all the cells  $(i, j)$  of the computational domain. The parallel programming model provided by CUDA is thus more than suitable for this purpose.

The multiprocessors within the GPU are responsible for executing the different thread blocks. This is achieved by splitting each block into SIMD (Single Instruction, Multiple Data) 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.

### IV. 2D IMPLEMENTATION

One of the main advantages of the CPML absorbing boundary condition is that its computation can be performed completely independently from the computational domain. This eases the implementation on a SIMD architecture because it eliminates the need for having to decide in each kernel, whether the current pixel belongs to the absorbing border or to the Computational Domain. Kernels are most effective 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, loosing therefore the advantage of parallel computing.

A direct conclusion is thus that dividing the computation into several kernels for the different parts of the scenario is more suitable than having one single kernel computing the whole environment. We have thus designed five different types of kernels that will compute the different parts of the

scenario. Figure 1 displays this division of the environment and indicates the terminology that will be used to reference the pixel coordinates:

- 1) CPML update in X bottom (including corners).
- 2) CPML update in X top (including corners).
- 3) CPML update in Y bottom.
- 4) CPML update in Y top.
- 5) Fields update in the computational domain.

The steps that each kernel must follow in order to perform the FDTD field updates, can be further divided into three different parts:

- 1) Loading of the previous field values.
- 2) Computation of the new field values.
- 3) Writing of the result to memory.

In order to exploit the SIMD architecture of the GPU to carry out the above mentioned steps, the workload is divided between all of the threads that will be executed in the grid of thread blocks. As an example, the pseudocode<sup>2</sup> of the kernel that implements the electrical field update is shown in Algorithm 1.

```

Calculate the  $i$  and  $j$  indices;
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
Load material index;
Compute  $E_z|_{i,j}^{n+\frac{1}{2}}$  according to (3);
Store result to global memory;

```

**Algorithm 1:** Electrical field update

The next important design decision consists on choosing an appropriate block size. The bigger the block, the more threads it will contain. In our implementation of the FDTD algorithm, each thread is responsible for loading one field value (two 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 it will need. Of course, the amount of shared memory that each block can use is finite (see table I), so this imposes an upper limit for the size of the block. Furthermore, the amount of shared memory must be divided between all the blocks that are executed on a multiprocessor. Considering this limitations, several numerical experiments have proven that a block size of  $(B_x, B_y) = (16, 8)$  is optimal, yielding 128 threads per block and a *multiprocessor occupancy* of 100% for all kernels. Using the kernel that updates the magnetic field as an example, figure 2 shows the effect on multiprocessor

<sup>2</sup>Note that in Algorithm 1,  $i$  and  $j$  refer to the matrix coordinates in the  $x$  and  $y$  dimensions, not the physical Yee coordinates of (1), (2) and (3).

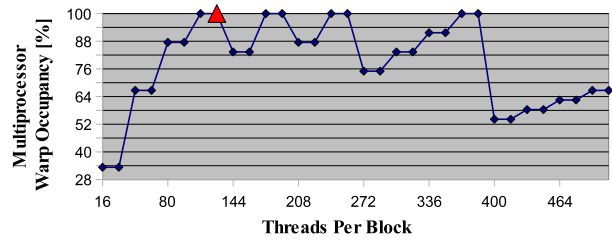


Fig. 2. Multiprocessor Occupancy

occupancy of using different block sizes. The red triangle indicates the multiprocessor occupancy reached with our block design.

In order to maximize the instruction throughput by reducing divergency 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 streets of the simulation scenario. But since the maximum dimension of the block is 16, a 16-cell CPML depth will be applied, reducing thus instruction branching within warps and obviously providing even lower reflection coefficients.

One of the most important parameters of a FDTD simulation is the spatial step ( $\Delta$ ), which must be smaller than the smallest obstacle within the computational domain. Since our objective is to perform urban coverage simulations, the spatial step will be determined by the resolution with which the simulation scenario (a city in this case) can be modelled.

TABLE I  
CARDS USED FOR COMPUTATION

	GF 8600M GT	TESLA C870
Global Memory	256 MB	1.5 GB
Constant Memory	64 KB	64 KB
Shared Memory per Block	16 KB	16 KB
Clock Rate	337.5 MHz	1.35 GHz
Memory Bandwidth	9.6 GB/s	76.8 GB/s
Number of Multiprocessors	4	32
Algorithm Running Time:	43 s	8 s

In our design, each thread will be responsible for calculating one pixel of the fields' matrices. The grid and block sizes must be therefore 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 will not need to compute anything, decreasing therefore the total instructions throughput. That is why the dimensions of the simulation scenario must be as close as possible to a multiple of the block size. CUDA uses the Floating-Point Standard and the *float* type for the representation of real numbers. Since the electrical and magnetic fields are always real physical magnitudes, the value of a field corresponding to one cell can be stored as a float type in 4 bytes of memory. This value together with the spatial step will determine the memory requirements for the simulation of a specific scenario.

As can be deduced from algorithm 1 and section III, the

calculation of the different fields consists mainly of some multiplications and additions, usually combined together by the compiler into a multiply-add (FMAD) instruction. On the other hand, several matrices of the size of the scenario must be read and written in each iteration from the graphics card global memory. The access to memory will thus be the bottleneck of the FDTD algorithm, so it is crucial to use an appropriate memory access scheme to get the maximum memory bandwidth. The graphics cards from NVIDIA have four different types of memory spaces [19] and offer thus an extremely versatile range of memory access patterns.

Since it is not possible with CUDA to synchronize the execution of the different thread blocks or to pass information between them, each individual block will have to load the update coefficients from memory itself. This redundancy in the memory access can be greatly reduced by making use of the constant memory space. Constant memory is backed by a small on-chip cache. Thanks to this, a reading from constant memory will access the device memory only on a cache miss. Furthermore and because this cache is physically located on each multiprocessor, almost all the thread blocks will read from the cache once it is loaded (usually by the first block to be executed).

Another fast memory space is the shared memory, which also resides on-chip. It is called shared because all the threads within the same block can exchange information by doing synchronized reads and writes. Since each cell in an FDTD algorithm is updated based on the previous values of the fields in its neighborhood, each pixel  $(i, j)$  will need to be accessed at least twice to compute one field update. In order to minimize access to global memory, the threads of a block will first load the corresponding submatrix into shared memory (see algorithm 1), and later use these values to perform the computation.

The GPU can read 4, 8 and 16 byte words from global memory in only one instruction. Moreover, some memory accesses can even be coalesced into a single and contiguous memory access. But to exploit this feature, there are some requirements regarding the way that the information is stored in global memory [19]. These include the fact that the width of a matrix stored in global memory is a multiple of 16. This is therefore another argument in support of simulating scenarios whose width is a multiple of 16.

## V. RESULTS

Taking the Munich scenario [17] as a reference, a value of 2 meters for the spatial step provides enough precision to accurately model the width of the streets and the buildings along with their corners. But in order to avoid anisotropy in the phase velocity of the propagating wave, it is very important in FDTD to establish correctly the proportion between the wavelength and the spatial step. Such relationship is better explained throughout the following formula.

$$\Delta = \frac{\lambda}{N_\lambda} \quad (4)$$

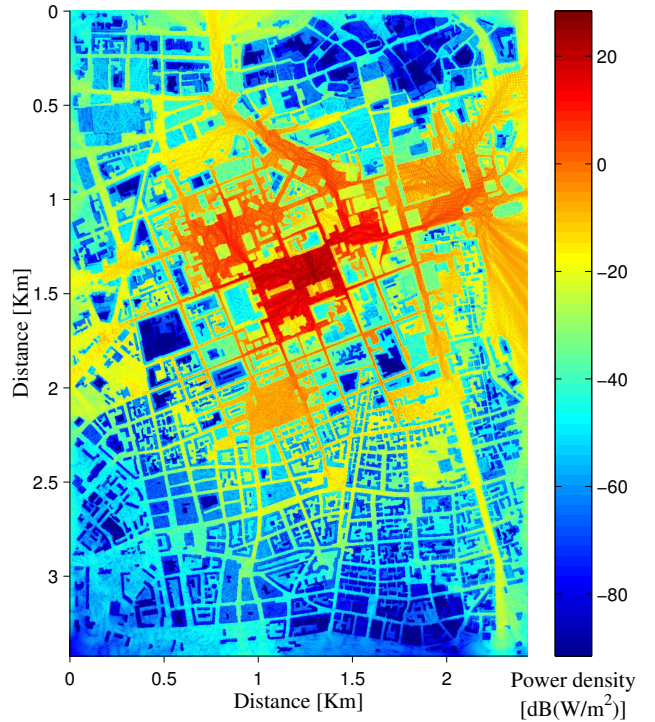


Fig. 3. FDTD Prediction in Munich

For the presented simulations  $N_\lambda = 10$  has been used, which gives a velocity-anisotropy error [18] of:

$$\Delta \tilde{v}_{aniso} \approx 0.9\% \quad (5)$$

The Munich scenario has a size of approximately  $(d_x, d_y) = (2.4, 3.4)$  kilometers, with  $d_x$  the length in the  $x$  dimension and  $d_y$  the length in the  $y$  dimension. Using  $\Delta = 2m$  the scenario can be sampled into a matrix of size  $(n_x, n_y) = (1204, 1704)$  (roughly 2 million cells). Since the width and height of a block of threads are 16 and 8, the environment has been extended by adding some extra cells containing air in both dimensions. This generates an scenario matrix of size  $(1216, 1704)$ . After adding the CPML cells, the total size of the matrix to simulate is  $(1248, 1736)$ , with an omnidirectional source located at position  $(663, 707)$ . The maximum distortion of the wavefront due to anisotropy of the wave velocity occurs when the signal has travelled  $n_{tc} = 1744 - 707 = 1037$  cells from the source in the vertical direction. Using (5) this distortion is estimated in  $\tilde{v}_{aniso} n_{tc} \approx 9$  cells. With bigger size scenarios, this distortion becomes even bigger and we therefore recommend to use not less than  $N_\lambda = 10$  for simulations in urban environments.

From (4),  $\lambda = 20m$  is calculated, which implies that the frequency used for this simulation is not equal to that of the measurements (GSM at 900 MHz), but  $f_{sim} = 15$  MHz. In [20] we already analyzed the consequences of reducing the simulation frequency, which requires a proper model calibration in order to compensate for the error introduced by such frequency reduction.

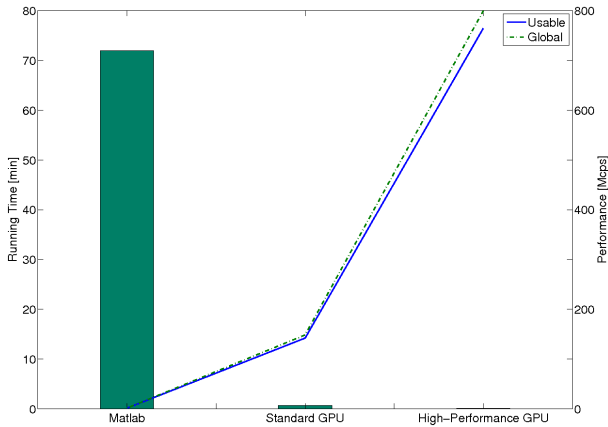


Fig. 4. The bar graph shows the running time (left axis) of the presented algorithm on three different platforms. The plot graph (right axis) presents the performance in terms of Megacells per second for the global scenario (including the ABC) and its usable part (without the ABC).

Our main objective is to perform fast and accurate simulations on standard PCs. Two different graphics cards have been used to perform the computation: an off-the-shelf and non-expensive laptop graphic card and a high-performance computing card. Their characteristics are gathered on table I. Since FDTD is a memory-intensive algorithm, the memory bandwidth is the most important feature of a graphic card to run it in the shortest period of time, so it is easy to predict that our algorithm running time will be much faster in non-mobile graphic card models.

For the matrix size given above, a total of 3000 FDTD iterations in the Munich environment have been performed. Using the first GPU shown in table I, the total computation time has been of 43 seconds, which compares favorably with other existing propagation models. We have also implemented the FDTD-CPML method 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 memory. Applying the same configuration parameters as with the GPU simulation, the total computation time in Matlab was of approximately 72 minutes. This indicates that for the implementation of FDTD-based algorithms a standard off-the-shelf graphics card for laptops like ours (GeForce 8600M GT) provides a speedup of around 100 times over highly 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 seconds with respect to the Matlab computation (speedup of 540). Figure 4 compares the measured running times as well as the performance of this FDTD implementation.

Table II presents the properties of the materials used in this simulation, where  $\mu_r$ ,  $\epsilon_r$  and  $\sigma$  are the relative magnetic permeability, relative electric permittivity and the electric conductivity. The configuration parameters of the CPML form a complex tensor (see [18]) and are described by  $a$ ,  $\kappa$  and

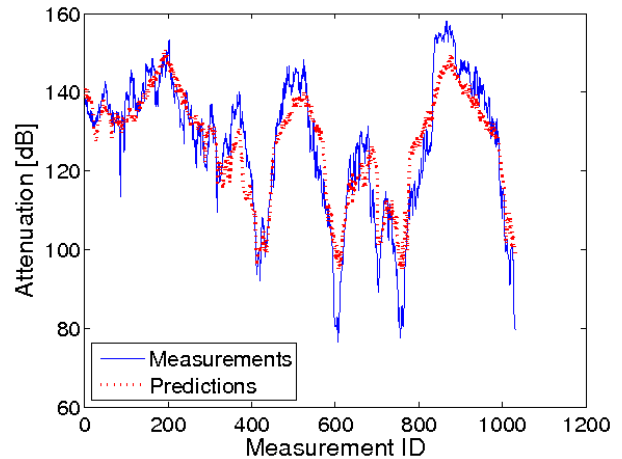


Fig. 5. Comparison of measurements and predictions

TABLE II  
PROPERTIES OF THE MATERIALS

	$\mu_r$	$\epsilon_r$	$\sigma$
Air	1	1	0
Concrete	4.5	6.5	0.005

$\sigma$ . Their values are shown in table III, being  $n$  the width in number of cells of the ABC,  $m$  the polynomial grade for  $\kappa$  and  $\sigma$ , and  $m_a$  the polynomial grade for  $a$ .

In figure 3 the high level of detail achieved by using our FDTD simulation to predict urban radio coverage can be verified. We also observe that, as opposed to other techniques such as ray-tracing, with FDTD an unlimited number of reflections can be very accurately modelled. The street canyon effects are also clearly displayed, along with diffractive distortions at corners of buildings at the end of some streets. We invite the reader to zoom into the coverage prediction of figure 3 to appreciate these details.

Since the different field updates are calculated by taking derivatives on the other fields, continuity must exist in the transition from one cell to the adjacent, i.e. no cell within the scenario can be skipped from the calculation. As seen from figure 3, even the pixels that correspond to the inside of the buildings have been computed, evincing the high attenuation introduced by the walls of the buildings for indoor reception. Nevertheless, the outdoor-to-indoor power prediction in this case must be considered optimistic due to the lack of information regarding the inner structure of the different buildings.

By comparing the measurements of the Munich scenario against our predictions, an RMSE of 8.1 dBs is obtained, which is low enough considering that no calibration of materials has yet been applied. In figure 5 a comparison of the accuracy of these attenuation predictions is displayed, where an overestimation of the path loss for locations that are close to the antenna due to the 2D approximation is shown.

## VI. CONCLUSIONS

This study has presented the design considerations that must be taken into account when using CUDA to implement

TABLE III  
PROPERTIES OF THE CPML

$n$	$m$	$m_a$	$\alpha_{max}$	$\kappa_{max}$	$\sigma_{max}$
16	3	1	0.001	1.1	0.0526

Finite-Difference algorithms. We have introduced an efficient memory access scheme that matches 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 to minimize memory accesses and increase simulation speed is 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. We have explained the tasks that are assigned to each execution thread, and carefully shaped the dimensions of each threads block.

Finally, simulation results for an urban environment are presented where the extreme 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 highly accurate predictions that implicitly consider all the distortions suffered by the propagation of an electromagnetic wave. Since this is a 2D implementation, we believe that this model is specially appropriate for near-flat urban as well as indoor environments.

Future lines of research will include studying potential calibration algorithms such as heuristics in order to minimize the error introduced by the lower frequency approximation [20]. Besides and with the advent of new and powerful graphic cards, our final goal is to design a complete 3D FDTD-based propagation model and to further reduce the computation time in order to integrate this model into the upcoming wireless networks planning and optimization tools, being designed within our research group.

#### ACKNOWLEDGMENT

This work is supported by the EU FP6 "RANPLAN-HEC" project on 3G/4G Radio Access Network Design under grant number MEST-CT-2005-020958.

#### REFERENCES

[1] S. Seidel and T. Rappaport, "914 MHz path loss prediction models for indoor wireless communications in multifloored buildings." *IEEE*

*transactions on Antennas and Propagation*, vol. 40, no. 2, pp. 207–217, 1992.

[2] A. Valcarce, *et al.*, "Empirical Propagation Model for WiMAX at 3.5 GHz in an Urban Environment," *Microwave and Optical Technology Letters*, vol. 50, no. 2, pp. 483–487, February 2008.

[3] S. Fortune, "Algorithms for prediction of indoor radio propagation," *AT&T Bell Laboratories report*, January 1998.

[4] F. Aguado, F. Perez, and A. Formella, "Indoor and outdoor channel simulator based on ray tracing," in *IEEE Vehicular Technology Conference*, Phoenix, Arizona, May 1997.

[5] K. Yee, "Numerical Solution of Initial Boundary Value Problems Involving Maxwells Equations in Isotropic Media," *IEEE Transactions on Antennas and Propagation*, vol. 14, pp. 302–307, May 1966.

[6] J. A. Roden and S. D. Gedney, "Convolutional PML (CPML): An efficient FDTD implementation of the CFS-PML for arbitrary media," *Microwave and Optical Technology Letters*, vol. 27, pp. 334–339, June 2000.

[7] J.-M. Gorce, K. Jaffres-Runser, and G. de la Roche, "Deterministic approach for fast simulations of indoor radio wave propagation," *IEEE Transactions on Antennas and Propagation*, vol. 55, pp. 938–942, March 2007.

[8] Y. Wang, S. Safavi-Naeini, and S. K. Chaudhuri, "A hybrid technique based on combining ray tracing and ftd methods for site-specific modeling of indoor radio wave propagation," *IEEE Transaction on Antennas and Propagation*, vol. 48, no. 5, pp. 743–754, May 2000.

[9] "EM Photonics," <http://www.emphotonics.com/>, 2001.

[10] D. K. Price, J. R. Humphrey, and E. J. Kelmelis, "GPU-based accelerated 2D and 3D FDTD solvers," in *Physics and Simulation of Optoelectronic Devices XV. Edited by Osinski, Marek; Henneberger, Fritz; Arakawa, Yasuhiko. Proceedings of the SPIE, Volume 6468, pp. 646806 (2007).*, ser. Presented at the Society of Photo-Optical Instrumentation Engineers (SPIE) Conference, vol. 6468, Feb. 2007.

[11] R. N. Schneider, L. E. Turner, and M. Okoniewski, "A software-coupled 2D FDTD hardware accelerator," in *IEEE Antennas Propagation Sociey International Microwave Symposium*, vol. 2, 2004, pp. 1692–1695.

[12] T. Rick and R. Mathar, "Fast edge-diffraction-based radio wave propagation model for graphics hardware," in *International ITG Conference on Antennas (INICA)*, Mar. 2007, pp. 15–19.

[13] S. Poman, "Time-Domain Computational Electromagnetics Algorithms for GPU Based Computers," in *EUROCON*, Warsaw, Poland, Sep. 2007, pp. 1–4.

[14] W. Yu, R. Mittra, T. Su, Y. Liu, and X. Yang, *Parallel Finite-Difference Time-Domain Method*. Artech House, 2006.

[15] G. S. Baron, E. Fiume, and C. D. Sarris, "Graphics hardware accelerated multiresolution time-domain technique: development, evaluation and applications," *IET Microwaves, Antennas & Propagation*, vol. 2, no. 3, pp. 288–301, Apr. 2008.

[16] S. Reynaud, Y. Cocheril, R. Vauzelle, C. Guiffaut, and A. Reineix, "Hybrid FDTD/UTD Indoor Channel Modeling. Application to Wifi Transmission Systems," in *Vehicular Technology Conference (VTC-Fall)*. Hyatt Regency: IEEE, Sep. 2006.

[17] "COST 231 - Urban Micro Cell Measurements and Building Data," *Mannesmann Mobilfunk GmbH*, 1996.

[18] A. Taflove and S. C. Hagness, *Computational Electrodynamics: The Finite-Difference Time-Domain Method*, 3rd ed. Artech House, 2005.

[19] *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*, 1st ed., NVIDIA Corporation, Nov. 2007.

[20] A. Valcarce, G. De La Roche, and J. Zhang, "On the use of a lower frequency in finite-difference simulations for urban radio coverage," in *VTC-Spring*, Singapore, May 2008.