Hindawi Publishing Corporation
International Journal of Antennas and Propagation
Volume 2014, Article ID 721580, 13 pages
http://dx.doi.org/10.1155/2014/721580

Research Article

Application of the Characteristic Basis Function Method Using CUDA

Juan Ignacio Pérez,[1] Eliseo García,[1] José A. de Frutos,[1] and Felipe Cátedra[2]

[1] *Automatics Department, University of Alcala, 28871 Alcalá de Henares, Spain*
[2] *Computer Science Department, University of Alcala, 28871 Alcalá de Henares, Spain*

Correspondence should be addressed to Felipe Cátedra; felipe.catedra@uah.es

Received 24 September 2013; Accepted 9 February 2014; Published 3 April 2014

Academic Editor: Ahmed A. Kishk

The characteristic basis function method (CBFM) is a popular technique for efficiently solving the method of moments (MoM) matrix equations. In this work, we address the adaptation of this method to a relatively new computing infrastructure provided by NVIDIA, the Compute Unified Device Architecture (CUDA), and take into account some of the limitations which appear when the geometry under analysis becomes too big to fit into the Graphics Processing Unit's (GPU's) memory.

1. Introduction

The MoM [1] is a popular method for performing electromagnetic analyses of arbitrary 3D geometries. It works by converting a set of integral-differential equations into a system of linear equations, in which the unknowns are the coefficients of the induced currents with respect to a set of basis functions. This conversion entails a discretization process which, in order to provide accurate results, must divide the geometry into subpatches of usually around one tenth of the wavelength of the excitation in size. As the size of the geometry grows, the problem becomes computationally large very quickly. That is the reason for the appearance of several techniques whose goal it is to make the problem tractable for bigger geometries. One of the most interesting is the characteristic basis function method [2] in combination with the fast multipole method [3] and the multilevel spproach [4]. The CBFM consists of expressing the currents in terms of high-level macrofunctions, which reduce the size of the linear system.

With the proliferation of relatively nonexpensive high performance computers, some effort has been put into crafting parallel versions of these numerical methods [5–8]. Our work focuses on accelerating the CBFM by means of a relatively new programming paradigm called CUDA (Compute Unified Device Architecture). The main objective of CUDA is to allow the use of graphics processors for high performance scientific computing. The performance gains using CUDA are quite remarkable. Recently, it has been used successfully for tasks like the calculation of the impedance matrix in the MoM or for the solution of the linear system which the MoM yields. One example of the first case is given in [9]. The reported speedup is up to 70, when compared with the same operation performed on the CPU. In [7], they obtain speedup for the same task up to 140. Then they perform an LU decomposition of the matrix, but this does not benefit so much from the GPU, and the total speedup reported for the whole MoM execution is about 45. The LU decomposition of the system matrix is specifically addressed in [10]. The reported speedup, using CUDA, is of up to 20 with 3 GPUs. This work would fit in a category where it is shown how to further optimize the process or to deal with some limitations. In order to be able to tackle arbitrarily big geometries, they divide the matrix in blocks and store it in hard disks, not in system RAM memory. Similarly, to overcome the relatively low amount of memory available in off-the-shelf graphics cards, [11] proposes a scheme in which the impedance matrix is divided into submatrices and one submatrix is calculated

at a time. The overall gain is, in this case, a factor of 30 for the calculation of the impedance matrix, even though by partitioning it, some penalty is naturally incurred. This work is even more remarkable because the authors did not have at their disposal the tools later provided by NVIDIA for CUDA. Reference [12] also addresses this limitation. By the nature of the problem, in the straightforward implementation of the MoM, some calculations are performed multiple times. In [13], Kiss et al. try to avoid unnecessary calculations by computing each value once and then reusing it when applicable. A good example of the fine-tuning which is sometimes necessary when using CUDA is [14], in this case applied to wire-grid models. They analyze the resources of the graphics card and show how to partition the computing tasks among its computing cores so as to achieve the maximum possible gain.

In this work we use all the expertise provided by these papers and gained on our own to accelerate the CBFM with CUDA. Since the CBFM bases on the MoM, we must first calculate the impedance matrix for the geometry. Most of the works above use a characterization of the geometry based on triangulation. We use square NURBS patches because, in our experience, these kinds of patches can fit the geometry better, requiring the use of fewer surfaces [15]. When calculating the elements of the impedance matrix, we use a variable number of integration points, depending on the distance between the two subdomains in the geometry whose coupling is each being calculated. When they are close, a high number of integration points is advisable in order to provide accuracy. When they are farther apart, we can lower this number and still obtain good accuracy. We split the geometry into blocks in order to overcome memory limitations. This is similar to the approach in [11], but, in the case of the CBFM, care must be taken to guarantee electrical continuity from block to block. The practical consequence is that, when calculating the impedance matrix for each block and its high-level macrofunctions, an extension of the said block must be processed instead of the original one. At the end of the process, we obtain a reduced characteristic-basis-function-based-matrix for the complete geometry.

The rest of this paper is organized as follows. In Section 2 we review the MoM and the CBFM and describe the process whereby the geometry is divided into blocks, and we describe the consequences for the algorithm as well. The implementation of the algorithm in CUDA is described in Section 3, which is followed by the experimental results in Section 4. We dedicate Section 5 to the conclusions.

2. MoM and CBFM

2.1. Method of Moments. The MoM [1] is used to convert a set of integral-differential equations into a set of linear equations. When applied to the electromagnetic analysis of a geometry, the final result is a linear system of equations (1), where Z is known as the *coupling matrix*, V are the *impressed voltage vectors*, which represent the excitations, and J are the induced current coefficients to be found. Consider

$$V = ZJ. \tag{1}$$

Using rooftop and razor-blade functions as basis and testing functions, respectively [15], the term z_{ij} of the coupling matrix can be expressed as

$$z_{ij} = z_{ij}^{\text{ind}} + z_{ij}^{\text{cap}}, \tag{2}$$

where

$$z_{ij}^{\text{ind}} = \int_{r_a}^{r_b} \left[\frac{j\omega\mu_0}{4\pi} \int_{S_{j1}} G\left(r, r'\right) J_j^{S_{j1}}\left(r'\right) dS' + \frac{j\omega\mu_0}{4\pi} \int_{S_{j2}} G\left(r, r'\right) J_j^{S_{j2}}\left(r'\right) dS' \right] dl, \tag{3}$$

$$z_{ij}^{\text{cap}} = P^{S_{j1}}\left(r_b\right) - P^{S_{j1}}\left(r_a\right) - \left[P^{S_{j2}}\left(r_b\right) - P^{S_{j2}}\left(r_a\right)\right]. \tag{4a}$$

Terms like $P^{S_{j1}}(r_b)$ can be obtained from

$$P^{S_{j1}}\left(r_b\right) = \frac{-1}{j4\pi\omega\varepsilon_0} \int_{S_{j1}} \frac{G\left(r_b, r'\right)}{S_{j1}} dS'. \tag{4b}$$

Equations (3) and (4b) include terms like the Green's function $G(r, r')$, defined as

$$G\left(\bar{r}, \bar{r}'\right) = \frac{e^{-jk\left|\bar{r}-\bar{r}'\right|}}{\left|\bar{r}-\bar{r}'\right|}. \tag{5}$$

They also include terms like $J_j^{S_{j1}}$, which is the density of current of subdomain j in patch S_{j1}, r', which is the position vector for points in the patches of subdomain j, r, which is the position vector for points in subdomain i, and r_a and r_b, which are the end-points of the razor-blade for subdomain i. A more extensive and detailed mathematical description of the process which arrives at these equations can be found in [15].

Computationally, the integrals for (3) and (4b) are calculated using Gaussian quadratures [16]. The integral is approximated by a series in which we select appropriately weights and abscissas:

$$\int_a^b W(x) f(x) dx \approx \sum_{j=1}^{N} w_i f\left(x_i\right), \tag{6}$$

where w_i are the weights chosen, f is a polynomial, and $W(x)$ is a function chosen to remove integrable singularities from the integral. All integrals are treated similarly. In the case of the integrals between the brackets of (3), the transformation is

$$\int_{S_{j1}} G\left(r, r'\right) J_j^{S_{j1}}\left(r'\right) dS' \approx \sum_{i=1}^{N}\sum_{l=1}^{M} w_i w_l G\left(u_i, v_l\right) J_j^i, \tag{7}$$

where $G(u_i, v_l)$ is the evaluation of the Green's function at the point with parametric coordinates (u_i, v_l) in the jth subdomain. In this case, we choose $W(x) = 1$, so that $f(x)$ is the product of G and J. N is the number of abscissas chosen and depends on the precision desired. In the results section, we first analyse the case where this number is fixed at 10 points and then adapt the number of points depending on the distance between the subdomains involved in the calculation. For this, we use values of N of 2, 4, and 10.

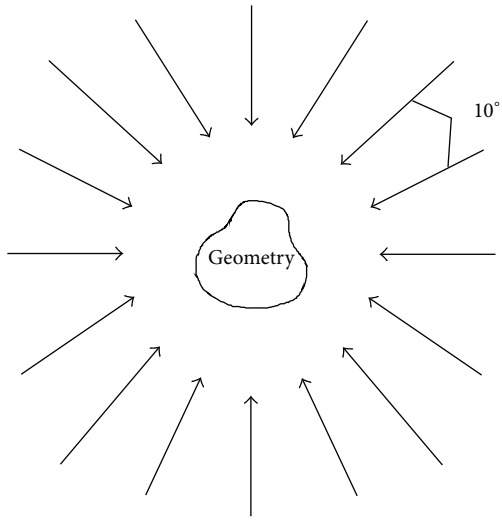


Figure 1: Generation of excitations.

2.2. Characteristic Basis Function Method. Once matrix Z has been calculated, it can be used to solve the electromagnetic problem. However, Z becomes very large quickly as the size of the geometry increases. Directly solving the linear system in (1) can be very time- and memory-consuming.

In order to reduce this burden, some techniques can be used. One of them is the characteristic basis function method (CBFM) [2]. This method introduces the use of high-level basis macrofunctions. These functions are defined over relatively large domains called blocks. The CBFs can be expressed in terms of conventional subdomain functions (in this case, rooftops). The CBFM can reduce the size of matrix Z considerably, which allows us to perform analyses on the geometry much faster.

The method works as follows: once the coupling matrix Z is calculated, we generate the Plane Wave Spectrum (PWS). The PWS is a set of plane waves which impinge on the geometry of each block from different angles. In this work, the directions of incidence are evenly spaced at 10° intervals, both with respect to their polar angle θ and to their azimuthal angle ϕ. Then, the total number of plane waves (excitations) is 684. Figure 1 depicts the excitations contained in one plane with constant ϕ.

The currents induced by the excitations can be calculated using the MoM by solving the system of (1) for each block or can be approximated by Physical Optics. In this work we choose the MoM option. We obtain, for each excitation, an induced current J, which is expressed as a column vector; each component corresponds to the amplitude of a subdomain. Then, the vectors that define J for all excitations are orthogonalized. For that we use the Singular Value Decomposition (SVD). The resulting orthogonal vectors are the Characteristic Basis Functions (CBFs) of this block. Each CBF has an associated Singular Value. A threshold is applied to the CBFs based on their singular values, which allows us to only retain a small number of them. This threshold can be selected such that the problem is greatly reduced in size, while precision remains good. The process is described with more detail in [17].

The problem is thus transformed from subdomain representation to CBF representation. The new coupling matrix, Z', is much smaller than the old one and has the general form:

$$Z' = \begin{bmatrix} \langle L(J_1), W_1\rangle & \langle L(J_1), W_2\rangle & \cdots & \langle L(J_1), W_m\rangle \\ \langle L(J_2), W_1\rangle & \langle L(J_2), W_2\rangle & \cdots & \langle L(J_2), W_m\rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle L(J_m), W_1\rangle & \langle L(J_m), W_2\rangle & \cdots & \langle L(J_m), W_m\rangle \end{bmatrix}, \tag{8}$$

where $\langle L(J_i), W_k\rangle$ represents the inner product of the jth CBF with the kth high-level testing function. In terms of subdomain functions, the CBFs can be expressed as

$$\begin{aligned} J_i(u,v) &= \sum_{l=1}^{n} \alpha_i(n)\, T_l(u,v) \\ W_i(u,v) &= \sum_{l=1}^{n} \alpha_i(n)\, R_l(u,v), \end{aligned} \tag{9}$$

where $T_n(u,v)$ and $R_n(u,v)$ are the nth rooftop and razor-blade functions, respectively. The elements of reduced matrix Z' can then be computed as

$$\left\langle L(J_i), W_j\right\rangle = \sum_{k=1}^{m}\sum_{l=1}^{n} \alpha_i \alpha_j^{*} \left\langle T_l(u,v), R_k\left(u', v'\right)\right\rangle. \tag{10}$$

If the inner product is substituted by the low-level impedance matrix coefficient $z_{k,l}$, then the result is

$$\left\langle L(J_i), W_j\right\rangle = \sum_{k=1}^{m}\sum_{l=1}^{n} \alpha_i \alpha_j^{*} z_{k,l}. \tag{11}$$

2.3. Decomposition into Blocks. The CBF method greatly reduces the size of the problem, but it is still necessary to calculate the complete coupling matrix Z. As has been explained above, this matrix grows rapidly with the original problem size, and it is soon too big to be handled computationally. One of the advantages of the CBFM is that it opens the possibility of dividing the geometry into blocks. This is what we do when it becomes too large to be treated in one block: we divide it into several of them and handle them separately. When dividing the geometry, we must take into account the electrical continuity from block to block. For that, for each block in the geometry, a corresponding block is introduced, called the *extended* block, which contains all subdomains in the standard block, plus all subdomains which are contiguous to the subdomains in it, up to a certain predefined distance. The geometry is, thus, processed as extended block by extended block, instead of in one piece. The CBF method, as described in the previous section, is applied to each extended block separately, which is computationally feasible. The use of extended blocks allows the calculation of the induced currents without loss of precision. The concept of extended block is shown graphically in Figure 2.

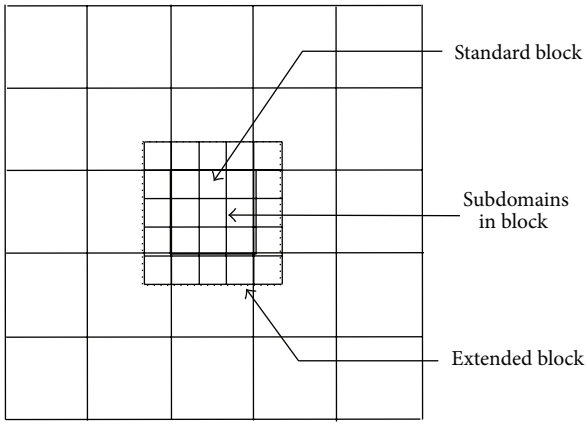


Figure 2: Blocks and subdomains.

The extended blocks are only used to calculate the induced currents. Once they are calculated, the electrical continuity has been guaranteed, and from then on, only those currents which are inside the original blocks are retained and orthogonalized.

The effect which this division of the geometry into blocks has on the coupling matrices (both regular and reduced) is that they are also, in turn, divided into submatrices. Equation (12) shows this effect for the reduced matrix, but the effect is analogous for matrix Z. Each submatrix in Z' reflects the coupling of two blocks in the geometry: the active block and the victim block. The elements of each submatrix are calculated in a manner analogous to that described above (see (8)), only taking into account that we work with blocks now.

The complete reduced matrix Z' is

$$Z' = \begin{bmatrix} [Z'_{1,1}] & [Z'_{1,2}] & \cdots & [Z'_{1,K}] \\ [Z'_{2,1}] & [Z'_{2,2}] & \cdots & [Z'_{2,K}] \\ \vdots & \vdots & \ddots & \vdots \\ [Z'_{K,1}] & [Z'_{K,2}] & \cdots & [Z'_{K,K}] \end{bmatrix}, \tag{12}$$

where

$$[Z'_{i,j}] = \begin{bmatrix} \langle L(J_{j,1}), W_1^i \rangle & \langle L(J_{j,1}), W_2^i \rangle & \cdots & \langle L(J_{j,1}), W_m^i \rangle \\ \langle L(J_{j,2}), W_1^i \rangle & \langle L(J_{j,2}), W_2^i \rangle & \cdots & \langle L(J_{j,2}), W_m^i \rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle L(J_{j,m}), W_1^i \rangle & \langle L(J_{j,m}), W_2^i \rangle & \cdots & \langle L(J_{j,m}), W_m^i \rangle \end{bmatrix} \tag{13}$$

$[Z'_{i,j}]$ is the reduced coupling matrix for blocks i and j and its elements are calculated according to the following equation:

$$\langle L(J_{i,n}), W_i^m \rangle = \sum_{k=1}^{M} \sum_{l=1}^{N} \alpha_{i,n} \alpha_{j,m}^* z_{k,l}. \tag{14}$$

Equation (14) shows that the elements of the reduced impedance matrix associated with the characteristic basis functions can be generated from the elements of the conventional MoM matrix.

2.4. Matrix Formulation. In order to make the computational work easier, it is convenient to express the relationship between induced currents and CBFs and between Z and Z' in terms of matrix algebra. Thus, if J represents the currents induced on the subdomains of the geometry, where each column vector is the current induced by one excitation, we can express the problem as (1). The SVD yields

$$J = U * S * V^T, \tag{15}$$

where U and V^T are orthonormal matrices and S is the diagonal matrix which contains the singular values.

When working with blocks, once the currents for the subdomains not included in the original block have been removed, the induced currents J_i can be decomposed, similarly, as

$$J_i = U_i * S_i * V_i^T. \tag{16}$$

The reduced matrix for this block can be obtained, similarly, by expressing (14) in matrix form:

$$Z'_{ii} = U_i^* Z_{ii} U_i, \tag{17}$$

where Z_{ii} is the matrix which represents the coupling of block i with itself and Z'_{ii} analogously in reduced form. Matrices Z_{ii} and Z'_{ii} represent the submatrices in the diagonal of Z and Z', respectively. The transformation for submatrices not in the diagonal is similar:

$$Z'_{ij} = U_i^* Z_{ij} U_j. \tag{18}$$

In this case, in order to calculate the reduced submatrix Z_{ij}, it is necessary to use the transformation matrices U_i and U_j, which correspond to blocks i and j, respectively. This suggests an order when obtaining the submatrices of Z'. First, those submatrices in the diagonal must be processed, since this yields the transformation matrices U_i, which are later used in the submatrices of Z' which are not on the diagonal.

The goal of the algorithm is to generate the reduced matrix Z', since with it, it is possible to analyse the effect of an arbitrary excitation on the geometry. To summarize everything discussed so far, the process to calculate the currents induced on the geometry by an arbitrary excitation is as follows.

(a) Divide the geometry into blocks.

(b) Generate a set of excitations (the PWS) which impinge on the geometry at evenly spaced intervals.

(c) Calculate the coupling matrix, in terms of subdomains, for every possible pair of blocks.

(d) Calculate the currents which the PSW induces on each block.

(e) Calculate the SVD for the set of currents induced on each block.

(f) Calculate the reduced coupling matrix for each pair of blocks.

(g) If necessary, assemble the reduced matrix Z' for all the geometry from the set of reduced coupling matrices calculated for each pair of blocks.

(h) To analyse the effect of an arbitrary excitation on the geometry, express the voltage V impressed by the excitation in terms of CBFs (V').

(i) Solve the linear system: $V' = Z'J'$, where V' is the voltage calculated in the previous step, and obtain J', the current induced by this voltage.

(j) Express J' in terms of subdomains, reversing the transformation from step (h).

It is important to note that steps (a) through (g) need only be performed once for each geometry. Once the reduced matrix Z' has been calculated, it can be used to analyse the effect of any excitation.

3. Implementation

3.1. CUDA. The method described above can greatly reduce the amount of time and memory required to solve problems using the MoM. However, it is still necessary to calculate the coupling submatrices for the blocks in the geometry, to solve the linear equations which provide the induced currents in the block, and to apply the SVD to obtain the transformation matrix for the block. These tasks still take up a considerable amount of time and memory, and to reduce both, some parallel computation methods have been put to use.

In this paper, we present one of such methods, which uses a recent innovation in high performance computing: CUDA (Compute Unified Driver Architecture) [18]. CUDA is a recent technology which makes use of GPUs (Graphics Processing Units). GPUs have been traditionally used to process graphics on a computer. With CUDA, they can also be used to perform scientific calculations. GPUs are designed with high numbers of processing devices, which can be put to work in executing a highly parallel scientific code. CUDA often requires a significant redesign of the scientific applications, but it provides considerable speedups. The CUDA programming paradigm and system architecture are defined to a better extent in [19]. In this section we present a brief description of them. Most of the details we present correspond to the first and second generation of CUDA architectures, Tesla (GT codenames) and Fermi (GF codenames).

With CUDA, programmers are allowed access to the hardware resources of GPUs which basically are processing cores and memory. Cores are simple processing units. They are physically grouped into clusters called Streaming Multiprocessors (SMs). An SM has several processing cores, registers, and shared memory. A GPU typically has several SMs. The number of cores per SM depends on the architecture. First generation Tesla cards have 8 cores per SM, and second generation Fermi cards have 32. The most recent Kepler models (GK codenames) have up to 192 cores per SM.

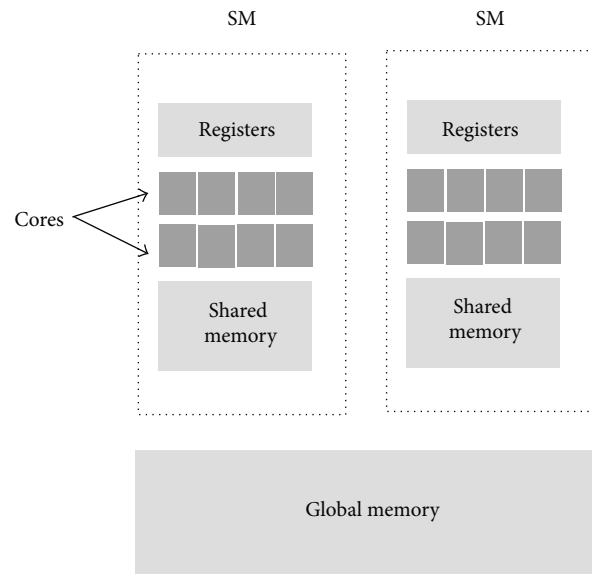


Figure 3: GPU architecture.

The GPU memory is organized in a hierarchy, which spans from relatively slow global memory, accessible to all the SMs, through fairly fast shared memories, accessible only by cores in the same SM, all the way to very fast registers, visible only by one core each. A simplified diagram of the architecture of a GPU is shown in Figure 3.

Given the parallel resources of the GPU, it is particularly indicated for applications which have a great amount of data parallelism, that is, those applications which repeatedly perform the same operations on big sets of data. In order to be able to use the resources made available by CUDA, the application must be designed with a special structure from a programming point of view. The application is organized as a succession of *kernel* calls. A *kernel* is a special kind of function or procedure, which runs on the GPU. It executes in Single Instruction, Multiple Data (SIMD) fashion, which means that several instances of the sequence of instructions that make up the function are executed in parallel. These instances are called *threads*. Each thread works on a different piece of the data and runs on one core. Since cores work in parallel, the overall effect is that several portions of the data are being processed at the same time.

The threads in a kernel are grouped into logical units called thread-blocks for easier handling inside the GPU. One thread-block is completely (meaning all its threads) assigned to one SM in the GPU. The number of threads in a thread-block is specified by the programmer, but there is a limit. This limit is 512 for older cards and 1024 for more recent ones. Since this number of threads may not be enough for regular applications, a kernel consists of several thread-blocks. Thread-blocks are arranged in a 2D logical grid, and a kernel can have up to 2^{32} thread-blocks, roughly over 4 billion blocks, or 2 trillion threads (for older generation cards).

At the time of execution, the thread-blocks in a kernel are dynamically assigned to the SMs for execution. The number

of thread-blocks assigned to every single SM depends on the hardware requirements of the thread-blocks (in terms of registers, shared memory, and number of threads). Typical numbers are 1 to 3 thread-blocks at any one time. When all SMs are full, the remaining thread-blocks must wait until they can be assigned. One of the benefits of handling the threads in this way is that the application can run on a wide variety of GPUs, from those with only 1 or 2 SMs to others with many more, without the need to be modified.

With all this in mind, the design of the application consists of several steps. First, it is necessary to decide which parts of it will be executed on the GPU and which ones will run on the CPU. Typically, those parts which exhibit a fair amount of data parallelism will be executed on the GPU and the rest will run on the CPU. The next step is to organize the parts assigned to the GPU into kernels. Sometimes it is possible to fit each part in one kernel and other times it is necessary to divide it into several, depending on the logical structure of the application. Lastly, the disposition of the data in the GPU memory must be decided, and the necessary data transfers must be included. Data need to be transferred into the GPU for use by the kernels, and results must be transferred out.

In the process of designing the GPU code, it is important to bear in mind that the code in one kernel is executed as is by all threads in it. There are thread and thread-block indices, so that each thread can identify itself (find out which one it is) and, based on that, decide which data it must work on. In principle, it is possible that, by using conditional branches based on those indices, different threads can do different things, but this is to be avoided whenever possible. The cost of a conditional branch which some threads take and others do not (i.e., a divergent branch) is a loss of efficiency. Whenever possible, all threads must execute exactly the same instructions.

3.2. Algorithm

3.2.1. General Overview. We have worked on a previous sequential version of the application [17]. We have made the necessary modifications in order for the desired part of the code to run on the GPU. Most of this code has been transformed into custom-made CUDA kernels. For some sections of the original application, however, we have used existing libraries. We have used cuBLAS [20], which is a version of BLAS (Basic Linear Algebra Subroutines) [21] adapted to CUDA, and also CULA [22], which is a version of a set of LAPACK (Linear Algebra Package) [23] routines also adapted to CUDA.

Our application has the general structure described in Algorithm 1. The first step is preparing all the data that kernels will use later and transfer it to the GPU memory. This step runs on the CPU and involves characterising the geometry, applying the mesh, defining the subdomains, calculating the matrix of excitations, and so forth. The result is a set of data structures which are necessary for navigating the geometry and for later processing.

After that, the first loop processes submatrices Z_{ii}, which are in the diagonal of the coupling matrix Z. Sub-matrices in the diagonal correspond to couplings of one block with itself:

$$\begin{bmatrix} [Z_{11}] & & & \\ & [Z_{22}] & & \\ & & \ddots & \\ & & & [Z_{nn}] \end{bmatrix}. \tag{19}$$

This corresponds to couplings of each block with itself. Processing is longer for these submatrices than for the rest (submatrices not in the diagonal).

After generating the coupling submatrix for the extended block, a process which will be described in more detail below, the currents induced by each excitation of the PWS in the subdomains of the extended block are calculated in the GPU by solving the linear system:

$$VE_k = ZE_{kk} * JE_k, \tag{20}$$

where VE_k is the excitation vector of the extended block (the generation of the excitations was explained in the previous section; see Figure 1), ZE_{kk} is the coupling submatrix for the extended block, and JE_k is the vector of induced currents in the extended block. For solving the linear system we use CULA routine `culaDeviceCgesv`, which runs on the GPU.

The matrix of induced currents is then trimmed in the GPU, whereby currents corresponding to subdomains which are in the extended block, but not in the standard block, are removed. At this point, boundary conditions have already been taken into account, and it is only necessary to process data for the standard block. This yields matrix J_k. This matrix is composed of column vectors, one for each induced current. Still in the GPU, it is orthogonalized with the SVD, which produces matrix U_k and a set of singular values. For calculating the SVD we use CULA routine `culaDeviceCgesvd`.

Matrix U_k is the transformation matrix between the subdomain space and the CBF space and consists of a set of orthogonal column vectors (the CBFs, expressed in terms of subdomains). The singular values are usually obtained in decreasing order of magnitude. A threshold is applied to them, discarding all singular values (and the corresponding columns of U_k) below the said threshold. In our case, we discard all values smaller than 1/500 times the greatest singular value. This process consists of inspecting all singular values in sequence until finding the first one which is smaller than 1/500 times the greatest one. This is a sequential procedure, which is more convenient to execute in the CPU. To this effect, we transfer the list of singular values to the CPU, where the desired singular value is found. Singular values up to this one (and the corresponding columns in matrix U_k) are retained and the rest are discarded. This latter operation is performed on the GPU, where matrix U_k has remained. Since the discarded columns are removed immediately, in order not to use too many names, in the following paragraphs we use the name U_k to signify the matrix obtained from the SVD minus the removed columns. This is the matrix we use to transform from the subdomain representation to the CBF representation, and vice versa.

```
;prepare data
⋮
;generate matrix blocks on the diagonal
for each geometry block:
     calculate extended coupling sub-matrix ZE_kk
     calculate extended induced currents JE_k
     obtain J_k by trimming JE_k(trim induced currents not on standard block k)
     apply SVD to J_k, obtain U_k and singular values
     apply threshold to orthogonal vectors of U_k
     trim extended coupling sub-matrix ZE_kk into standard coupling sub-matrix Z_kk
     calculate reduced sub-matrix Z'_kk
⋮
;generate matrix blocks not on the diagonal
for each coupling matrix block not on diagonal:
     calculate standard coupling sub-matrix Z_k1
     calculate reduced sub-matrix Z'_k1
```

Algorithm 1: Pseudocode for the application.

Matrix U_k can now be used to transform submatrix Z_{kk}, which contains those elements of submatrix ZE_{kk} which correspond to the standard block (in a process analogous to that performed on JE_k to get J_k and which is also executed in the GPU), from subdomains to CBFs, which gives submatrix Z'_{kk}. This submatrix is the kth element in the diagonal of Z'. The transformation is

$$Z'_{kk} = U_k^* * Z_{kk} * U_k, \tag{21}$$

where U_k^* is the transpose conjugate of U_k. For the matrix multiplications, we use cuBLAS routine `cublasCgemm`. This routine runs on the GPU. The resulting matrix Z'_{kk} is transferred from the GPU to the CPU memory and is stored there for later use. Matrix U_k, however, is left on the GPU, since it will be used in the second loop, which is explained below.

The second loop generates submatrices of Z' which are not in the diagonal. In these cases, the active block and the victim block are different, and it is only necessary to generate the coupling submatrix Z_{kl}, in a process very much like the one which generates Z_{kk}. The reduced submatrix Z'_{kl} is obtained from (11):

$$Z'_{kl} = U_k^* * Z_{kl} * U_l. \tag{22}$$

Matrices U_k and U_l have already been calculated in the first loop and were left on the GPU. For the matrix multiplications, we again use cuBLAS routine `cublasCgemm`.

Once all Z' submatrices have been calculated, all that remains is to assemble matrix Z'. Matrices Z'_{kl} are sent back to the GPU for assembly. This yields matrix Z'. The rest of the data used so far (all data necessary to calculate matrices Z_{kl}) is removed, since it is no longer necessary, and only matrix Z' and matrices U_k remain in the GPU.

Matrix Z' can now be used whenever it is necessary to find out the currents induced by a given excitation E. In order to do this, it is necessary to transform the excitation to the CBF space using the transformation matrices U_k:

$$E'_k = U_k^* * E_k, \tag{23}$$

where

$$E = \left(E_k\right)^T; \quad k = 1, \ldots, n. \tag{24}$$

E_k represents, in matrix form, the excitations in the subdomains of standard block k, with one column vector for each excitation. The transformation is done block by block. Once the transformed excitations are assembled together again, the linear system:

$$E' = Z' * J' \tag{25}$$

has to be solved, which yields the currents J'. This is done in the GPU. The resulting currents can then be transformed back into the subdomain space. For solving the linear systems and for matrix multiplications, we use the CULA and cuBLAS routines, respectively, already mentioned above.

3.3. Coupling Submatrix. The process for generating each submatrix of the coupling matrix is similar for standard blocks and for extended blocks. The only difference lies in that, for extended blocks, those subdomains which are close enough to the standard block must be included for the calculation. But, once the list of subdomains involved has been obtained, the rest of the process is the same. The pseudocode for generating a coupling matrix looks as follows (Algorithm 2).

The coupling of two subdomains is calculated according to (2). It has an inductive term, described in (3), and a capacitive term, described in (4a) and (4b). The inductive term is calculated as the line integral along the razor-blade of the victim subdomain of a function which is the surface integral

```
;generate input data for kernels
⋮
calculate_list_of_integration_points();
⋮
kernel_calculate_2_point_surface_integrals<<< ... >>>(...);
⋮
kernel_calculate_4_point_surface_integrals<<< ... >>>(...);
⋮
kernel_calculate_10_point_surface_integrals<<< ... >>>(...);
⋮
kernel_calculate_2_point_line_integrals<<< ... >>>(...);
⋮
kernel_calculate_4_point_line_integrals<<< ... >>>(...);
⋮
kernel_calculate_2_point_line_integrals<<<...>>>(...);
```

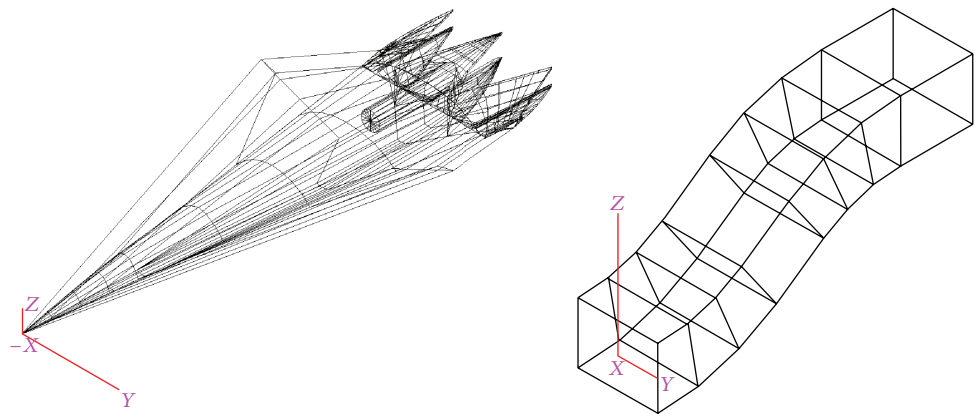
Algorithm 2: Pseudocode of the submatrix generation.

of the rooftop function, which extends on the two subpatches on either side of the active subdomain. The capacitive term needs no line integral but only four surface integrals. The integrals are calculated using Gaussian quadratures like the one in (7). The number of integration points can be fixed at 10 for excellent accuracy. Still good accuracy, but with a lot less computing load can be obtained if the number of integration points is reduced, depending on the distance between the active and passive subdomains considered. We use 10, 4, or 2 points, depending on the distance.

The general algorithm for the case when we use a variable number of integration points is described in Algorithm 2. In this description, some parts are omitted for clarity and each omission is indicated by an ellipsis. Omissions represent data allocation, handling, copying, or retrieval. Before executing the loops in Algorithm 1, a kernel in the GPU analysed if each subdomain belongs to each extended block. The result has been a list of subdomains for each block in the geometry and it has been transferred to the CPU for use at this point, when matrix Z_{kl} is calculated. What is needed is a list of subdomain pairs, one active and one victim. In order to prepare this list, the lists for the corresponding geometry blocks are consulted now. The victim subdomains correspond to extended block l and the active subdomains to extended block k. Further preprocessing involves calculating the number of integration points for each subdomain pair. All these operations are executed in the CPU. Once the number of integration points is calculated for all subdomain pairs, each one is included in one of three lists, one for each number of integration points (10, 4, and 2). We create a different kernel to process each list, because the amount of processing varies a lot with the number of integration points. The integration kernels are described below.

Kernels `kernel_calculate_X_point_surface_integrals` each process one of the lists above, and all three work similarly. Each kernel is organized in thread-blocks. One thread-block calculates one surface integral. One subdomain pair will require several blocks of threads, depending on how many points the line integral is calculated with. The results of the integrals are stored in a data structure in global memory to be used later by other kernels. The integrals correspond either to an inductive term or to a capacitive term of the coupling. The calculations of each type of integral are similar, but slightly different, and each thread-block behaves a little different in order to adapt to the calculation required. Our original implementation had only one kernel to handle all numbers of integration points. The result was decreased performance due to two factors. First, the thread-blocks had to find out how many integration points to use, which implies a conditional programming structure. This usually degrades performance. The second factor was that the workload for the thread-blocks was uneven, and this also degrades performance. When we decided to use three separate kernels for the surface integrals, the improvement was considerable.

All surface integrals work analogously, varying only the number of integration points. Each thread in one thread-block evaluates the rooftop function in one point in the surface of one of the two subpatches whose common border is the active subdomain. Then, the results are added following the Gaussian quadratures method, first on one dimension of the parametric space, then on the other. This process is called a "parallel reduction" in computing jargon, and to reduce the number of branches in a thread and to improve efficiency, all threads calculate the addition, and we keep the result of just one of them. The single rooftop evaluation values and the partial addition results are stored in shared memory, so that all threads in the thread-block can have access to all the results of all other threads in order to add them together.



(a) Airplane (b) Cobra cavity

Figure 4: Geometries.

Finally, only one of the threads stores the final result of the integral in global memory.

When all the surface integrals have been calculated, it is the turn for the line integrals. The results of the surface integrals are stored in a structure in global memory, as explained above, so that the kernels which calculate the line integrals can have access to them. These kernels, `kernel_calculate_X_point_line_integrals`, use Gaussian quadratures to calculate the final value of the corresponding element of the coupling submatrix and add up the capacitive term, as described in (4a) and (4b). The rationale for using a kernel for each number of integration points is the same as for the surface integrals. The result of each line integral is one element of matrix Z_{kl}. The combined results make up the whole coupling submatrix, which will be processed as explained in the previous subsection.

4. Results

This section is organized as follows. First, we validate the version of the application which uses the GPU by comparing its results with those from the version which runs only on the CPU. This will prove that the results obtained when using the GPU are accurate enough that they can be accepted. At the same time, we measure the speedup obtained when using the GPU to calculate the coupling matrix and the reduced matrix in a single block to get an idea of the highest bound of the speedup which can be obtained. After this, we analyse the impact of dividing the geometry into blocks. This division causes a drop in performance, but it is necessary if we want to deal with electromagnetically large geometries which would take up too much memory to deal with in just one block.

The computer where the applications run has an Intel i3 processor with 8 GB RAM and a TESLA C2075 GPU with 6 GB RAM. It runs 64-bit Windows 7. The CUDA toolkit we use is 3.1, which includes the cuBLAS libraries, and the version of CULA is 2.1. In order to compare to sequential execution time, when necessary, we use Intel's Math Kernel Library, which includes routines for system solve, SVD, and matrix multiplication. Intel's Math Kernel Library routines run on the CPU. The geometries we have considered are mainly two: an airplane model and the cobra cavity (see Figure 4). In some measurements we also use results for a third geometry, a dihedral. Depending on how many subdomains we need for our experiments, we use a different frequency. In the case of the airplane, when we want to be able to process it in one block, we use 21e8 s^{-1}. This lets it have an electrical size of about 4 lambdas. The number of subdomains is about 3500 if we use $\lambda/10$ as discretization factor. Then, when we want to test the method which deals with the geometry in several blocks, we use 28e8 s^{-1}, and this yields about 7000 subdomains. In the case of the cobra cavity, the results obtained are similar, so we only show results for the case when we deal with it in several blocks. For this, we use a frequency of 40e8 s^{-1}, and similarly obtain about 7000 subdomains. The electrical size in this case is about 6 lambdas.

4.1. Validation. To validate the application, we use both the airplane and the cobra cavity, shown in Figure 4. We calculate the Monostatic Radar Cross-Section (RCS) to compare the results obtained from the GPU and the CPU. For the PWS, we use a set of θ-polarized plane waves incident on each geometry with angle θ ranging from 0 to 180 degrees, in steps of 10 degrees, and φ ranging from 0 to 350, also in steps of 10 degrees. Then, we calculate the RCS for θ-polarized waves impinging on the geometry, similarly from all directions, in one case using the CPU for the calculations, and in another using the GPU as described in previous sections. Figures 5 and 6 show some of these results obtained. For all incident waves (i.e., for the full ranges of both θ and φ), the results are practically identical, and so, we only show an example. In both Figures 5 and 6, the full range of θ is shown for a plane with a constant φ of 20 degrees. Also in both Figures 5 and 6, the thick, solid, light stroke corresponds to the CPU results, and the thin, intermittent, dark stroke to the results of the GPU. The results clearly validate the adaptation of the CBFM to CUDA.

The next step is to compare execution times for the CPU and the GPU methods, but, before that, we want to

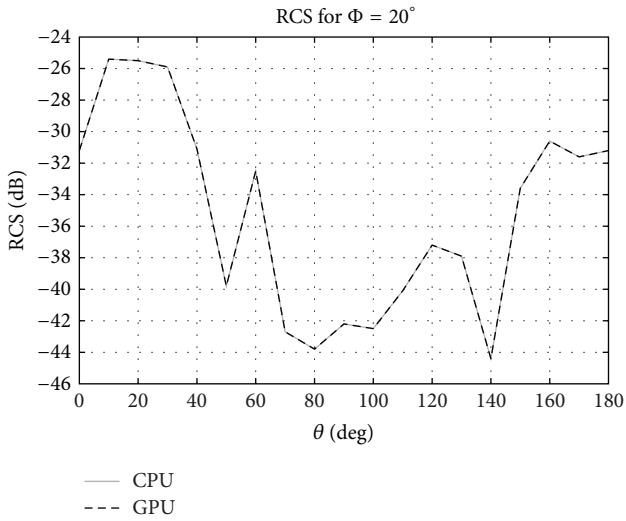


Figure 5: Monostatic RCS for the cobra cavity.

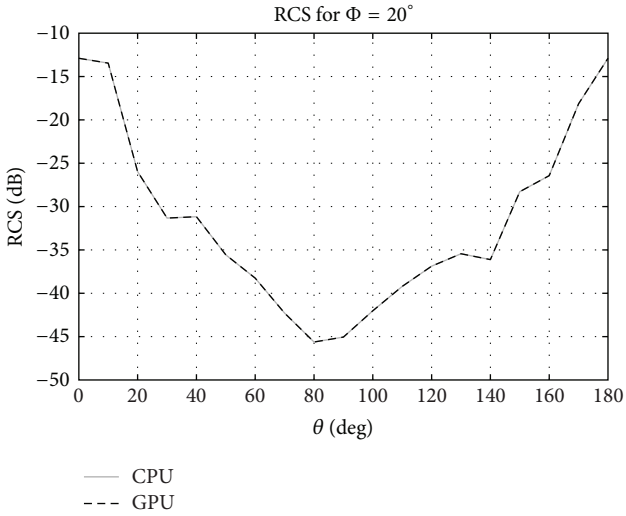


Figure 6: Monostatic RCS for the airplane.

know the effect on the algorithm of adapting the number of integration points in the integrals to the distance between the subdomains. To do this, we execute two versions of the GPU code which calculates the impedance matrix in the method of moments. In one of them, the number of integration points is fixed, and in the other, it is adapted to the distance between subdomains. By reducing the number of integration points when the subdomains are far from each other, the workload of the application is reduced without an impact on its accuracy. But, on the other hand, the room for improvement (in terms of execution time) is also reduced. Figure 7 shows the speedup obtained for these two versions of the GPU code when compared with an equivalent version running on the CPU.

As can be seen, the speedup obtained for a constant number of integration points is much greater. This comes

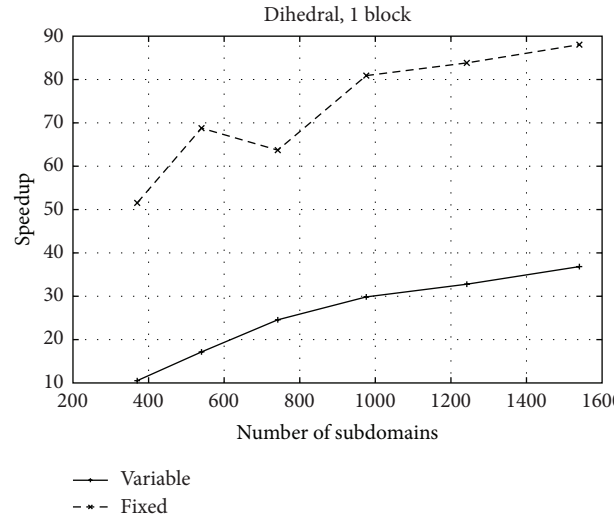


Figure 7: Comparison of speedup for the MoM matrix for fixed and variable number of integration points.

from two factors: first, the amount of computer work done, and second, the complexity of the GPU version. With a fixed number of 10 integration points, the algorithm takes much longer to run, especially on the CPU. The GPU version has more to work on, and the speedup is greater. As for the second factor, as has been explained above, if the number of integration points varies, the result is that the number of different kernels needed for calculating the integrals grows. Nonetheless, the speedups obtained, even under these circumstances, are considerable.

In order to be able to obtain results for different numbers of subdomains, we use the same geometry and modify the discretization factor. We start with $\lambda/10$, which is a reasonable value and which provides about 1600 subdomains. Then we use λ/i, with i ranging from 9 to 6. This way, we can easily obtain results for geometries with fewer subdomains and the process of adapting the geometry is less cumbersome. We use this process in all the graphs in this section. The bigger number of subdomains in the graphs always corresponds to a discretization factor of $\lambda/10$.

The group of Figures 8–12 corresponds to a comparison between the GPU and the CPU version of the CBFM. We want to know how much faster the GPU version is when calculating the MoM system matrix Z, when calculating the reduced CBFM system matrix Z', and when solving the linear system of equations once it has been reduced with the CBFM. From now on, we use a variable number of integration points, since the overall execution time is less, both in the CPU and in the GPU versions of the algorithm. Also, at first we use geometry sizes which the GPU can handle in one block. Later we will be interested in the effect of partitioning the geometry, but for now we concentrate on the geometry in one piece. Figure 8 shows the speedups for the calculation of Z and Z'. Note that, in order to obtain Z', Z is needed, so the speedup for Z' is a sort of aggregate value. The results shown correspond to the airplane, but similar values are obtained for the cobra cavity.

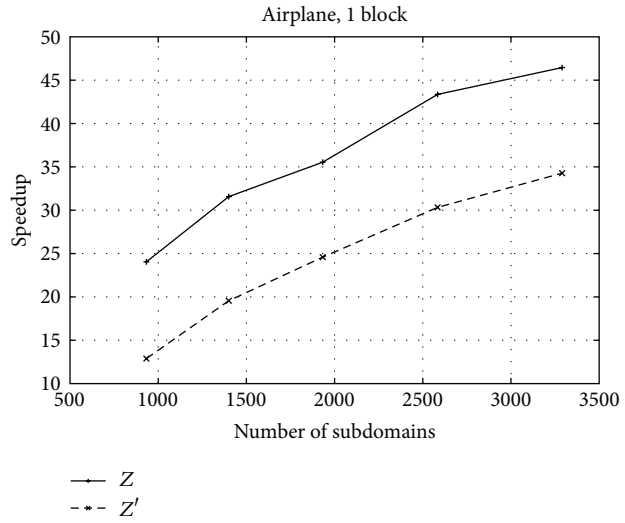


Figure 8: Speedups for calculating the system matrix (Z) and the reduced system matrix (Z') for the airplane.

The results are obtained using a single CPU thread, since we need a reference value for comparison purposes. As was expected, the improvement provided by the GPU grows with the size of the problem. There are a number of tasks which take an amount of time which is roughly independent of the problem size, like memory initialization, transfer of geometry data to the GPU, and so forth. These tasks have a proportionally greater influence on total execution time for smaller problems. As problems become greater, their impact is smaller and the speedup grows. Also worth noticing is how the aggregate speedup is smaller than the speedup for matrix Z. This is because the calculation of matrix Z is a very parallel task, where the calculation of each element of the matrix is independent of the rest. This is not the case for the subsequent steps of the algorithm, since solving the system matrix has a smaller degree of parallelism. Something similar happens with the Singular Value Decomposition (SVD). The net speedup is slightly reduced.

If the GPU is to handle the geometry in one block, it is interesting to know where the limit is, in terms of subdomains. This limit is more or less independent of the geometry, since its geometrical and electromagnetic data take up only a small space. Rather, it depends on the information which is provided to the kernels regarding the subdomains that must be processed by each thread, when calculating the MoM matrix Z, and the space needed for the intermediate results. Currently, our application can handle a geometry of about 3500 subdomains in one block. After the matrix is calculated, the rest of the process takes up less space, so this is the global limit. This information is important for those times when the geometry must be divided into blocks, as the bigger the blocks, the more efficient the process.

Figure 9 shows how much the system matrix Z is reduced by the CBFM. Again we show results for the airplane, but they are similar for the cobra cavity. We continue to use a system size which allows us to deal with it in the GPU in a single block. Figure 9 shows how many times the order of matrix Z is greater than that of matrix Z', that is, the ratio of regular basis functions to macrobasis functions or CBFs. As can be seen, the reduction which the CBFM provides is considerable. Table 1 shows the numerical data.

Table 1: Number of regular basis functions and resulting CBFs.

No. of regular basis functions	933	1400	1933	2584	3290
No. of CBFs	91	86	87	89	90

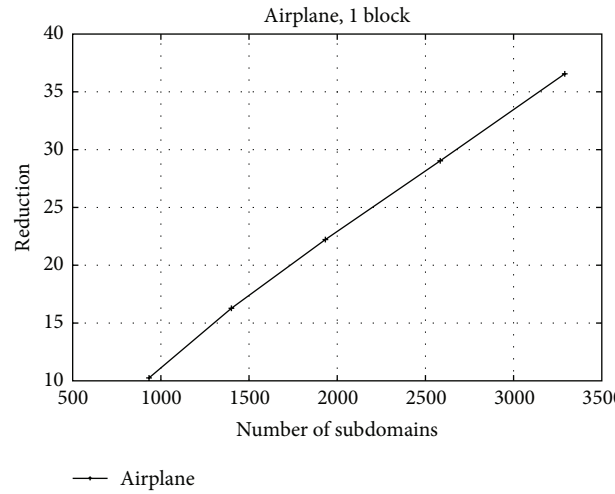


Figure 9: Reduction of problem size by the CBFM.

Once the problem has been transformed to CBFs, the reduced matrix Z' obtained can be used to analyse the effect of any excitation. To do this, all there is to do is to express this excitation in terms of CBFs, solve the reduced linear system in CBF representation to obtain the induced currents, and transform these back to subdomain representation. Figure 10 shows the speedup obtained when solving the reduced linear system in the GPU with respect to the same task executed on the CPU. The time for conversion to and from CBFs is negligible. This operation is somewhat independent of all the previous tasks in that, if the reduced matrix Z' has already been obtained and stored, the only work left to do is to solve the reduced system.

The results above are obtained from configurations with which the geometry can be handled in one block. But, for arbitrarily sized geometries, it is interesting to know how dealing with them in blocks affects the general performance of the method. Figures 11 and 12 present the speedups for the calculation of Z and Z' for the airplane and for the cobra cavity, respectively. It must be noted that, for this comparison, when running on the CPU, the geometry is still dealt with in one block, as opposed to the GPU, where it is divided into pieces as great as can fit in the memory of the card. This does not provide a fair comparison between CPU and GPU performance, but for us it is interesting, as it provides an insight into how much performance is lost by dividing the geometry in blocks. As can be seen, the speedup is less

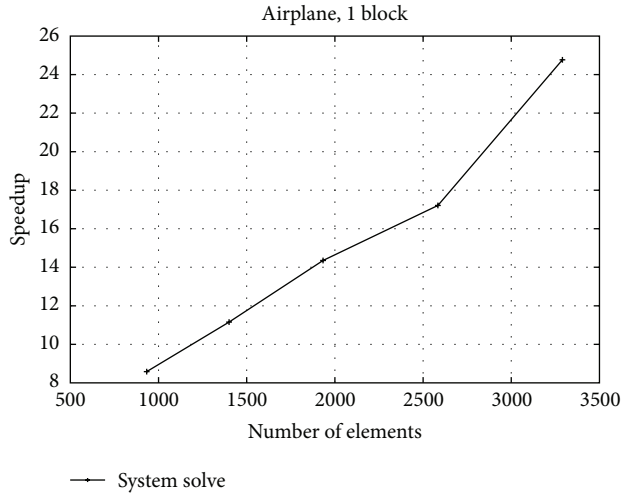


Figure 10: Speedup when solving the reduced linear system.

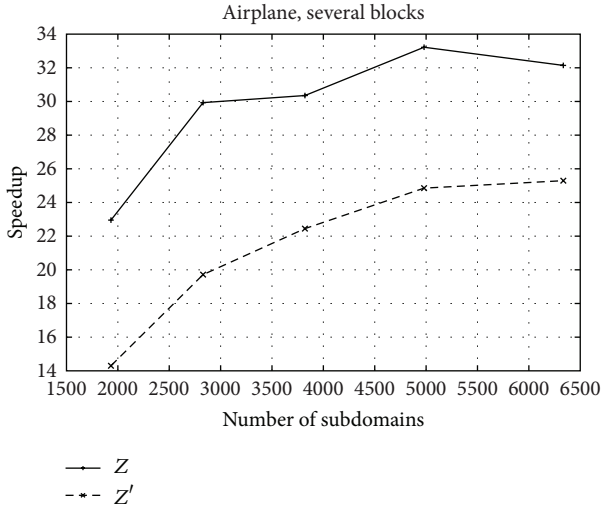


Figure 11: Speedup when dividing the airplane geometry in several blocks.

than what Figure 8 shows, but it is still a very interesting result.

Dividing the geometry into blocks causes a loss of performance, but, on the other hand, it allows us to circumvent the limitation imposed by the reduced GPU memory.

Reduced matrix Z' is calculated block-wise, and the number of blocks of the geometry is not a problem anymore. However, when using the complete Z' matrix to calculate the RCS for an incident wave, it is necessary to solve the corresponding linear system. In that case, the size of the system is a limiting factor. We have tested the system solve function on our GPU and it can handle a system with up to 13000 unknowns which, using the results in Table 1, would indicate an overall geometry limit of 450000 subdomains. Note that this is just an estimation, though, as we have not used our application on a system this big yet.

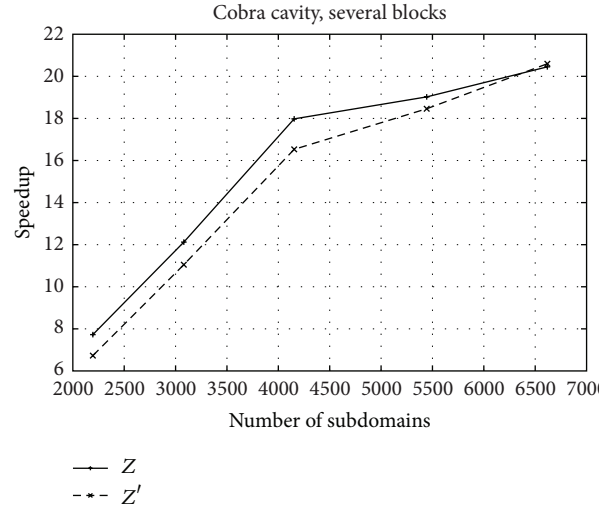


Figure 12: Speedup when dividing the cobra cavity geometry in several blocks.

5. Conclusions and Future Work

We have presented a procedure for applying the CBFM which takes advantage of current GPUs and overcomes one of the most important limitations which this novel computing system suffers, namely, the limitation of GPU memory. We have validated the use of a GPU in terms of accuracy of the results, and we have shown that, when the surface under analysis can fit in the memory of the card, the speedups obtained are remarkable, which agrees with other similar results present in the literature. In order to be able to deal with big geometries, we have adapted the existing CBFM to process the geometry in blocks, and we have found that, even though there is a drop in performance, the speedup is still considerable.

For still better performance, some issues have to be dealt with, mostly the partitioning of the geometry in blocks. It is advisable to find a way to partition it so that the subdomains are evenly distributed among the blocks, as this would allow the filling of the GPU with as much data as it can handle, thus improving efficiency.

Better performance could also be obtained by tweaking the application with respect to the use made of the different layers of the architecture. In this work, we have taken care to map all the data which are shared by the threads of each block to shared memory, but we have not taken into consideration how the data is arrayed in global memory. This disposition of data in global memory can have an important influence on the general running time of the application and is a factor which must be studied in future work.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

This work has been supported in part by the Comunidad de Madrid Project S-2009/TIC1485, the Spanish Department of Science, Technology Projects TEC2010-15706, and CONSOLIDER-INGENIO no. CSD-2008-0068.

References

[1] R. F. Harrington, *Field Computation by Moment Methods*, McMillan, New York, NY, USA, 1968.

[2] V. V. S. Prakash and R. Mittra, "Characteristic basis function method: a new technique for efficient solution of method of moments matrix equations," *Microwave and Optical Technology Letters*, vol. 36, no. 2, pp. 95–100, 2003.

[3] N. Engheta, W. D. Murphy, V. Rokhlin, and M. S. Vassiliou, "The Fast Multipole Method (FMM) for electromagnetic scattering problems," *IEEE Transactions on Antennas and Propagation*, vol. 40, no. 6, pp. 634–641, 1992.

[4] W. C. Chew, J. Jin, E. Michielssen, and J. Song, Eds., *Fast and Efficient Algorithms in Computational Electromagnetics*, Artech House, Norwood, NJ, USA, 2001.

[5] D. J. Ludick and D. B. Davidson, "Investigating efficient parallelization techniques for the Characteristic Basis Function Method (CBFM)," in *Proceedings of the International Conference on Electromagnetics in Advanced Applications (ICEAA '09)*, pp. 400–403, September 2009.

[6] E. García, L. Lozano, M. J. Algar, and F. Cátedra, "A study of the efficiency of the parallelization of a high frequency electromagnetic approach for the computation of radiaton and scattering considering multiple bounces," *Computer Physics Communications*, vol. 184, no. 1, pp. 45–50, 2013.

[7] E. Lezar and D. B. Davidson, "GPU-accelerated method of moments by example: monostatic scattering," *IEEE Antennas and Propagation Magazine*, vol. 52, no. 6, pp. 120–135, 2010.

[8] D. P. Zoric, D. I. Olcan, and B. M. Kolundzija, "GPU accelerated computation of radar cross sections with multiple excitations," in *Proceedings of the European Conference on Antennas and Propagation (EUCAP '13)*, 2013.

[9] E. Lezar and D. B. Davidson, "GPU acceleration of method of moments matrix assembly using Rao-Wilton-Glisson basis functions," in *Proceedings of the International Conference on Electronics and Information Engineering (ICEIE '10)*, pp. V156–V160, August 2010.

[10] D. P. Zoric, D. I. Olcan, and B. M. Kolundzija, "Solving electrically large EM problems by using out-of-core solver accelerated with multiple graphical processing units," in *Proceedings of the IEEE International Symposium on Antennas and Propagation and USNC/URSI National Radio Science Meeting (APSURSI '11)*, 2011.

[11] S. Peng and Z. Nie, "Acceleration of the method of moments calculations by using graphics processing units," *IEEE Transactions on Antennas and Propagation*, vol. 56, no. 7, pp. 2130–2133, 2008.

[12] E. Lezar and D. B. Davidson, "GPU acceleration of electromagnetic scattering analysis using the method of moments," in *Proceedings of the 13th International Conference on Electromagnetics in Advanced Applications (ICEAA '11)*, pp. 452–455, September 2011.

[13] I. Kiss, P. T. Benkö, and S. Gyimóthy, "Fast analysis of metalic antennas by parallel moment method implemented on CUDA," *International Journal of Applied Electromagnetics and Mechanics*, vol. 39, no. 1–4, pp. 677–683, 2012.

[14] T. Topa, A. Karwowski, and A. Noga, "Using GPU with CUDA to accelerate MoM-based electromagnetic simulation of wire-grid models," *IEEE Antennas and Wireless Propagation Letters*, vol. 10, pp. 342–345, 2011.

[15] F. Rivas, L. Valle, and M. F. Cátedra, "A moment method formulation for the analysis of wire antennas attached to arbitrary conducting bodies defined by parametric surfaces," *Applied Computational Electromagnetics Society Journal*, vol. 11, no. 2, pp. 32–39, 1996.

[16] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in FORTRAN: The Art of Scientific Computing*, Cambridge University Press, Cambridge, UK, 2nd edition, 1992.

[17] C. Delgado, M. F. Catedra, and R. Mittra, "Application of the characteristic basis function method utilizing a class of basis and testing functions defined on NURBS patches," *IEEE Transactions on Antennas and Propagation*, vol. 56, no. 3, pp. 784–791, 2008.

[18] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[19] NVIDIA Corporation, "*CUDA Programming Guide*," NVIDIA, Santa Clara, Calif, USA, 2013, http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[20] NVIDIA Corporation, "*CUBLAS Reference Manual*," NVIDIA, Santa Clara, Calif, USA, 2013, http://docs.nvidia.com/cuda/cublas/index.html.

[21] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra 'subprograms for FORTRAN usage'," *ACM Transactions on Mathematical Software*, vol. 5, no. 3, pp. 308–323, 1979.

[22] J. R. Humphrey, D. K. Price, K. E. Spagnoli, A. L. Paolini, and E. J. Kelmelis, "CULA: hybrid GPU accelerated linear algebra routines," in *Defense and Security Symposium (DSS '10)*, Proceedings of SPIE, 2010.

[23] E. Anderson, Z. Bai, C. Bischof et al., *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, Pa, USA, 1999.