Noname manuscript No.
(will be inserted by the editor)

Development of a unified FDTD-FEM library for electromagnetic analysis with CPU and GPU computing

Jorge Francés · Sergio Bleda · Sergi Gallego · Cristian Neipp · Andrés Márquez · Inmaculada Pascual · Augusto Beléndez

Received: date / Accepted: date

Abstract The present paper describes an optimized C++ library for the study of Electromagnetics. The implementation is based on the Finite-Difference Time-Domain method for transient analysis, and the Finite Element Method for electrostatics. Both methods share the same core and are optimized for CPU and GPU computing. To illustrate its running, FEM method is applied for solving Laplace's equation analyzing the relation between surface curvature and electrostatic potential of a long cylindrical conductor, whereas FDTD is applied for analyzing Thin Film Filters at optical wavelengths. Furthermore, a comparison of the performance of both CPU and GPU versions is analyzed as a function of the grid size simulation. This approach allows the study of a wide range of electromagnetic problems taking advantage of the benefits of each numerical method and the computing power of the modern CPUs and GPUs.

Keywords Electromagnetic analysis · Finite-Difference Time-Domain · Finite Element Method · electrostatic potential · Thin Film Filters · Optical wavelengths · Graphics Processing Units

PACS 42.25.Bs · 02.70.Bf · 02.70.Dh

Mathematics Subject Classification (2000) 78A97 · 81V80 · 80M50

J. Francés · S. Bleda · S. Gallego · C. Neipp · A. Márquez
Department of Physics, Systems Engineering and Signal Theory, University of Alicante, Spain
University Institute of Physics to Sciences and Technologies, University of Alicante, Spain
E-mail: jfmonllor@ua.es

I. Pascual
Department of Optics, Pharmacology and Anatomy, University of Alicante, Spain
University Institute of Physics to Sciences and Technologies, University of Alicante, Spain

A. Beléndez
Department of Physics, Systems Engineering and Signal Theory, University of Alicante, Spain
University Institute of Physics to Sciences and Technologies, University of Alicante, Spain

1 Introduction

The increasing processing power of the modern CPUs has allowed to deal with inaccessible problems from different areas such as Electromagnetics, Optics and Acoustics. More recently, Graphics Processing Units (GPUs) have become an appealing alternative since they provide even more processing power than CPUs. Thus, the manufacturers of GPUs have redesigned them thinking also in numerical computing, like NVidia has done with the new Fermi family [1]. Unfortunately, many numerical methods are not suitable to be directly implemented in a GPU and their adaptation is still in progress.

For the present paper we have implemented two different numerical methods for Electromagnetics: Finite Element Method (FEM) and the Finite-Difference Time-Domain (FDTD) method. For each method we have developed two different versions, one for the CPU and another for the GPU. Each version required a thorough analysis of the algorithm operations, a rearrangement of the instructions and a correct memory alignment of the data structures.

The CPU versions have been designed taking into account the programming skills of the average scientific programmer, using all the automatic optimization facilities provided by modern compilers, which are mainly focused on parallelize the main loops of the algorithms [2]. Then, the GPU improvement is computed comparing with the sequential implementation. This analysis can be interesting to those researchers that work with these or similar numerical methods and are considering GPU computing instead of learning optimization techniques based on low level programming such as SSE with parallel schemes (OpenMP or MPI).

Since CPU and GPU programming are quite different, it is not always trivial to port a numerical method to the GPU and it will be useful to determine if it worths the effort. Thus, we have focused our study on analyzing the performance achieved as a function of the simulation size between the CPU and GPU approaches. The library allows to use both FEM and FDTD for solving a wide range of electromagnetic problems. Specifically, to check it, we have chosen two different experiences, one suitable for each numerical method since FEM is used for static problems and FDTD for transient analysis [3,4].

2 Theory

This section gives a brief summary of the two experiments used to check the implementation of both numerical methods. In the first place, FEM is used for solving the Laplace's equation in Electrostatics, comparing the analytical expressions developed in [5] for analyzing the relation between the surface curvature of an isolated charged conductor with uniform cross section and the resultant electrostatic potential (see Fig. 1a). In the second place, FDTD is used for analyzing optical diffractive elements such as Thin Film Filters (TFF). Specifically, it is applied for analyzing the reflectance of High-Reflecting Coatings (HRCs') [6] at optical wavelengths (see Fig. 1b). HRC's have many applications such as photovoltaic cells and Micro-Electro-Mechanical Systems (MEMS) [7].

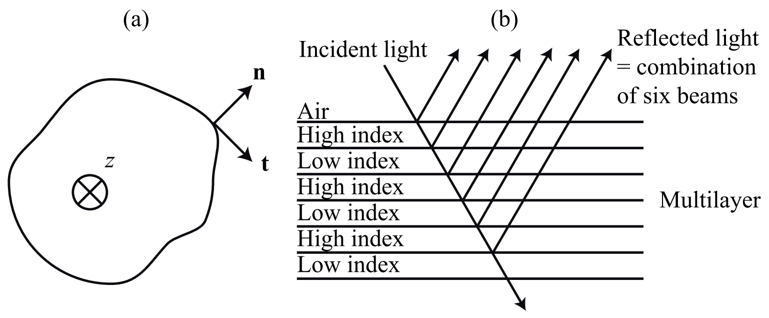

Fig. 1 (a) Long conductor with uniform cross section. (b) Scheme of a high-reflectance coating.

2.1 FEM analysis of Laplace's equation

An infinite cylinder perturbed by a small cosine function can be described as

$$r(\theta) = a\left(1+\beta\cos(q\theta)\right), \tag{1}$$

where a is the radius of the cylinder, q is an integer parameter and β is a small number. The analytical expression that relates the curvature of an infinite conductor and the electrostatic potential is

$$\phi(r,\theta) = \phi_0 + b_0 \ln\frac{r}{a} - \beta b_0 \left(\frac{a}{r}\right)^q \cos(q\theta), \tag{2}$$

where r and θ are the cylindrical coordinates, ϕ_0 is the potential at the surface of the conductor and b_0 is a constant coefficient defined in [5]. Considering the Laplace's equation $\nabla^2\phi = 0$, the region in which the potential distribution $\phi(x,y)$ is defined, must be divided into a number of finite elements. Specifically, in this work, the solution region is subdivided into triangles (quadrilateral elements divided across the shorter diagonal) [8,9]. With this grid, the approximate solution can be computed as a combination of each mesh element solution. After some mathematical manipulation [10,11], the solution can be represented as a sparse system of equations, solved using the Biconjugate Gradient Method (BGM) [12,13]. This algorithm is an iterative approach that mainly uses matrix-vector multiplications and inner products. These operators are critical and, therefore, the objective of our optimization.

2.2 FDTD analysis at optical wavelengths

Light propagation is described by means of Maxwell's time-dependent curl equations [14]. To solve these equations, the FDTD algorithm uses the Yee lattice [15]. The electrical field components **E** and the magnetic field component **H** are defined in a bidimensional cell [3,4,15]. As a result, the Maxwell's curl equations can be discretized and solved using the central-difference expressions, for both time and space derivatives. So, considering TE polarization and a 2D analysis, the flux density en z direction can be expressed as follows:

$$\widetilde{D}_z\big|_{i,j}^{n+1/2} = \widetilde{D}_z\big|_{i,j}^{n-1/2} + \frac{\Delta t}{\sqrt{\varepsilon_0 \mu_0}} \left[\frac{H_y\big|_{i+1/2,j}^{n} - H_y\big|_{i-1/2,j}^{n}}{\Delta x} - \frac{H_x\big|_{i,j+1/2}^{n} - H_x\big|_{i,j-1/2}^{n}}{\Delta y} \right] \tag{3}$$

where Δx, Δy and Δt are the spatial resolution in x and y directions and the time resolution respectively. Regarding the add-ons needed, a simplified version of the Perfectly Matched Layers (PML) developed by Berenger [16–18] has been implemented to simulate unbounded free space. Moreover, related with the illumination method, the incidence is assumed to be from air to medium. In connection with the propagation in the FDTD region, the source is introduced along the connecting boundary by using a Total Field/Scattered Field (TF/SF) technique [3], where the linearity of Maxwells's equations and their decomposition on incident and scattered fields are assumed. This method avoids the computation of the incident wave in the whole bidimensional grid and only two one-dimensional arrays are needed.

3 Computational optimization

In this section the approach followed to implement both numerical methods as a unified C++ library is shown. The implementation and the application for GPU computing is also detailed. The library is implemented in C++, from which only classes and overloading were considered, since advanced inheritance directives are not already allowed in CUDA [19, 20] and in some cases can reduce the performance. The library contains an `Array` class that is common to both methods. This class implements a one-dimensional array that can store a matrix of size $n \times m \times p$ in column major order with the advisable padding, providing an easier physical memory alignment. This class also implements several arithmetic operations such as matrix vector product, inner product, dot product, etc. These operations are implemented taking into account the spatial proximity of the data in order to avoid cache misses and improve both, sequential and parallel versions. Therefore, all these methods can be used in both FEM and FDTD. However, a `Sparse Array` class is used in FEM in order to store efficiently the sparse matrices obtained in this method. The structure for storing sparse matrices is detailed in Section 3.1. Taking into account that the way of storing a matrix for FEM is quite different than for FDTD, it is necessary a new class which is focused on solving the system of equations. This class is known as `Sparse Solver` and implements the sequential and CUDA versions of the BGM algorithm.

Regarding FDTD, the `FDTD_TE` class is in charge of updating the field components. These operations are conditioned by the add-ons included in the current implementation (PML and TF/SF formulations). The computation in parallel architectures instead of CPU is chosen by the user in class invocation by means of the constructor of both classes: `SIM_FEM_Poisson_2d` and `FDTD_TE`.

All the results presented here use single precision data, more than enough accuracy for the proposed problems. However, the library use templates allowing double precision if needed. The GPU architecture under study is the NVidia Fermi, it is used with single precision only but it allows double precision also. Specifically, the GPU is a GTX-470 GPU with a theoretical upper limit of 1088.64 GFLOPS. All the test are done under a Unix based platform with an Intel i7-950 Processor with 8MB of cache, a clock speed of 3.06 GHz and 6 GB of global DDRAM3.

3.1 Instruction rearrangement for auto-vectorization in CPU

The auto-vectorization provided by modern compilers (flag `-O3` in gcc) is based on predicting which loops can be automatically vectorized, or converted into SSE instructions [2]. Although a theoretical 4x performance gain is expected (single precision), usually only 2x is achieved due to I/O latency and other issues [21].

The auto-vectorization is sensitive to the layout of the loop, the data structures in use, and the dependencies among the data accesses in each iteration and across iterations. Once the compiler has made such a determination, it can generate vectorized code for the loop. Therefore, the following strategies have been used in order to take advantage of this capability: memory alignment, use of arrays to make data contiguous and padding for avoiding misalignment.

One of the most common operators in FEM is the matrix vector multiplication $\mathbf{A} \cdot \mathbf{b} = \mathbf{c}$, so it must be optimized to improve the performance. For that purpose the matrix A can be redefined as $\mathbf{A} = \left[\mathbf{a}^1 \ldots \mathbf{a}^n\right]$, where $\mathbf{a}^i$ is the i-th column of the matrix $\mathbf{A}$. Therefore, the matrix vector multiplication can be rearranged as $\mathbf{c} = b_1\mathbf{a}^1 + \cdots + b_n\mathbf{a}^n$. This rearrangement improves the access to contiguous data, reducing the cache misses. The nonzero values of $\mathbf{A}$ are accessed sequentially in column order and multiplied by a single value of $\mathbf{b}$ indexed by the column of $\mathbf{A}$. With this configuration, the values of both $\mathbf{A}$ and $\mathbf{b}$ are accessed sequentially and only once in each iteration of the BGM, instead of the classical way of performing the operation of $\mathbf{A} \cdot \mathbf{b}$.

Due to the mesh used in the discretization, the maximum number of nonzero elements per rows is 7, so the global coefficient matrix is sparse (usually more than 95% sparse). Therefore, the Compressed Sparse Column (CSC) format was used for storing the values of the nonzero matrix by columns [13]. This scheme stores all nonzero values in a single column vector and uses a couple of pointers for indexing the values. The number of nonzero elements per column is padded to a multiple of 4, reducing the misalignment in memory accesses.

The same optimization techniques used for FEM can also be used for FDTD. Although, in this case to solve the FDTD equations the *leapfrog* algorithm is used [4]. To minimize cache misses Eq. (3) must be evaluated by columns, assuring the spatial proximity of the data. Also, the width of the simulation must be a multiple of four, since we are using single precision data. This can be accomplished by padding in the matrix or by making the PML boundaries wider. The same procedure must be done with the rest of field updates.

Fig. 2 shows the two main loops used by FEM and FDTD that allow the auto-vectorization provided by the compiler. Fig. 2a implements the inner product needed in FEM whereas Fig. 2b shows the implementation of Eq. (3)[1]. To force the GCC

[1] Note that the expressions here detailed do not consider the PML and the TF/SF formulations in order to simplify the notation. The specific equations related with this add-ons are detailed in [22]

(a)

```
        for (int i = 0; i < n_cells_; i++, a++, b++)
           aux += *a * (*b);
        *d = aux;}
```

(b)

```
      for(i = 1; i<rows_; i++){
       *pdz = *pdz + c_*(*phy-*phy_rm-*phx+*phx_cm);}}
```

Fig. 2 (a) Sequential code for the inner product method used in FEM. (b) Sequential code for the D_z update in FDTD.

compiler to optimize the code, the flags `-O3 -ffast-math` must be used, this way the compiler replaces automatically scalar operations by SSE instructions whenever possible [23]. Using the flag `-ftree-vectorizer-verbose = 2`, the compiler will show information about which loops are being vectorized, in our case both loops.

3.2 GPU implementation

The computation in a GPU is based on the division of the task in a myriad of small computations which ideally are all performed at the same time, each one on its own thread. The basic computing unit (a warp) consists of 32 threads, and the GPU has the advantage of swap warps into and out of context without any performance overhead. To achieve a good performance, it is mandatory to hide the high latency of the memory ensuring coalesced accesses.

The knowledge of the GPU architecture is mandatory to be successful in GPU computing. In our case, for FEM the instruction rearrangement already mentioned in Section 3.1 is used, and the operator matrix vector multiplication has been implemented by means the CUSPARSE Library [24]. The inner product needed for the product of two vectors is completed by means of the reduction technique developed in [19]. The general idea is that each thread will add two of the values obtained from the dot-multiplication of two vectors. Since each thread combines two entries into one, this step is completed with half of the entries. In the next step, the same process is repeated on the remaining half. Regarding the FDTD implementation in the

```
  for (int i = 0; i < tile; i++, Row += blockDim.x, index+= blockDim.x){
      if (index < n_cells){
        if (Row > 0 && Row < rows){
         curl_hy = d_TMz. Hy[index] - d_TMz. Hy[index-1];
         curl_hx = d_TMz. Hx[index] - d_TMz. Hx[index - rows];
         dz = d_TMz. Dz[index] + c * (curl_hy  - curl_hx);
         d_TMz. Dz[index] = dz;
         d_TMz. Ez[index] = dz * d_TMz. Ga[index];}}}}
```

Fig. 3 Cuda Kernel implementation for updating the E_z field component.

GPU it must be said that, electromagnetic field update is performed by several kernels focused on solving one field component at a time. An example of the instructions performed by the kernel in charge of updating the electric flux density in z direction is shown in Fig. 3. These instructions are directly related with Eq. (3), where `Ga` is a matrix related with the physical properties of the media as a function of the space. A number of blocks related with the number of columns is invoked by means of the kernel functions and an array of 256×2 threads are launched per block. Each column of threads works along one column of the simulation grid as many times as necessary to update the electromagnetic field. This scheme is known as the optimization tilling technique detailed in [19].

Besides the potential of the CUDA kernel, it is necessary to divide the whole computation process in several kernels focused on compute each component of the electromagnetic field. This segmentation improves the efficient use of the shared memory in the device and also the correct usage of the cache. Due to the nature of the task, the coalesced accesses are assured and, moreover, there is no need to explicitly use shared memory since the GPU is able to use it automatically as a cache. We tried a shared memory scheme avoiding the auto-caching but the throughput achieved was almost identical.

4 Results

Our first group of results shown in Fig. 4a are the comparison of the electrostatic potential ϕ obtained by Eq. (2) and FEM. In Fig. 4b the charge density σ is shown and it has been obtained taking into account that $\sigma \approx |\nabla\phi|$. As can be seen a good agreement between numerical an theoretical values is achieved in both cases, thus validating our approach. It must be said that, the discretization of the problem, i.e. the mesh used, is not uniform, the area of the elements in the neighborhood of the conductor surface is reduced ensuring an accurate approach of the shape as well as a high precision in the region of interest. Regarding FDTD, the results of the analysis

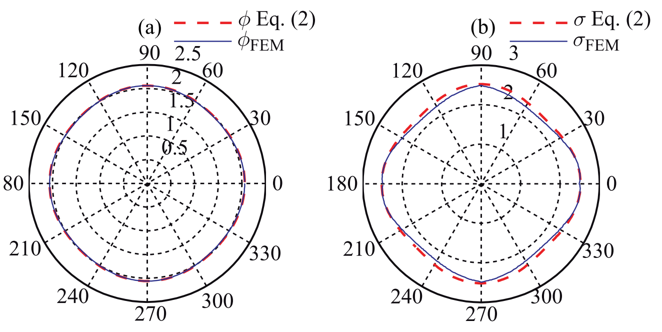

Fig. 4 (a) Analytical and numerical charge density for $\beta = 0.02$ and $q = 2$. (b) Analytical and numerical potential for $\beta = 0.02$ and $q = 4$.

of the reflectance ($\propto |E|^2$) for normal incidence of a set of low-pass stacks of thin film layers are given. Fig. 5 shows the results for two low-pass stacks defined by four repetitions of a fundamental period consisting of three layers: Air$[0.5HL0.5L]^4$Ga. L

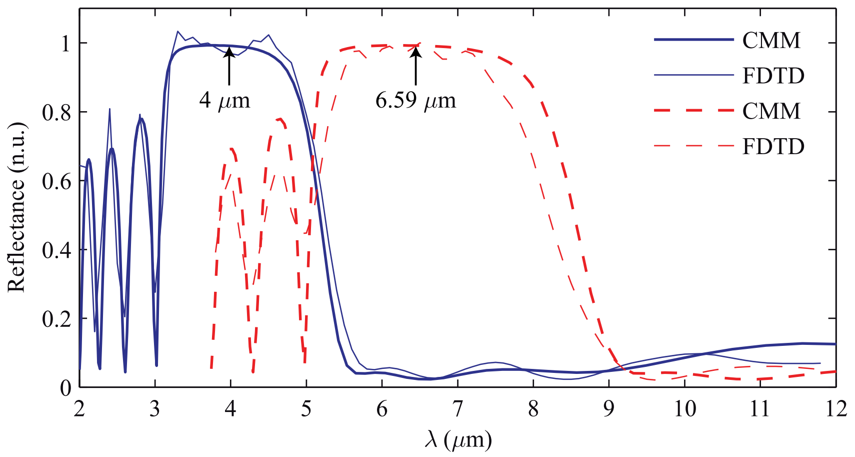

Fig. 5 Comparison between analytical and numerical values obtained by means of the CMM and FDTD respectively. Reflectances of two low-pass stacks Air$[0.5HL0.5H]^4$Ga with L and H being thin films of stibnite and chiolite one quarter-wave thick at reference wavelength $\lambda_0 = 4\mu$m or $\lambda_0 = 6.59\mu$m. FDTD parameters: $\lambda_{\min} = 2\mu$m, $\Delta = 0.02\mu$m, $\Delta t = 33.3 \cdot 10^{-9}$ ns.

and H are thin films of stibnite ($n_H = 2.7$) and chiolite ($n_L = 1.35$) one quarter-wave optical thick at reference wavelength $\lambda_0 = 4.36\mu$m or $\lambda_0 = 6.66\mu$m [6,25]. The substrate composed of Germanium has a refractive index of $n_s = 1.52$. The numerical reflectance obtained by means of FDTD is compared with the analytical reflectance defined by the Characteristic Matrix Method [6] that has been implemented in double precision. As can be seen in Fig. 5 the numerical values are close to the theoretical values given by the CMM and similar to those given in [6] and [25]. There are slight differences between numerical and analytical values produced by the spatial finite resolution of the FDTD method. Due to the fact that FDTD requires an spatial discretization of the media, the thickness of each stack is truncated as a function of the spatial resolution given by the method. Regarding the computational efficiency, several analysis are summarized in Fig. 6. In the case of GPU computing, the time needed to communicate host and GPU are included in the graphs. The relative speed up shown is defined as the ratio between the simulation times of the sequential (without auto-vectorization) and parallel versions (sequential version with auto-vectorization `-O3` and the GPU parallel version) and gives an idea of how faster is a parallel version compared with the sequential program in terms of time simulation.

Fig. 6a shows the computational performance of FEM by means of the relative speed up and the FLOPS as a function of the number of mesh elements. As can be seen, there is a region in which a single CPU with an auto-vectorized code is a better option. Nevertheless, the time reductions achieved with the GPU version are quite impressive and the trend of the relative speed up allows to affirm that GPU computing is mandatory for massive computations in FEM. Fig. 6b shows similar results regarding FDTD. In this case, the benefits obtained with GPU computing are greater and also more homogeneous as a function of the grid size. This is because FDTD is more suitable to be implemented in parallel architectures. FEM implies the use of

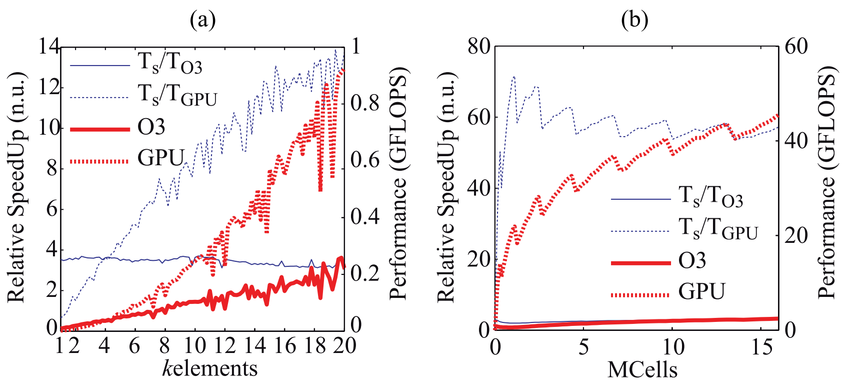

Fig. 6 Relative SpeedUp (thin lines) and GFLOPS (thick lines) as a function of the number of cells. (a) FEM: Sequential vs CPU auto-vectorized and GPU codes. Relative SpeedUp and GFLOPS as a function of the number of mesh elements. (b) FDTD: Sequential vs CPU auto-vectorized and GPU codes (square grids).

sparse matrices which inner structure depends on the problem. This aspect in some cases makes difficult to translate this process to highly parallel architectures [26, 27]. Whereas, in FDTD the update of the electromagnetic field mainly depends on the previous values and not in the geometry of the problem, thus making the implementation of each CUDA kernel easier. However, in FDTD the upper limit in terms of FLOPS is not achieved since it requires a high rate transfer between global memory and GPU processor. For example, to update D_z in Eq. (3) five memory accesses are needed. In this situation the performance is limited by the memory bandwidth instead of the computing power of the GPU. In [28] a deeper analysis of the operational intensity of FDTD is provided. However, the bandwidth achieved in the implementation was near the 90% of the maximum provided by the manufacturer.

5 Conclusions

We have implemented an unified library for electromagnetic analysis based on FEM and FDTD. FEM was used for comparing the analytical expressions obtained for the analysis of the surface curvature of an infinite cylinder in Electrostatics, whereas FDTD has been applied at optical wavelengths for analyzing the reflectance of high-reflecting coatings. In both cases, the analytical and numerical results are quite similar, thus validating our implementation.

For each numerical method we have implemented two different versions, one for the CPU and one for the GPU. The CPU versions have been designed to exploit all the automatic optimization facilities provided by modern compilers. This way, the optimized version achieves a speedup of near four in both FEM and FDTD cases. The GPU versions achieved quite different results depending on the method implemented. For FEM, the speed up increases with the number of elements, while for FDTD, the speed up remains more constant and in all cases is higher than the CPU auto-vectorized version. The authors are currently working on using implicit compiler intrinsics on the code for directly controlling the vectorization process in both FEM and FDTD. Moreover, shared memory schemes are being considered in order to achieve the best performance of multi-cores available in modern CPU. For FEM, the next step is to try alternative sparse matrix storage strategies, like the ones proposed in [26,27].

Acknowledgements This work was supported by the “Ministerio de Economa y Competitividad” of Spain under projects FIS2011-29803-C02-01 and FIS2011-29803-C02-02, by the “Generalitat Valenciana” of Spain under project PROMETEO/2011/021. The authors appreciate the useful comments from anonymous reviewers that helped to improve this manuscript.

References

1. N. Corporation, *Whitepaper NVIDA’s Next Generation CUDA Compute Architecture*, 1st edn. (2009)
2. I. Corporation, *Intel 64 and IA-32 Architectures: Optimization Reference Manual* (2011)
3. D.M. Sullivan, *Electromagnetic Simulation using the FDTD Method* (IEEE Press Editorial Board, 2000)
4. A. Taflove, *Computational Electrodynamics: The Finite-Difference Time-Domain Mehtod* (Artech House Publishers, 1995)
5. C. Neipp, J.C. Moreno, J.J. Rodes, J. Francés, M. Pérez-Molina, S. Gallego, A. Beléndez, J. of Electromagn. Waves and Appl. **24** (2010)
6. H.A. Macleod, *Thin-Film Optical Filters* (Institute of Physics Publishing, 2001)
7. M.A. Helmbrecht, in *Procedings of Lasers and Electro-Optics (CLEO) and Quantum Electronics and Laser Science Conference Intel Tecnology Journal*, vol. 1 (2010), vol. 1
8. M.N.O. Sadiku, *Numerical Techniques in Electromagnetics* (CRC Press, New York, 2001)
9. I.D. Faux, M.J. Pratt, *Computational Geometry for Design and Manufacture* (Ellis Horwood Publishers, Chichester, 1981)
10. J. Francés, S. Bleda, S. Gallego, C. Neipp, A. Márquez, I. Pascual, A. Beléndez, in *Proceedings of the 2011 International CMMSE*, vol. II, ed. by J. Vigo-Aguiar (2011), vol. II, pp. 520–531
11. J. Jin, *The Finite Element Method in Electromagnetics* (Wiley-Interscience, New York, 2002)
12. W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, *Numerical Recipes: The Art of Scientific Computing* (Cambridge University Press, 2007)
13. J.L. Volakis, L.C. Chatterjee, L.C. Kempel, *Finite Element Method for Electromagnetics* (Wiley-Interscience, IEEE Press, 1998)
14. A.A. Balanis, *Advanced Engineering Electromagnetics* (Wiley-Interscience, New York, 1991)
15. K.S. Yee, IEEE Trans. on Antennas and Propagation **AP**(17), 585 (1966)
16. J.P. Berenger, J. Comput. Phys **114**, 185 (1994)
17. J.P. Berenger, J. Comput. Phys. **127**(2), 363 (1995)
18. D.M. Sullivan, Microwave and Guided Wave Letters, IEEE **6**(2), 97 (1996)
19. J. Sanders, E. Kandrot, *CUDA by Example: An introduction to general-purpose GPU programming* (Addison-Wesley, Upper Saddle River, NJ, 2011)
20. N. Corporation, *NVIDA CUDA C Programming Guide*, version 3.2 edn. (2010)
21. S. Thakkar, Intel Technology Journal **Q2**, 1 (1999)
22. J. Francés, C. Neipp, M. Pérez-Molina, A. Beléndez, Computer Physics Communications **181**(12), 1963 (2010)
23. M.T.F. Cunha, J.C.F. Telles, F.L.B. Ribeiro, Advances in Engineering Software **39**(11), 888 (2008)
24. N. Corporation, *CUDA: CUSPARSE Library* (2010)
25. A.F. Turner, P.W. Baumeister, Applied Optics **5**(1), 69 (1966)
26. G. Ortega, E.M. Garzón, F. Vázquez, I. García, in *Proceedings of the 2011 International CMMSE*, vol. III, ed. by J. Vigo-Aguiar (2011), vol. III, pp. 908–917
27. F. Vázquez, G. Ortega, J.J. Fernández, E.M. Garzón, in *Proceedings of the 10th IEEE International Conference on Computer and Information Technology* (2010), pp. 1146–1151
28. K.H. Kim, K. Kim, Q.H. Park, Computer Physics Communications **182**(6), 1201 (2011)