Check for updates

IET Microwaves, Antennas & Propagation

Research Article

IET Journals
The Institution of Engineering and Technology

Efficient strategy for parallelisation of multilevel fast multipole algorithm using CUDA

ISSN 1751-8725
Received on 17th July 2018
Revised 27th February 2019
Accepted on 8th April 2019
E-First on 8th May 2019
doi: 10.1049/iet-map.2018.5568
www.ietdl.org

Eliseo García[1] ✉, *Carlos Delgado*[2], *Lorena Lozano*[2], *Felipe Cátedra*[2]

[1]*Automatics Dep., University of Alcalá, Alcalá de Henares, Spain*
[2]*Computer Sci. Dep., University of Alcalá, Alcalá de Henares, Spain*
✉ *E-mail: eliseo.garcia@uah.es*

Abstract: The multilevel fast multipole algorithm is a popular technique that enables the efficient solution of the method of moments (MoM) matrix equations. In this work, the authors address the adaptation of this method to the compute unified device architecture (CUDA), a relatively new computing infrastructure provided by NVIDIA, and the authors take into account some of the limitations that appear when the geometry under analysis becomes too large to fit into the memory of graphics processing units.

1 Introduction

The method of moments (MoM) [1] is a well-known technique for the electromagnetic analysis of arbitrary 3D geometries. It defines a system of linear equations by discretising the corresponding integro-differential equations. The unknowns of such system are the coefficients of the induced currents with respect to a set of basic functions. The discretisation process involved, in order to provide accurate results, must divide the geometry into subpatches of, typically, around one-tenth of the size of the wavelength. As the size of the geometry grows, the problem becomes computationally expensive very quickly, and this is the reason for the emergence of several techniques whose objective is to make larger problems manageable. Three of these efficient approaches are the characteristic basis function method (CBFM) [2], the fast multipole method (FMM) [3] and its multilevel approach, the multilevel fast multipole algorithm (MLFMA) [4–6]. The FMM and MLFMA compute the coupling matrix by only storing the interactions between geometrically close elements and consider the interactions between distant elements by means of efficient matrix–vector multiplications in the iterative process.

With the proliferation of relatively non-expensive high-performance computers, a noticeable effort has been made to impulse the development of parallel versions of these numerical methods [7–10]. Many research groups have implemented efficient strategies for the parallelisation of the MLFMA, applying the message passing interface (MPI) paradigm [11–13]. Our work focuses on accelerating the MLFMA through a relatively new programming paradigm called compute unified device architecture (CUDA). The main objective of CUDA is to allow the use of graphics processing units (GPUs) for high-performance scientific computing. The performance gains using CUDA are quite remarkable. It has recently been used for tasks such as the calculation of the impedance matrix in the MoM, or for the solution of the linear system that the MoM produces. An example of the first case is shown in [14]. The reported speedup is up to 70, compared to the same operation performed on the CPU. In [9], the authors obtain speedup results for the same task up to 140. They then perform a LU decomposition of the matrix, but this does not benefit too much from the GPU, and the total speedup reported for the entire MoM execution is about 45. The LU decomposition of the matrix system is specifically addressed in [15]. The reported speedup, using CUDA, is up to 20 with three GPUs.

In order to address arbitrarily large geometries, the authors divide in [16] the matrix into blocks that are stored in hard disk drives instead of RAM. They propose a scheme in which the impedance matrix is divided into submatrices, and each submatrix is calculated at the same time, in order to overcome the relatively low amount of memory available in off-the-shelf graphics cards. The overall gain is, in this case, a factor of 30 for the calculation of the impedance matrix, even taking into account the penalty introduced by this partitioning. This work is especially remarkable because the authors did not have at their disposal the tools later provided by NVIDIA for CUDA. A good example of the fine adjustment that is sometimes necessary when using CUDA is shown in [17], in this case applied to wire-grid models. The authors analyse the resources of the graphics card and show how to partition the tasks among its computing cores to achieve the maximum possible gain.

The application of GPUs for the application of the MLFMA is described here. Several works have been published in this field [18–20]. The main computational burden of GPUs is their limited memory size when compared to CPU capabilities. Some GPU cards with a memory capacity of 24 GB have been recently introduced, which improves the computational range of the previous cards, but it is still inadequate when analysing moderately sized electromagnetic problems using only the GPU memory.

The combination of OpenMP and CUDA parallelisation techniques applied to the MLFMA for complex medium-size problems is shown in [18]. Some techniques developed to apply GPU computing for the solution of large problems are based on transferring data between GPU and CPU during the computation [14] with the corresponding reduction of efficiency. A multi-GPU scheme that benefits from an increased capacity for the solution of larger problems is presented in [19].

Table 1 shows a comparison between the performance of different methods mentioned in this section and the size of the problems addressed.

This work aims to contribute to a further optimisation of the simulation process dealing with some computational limitations. Here, a novel technique for computing large problems in a single GPU scheme is presented, based on the reduction of the storage requirements for the MLFMA data. This approach involves the introduction of an algorithm that reduces the data that is stored at the expense of performing a higher number of operations. However, these operations have a good data-parallelism, so they are suitable to be run on GPU cards, obtaining an efficient solution that allows to overcome the memory limitations of the cards while taking advantage of their computational benefits. The mathematical description of the algorithm is shown in Section 3.

In our work, we use all the experience provided by the aforementioned papers and managed on our own numerical scheme

© The Institution of Engineering and Technology 2019

Table 1 Speedup comparison for some of the existing techniques

Technique reference	Speedup in the matrix computation	Speedup in the whole problem	Electrical size
[9]	140	45	about 5λ (7701 unknowns)
[14]	70	no applicable	about 1λ (3008 unknowns)
[15]	no applicable	20	about 240λ (151,898 unknowns)
[16]	30	18	about 2λ (9936 unknowns)
[18]	37	26	about 2λ (37,905 unknowns)
[19]	124	21	about 30λ (342,237 unknowns)

to accelerate the MLFMA with CUDA. Since the MLFMA is based on the MoM, we must first calculate the impedance matrix for the geometry. Most of the previous works use a meshing scheme based on triangulation. We use square NURBS patches because, in our experience, this kind of patches offer an improved fit to the geometry, requiring the use of fewer surfaces [21], but the proposed technique can be applied for any type of mesh. When calculating the elements of the impedance matrix, we use a variable number of integration points, depending on the distance between the active and passive subdomains whose coupling is being calculated. When they are geometrically close, a greater number of integration points are recommended to provide better accuracy. When they are further apart, we can reduce this number to make the process more efficient without the risk of compromising the accuracy.

The rest of this paper is organised as follows: in Section 2, we review the MoM and its combination with the MLFMA, and describe the benefits obtained from this combination. The implementation of the algorithm using CUDA is described in Section 3, which is followed by some experimental results in Section 3. We dedicate Section 5 to the conclusions that can be derived from this work.

2 MoM and MLFMA

The MoM [1] is used to transform a number of integral-differential equations into a set of linear equations. When applied to electromagnetic simulation the final result is the linear system of (1), where $\boldsymbol{Z}$ is known as the *coupling* or *impedance matrix*, the $\boldsymbol{V}$ vector contains the *impressed voltage* for each subdomain, representing the excitations, and the $\boldsymbol{J}$ vector contains the induced current coefficients to be calculated.

$$[\boldsymbol{V}] = [\boldsymbol{Z}][\boldsymbol{J}] \tag{1}$$

Using rooftop functions as basic functions, composed of S_1 and S_2 patches, and razor-blade functions as testing functions [22], the term z_{ij} of the coupling matrix can be expressed as:

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

where (see (3)) and

$$z_{ij}^{\mathrm{cap}} = P^{S_{j1}}(\boldsymbol{r}_b) - P^{S_{j1}}(\boldsymbol{r}_a) - \left[P^{S_{j2}}(\boldsymbol{r}_b) - P^{S_{j2}}(\boldsymbol{r}_a)\right], \tag{4a}$$

Terms like $P^{S_{j1}}(\boldsymbol{r}_b)$ can be obtained from:

$$P^{S_{j1}}(\boldsymbol{r}_b) = \frac{-1}{\mathrm{j}4\pi\omega\varepsilon_0}\int_{S_{j1}} \frac{G(\boldsymbol{r}_b, \boldsymbol{r}')}{A_{S_{j1}}}\mathrm{d}S' \tag{4b}$$

Equations (3) and (4b) include terms such as the Green's function $G(\boldsymbol{r}, \boldsymbol{r}')$, defined as:

$$G(\boldsymbol{r}, \boldsymbol{r}') = \frac{\mathrm{e}^{-\mathrm{j}k|\boldsymbol{r}-\boldsymbol{r}'|}}{|\boldsymbol{r}-\boldsymbol{r}'|}. \tag{5}$$

They also include terms like $J_j^{S_{j1}}$, which is the current density of the j-subdomain over the S_{j1} patch, whose area is represented by $A_{S_{j1}}$. $\boldsymbol{r}'$ and $\boldsymbol{r}$ are the position vectors for the points on the j-source subdomain and the i-observation subdomain, respectively. Finally, $\boldsymbol{r}_a$ and $\boldsymbol{r}_b$ are the end-points of the razor-blade for the i-subdomain. A more detailed mathematical description of the process that leads to these equations can be found in [23].

Computationally, the integrals for (3) and (4b) are calculated using the Gaussian Quadrature [24]. The integral of a given function f is approximated by a series in which we select weights and abscissas in an appropriate way:

$$\int_a^b W(x)f(x)\mathrm{d}x \simeq \sum_{j=1}^{N} w_i f(x_i) \tag{6}$$

where w_i is the Gaussian weights and $W(x)$ is a function chosen to eliminate the singularities of the integrand. All the integrals are treated similarly. In the case of the integrals between the brackets of (3), the transformation is:

$$\int_{S_{j1}} G(\boldsymbol{r}, \boldsymbol{r}')J_j^{S_{j1}}(\boldsymbol{r}')\mathrm{d}S' \simeq \sum_{l=1}^{N}\sum_{k=1}^{M} w_l w_k G(\boldsymbol{r}_l, \boldsymbol{r}'_k)J_j^{S_{j1}}(\boldsymbol{r}'_k) \tag{7}$$

where $G(\boldsymbol{r}_l, \boldsymbol{r}'_k)$ is the evaluation of the Green's function at the $\boldsymbol{r}'_k$ source and the $\boldsymbol{r}_l$ observation point. In this case, we choose $W(x) = 1$, so that $f(x)$ is the product of G and J. N is the number of sampling points chosen and depends on the desired precision.

2.1 Multilevel fast multipole algorithm

Once the matrix $[\boldsymbol{Z}]$ has been calculated, it can be used to solve the electromagnetic problem. However, $[\boldsymbol{Z}]$ becomes very large quickly as the size of the geometry increases. Solving the linear equation system given by (1) using a direct method can be very time-consuming and impose a memory bottleneck.

In order to reduce this problem, some techniques can be applied. One of the most common approaches to easing the burden on the computational resources involves storing only the interactions of geometrically close elements of the coupling matrix and computing the interactions between distant elements via the MLFMA [4].

In the application of the MLFMA, we compartmentalise all the geometry into several first-level cubical groups which, in turn, generate higher order cubes as they are grouped. For the first level, the cubes include a few basis-functions, and the coupling between basis functions associated with geometrically close cubes is calculated in a rigorous way (2) and stored for later use. The interactions between geometrically distant elements are computed very efficiently in the iterative solution process and do not need to be stored [25]. For these interactions, the equations include the Green function, shown in (5). Applying the addition theorem, it can be expressed as:

$$G(\boldsymbol{r}, \boldsymbol{r}') = \frac{\mathrm{e}^{-\mathrm{j}k|\boldsymbol{r}-\boldsymbol{r}'|}}{|\boldsymbol{r}-\boldsymbol{r}'|} = \frac{\mathrm{e}^{-\mathrm{j}k|\boldsymbol{D}+\boldsymbol{d}|}}{|\boldsymbol{D}+\boldsymbol{d}|} = \int \tau_{\mathrm{L}}(k, |\boldsymbol{D}|, \hat{k}\hat{D})\mathrm{e}^{-\mathrm{j}k(\mathrm{k}\mathrm{d})}d^2\hat{k} \tag{8}$$

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

© The Institution of Engineering and Technology 2019

17518733, 2019, 10, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/iet-map.2018.5568 by Universidad De Alcala, Wiley Online Library on [22/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

where the integral is defined over all the directions of the unit sphere, and considering that

$$\boldsymbol{D} + \boldsymbol{d} = \boldsymbol{r} - \boldsymbol{r}', |\boldsymbol{d}| < |\boldsymbol{D}| \tag{9}$$

If we denote $\boldsymbol{D}$ as the distance vector between the centre of the cubes m and m' containing subdomains i and j, respectively, and $|\boldsymbol{d}_{m'i}|$ and $|\boldsymbol{d}_{\mathrm{mj}}|$ as the distance vector between each subdomain and the centre of its cube, the coupling terms between distant subdomains i and j can be computed in the iterative process as matrix–vector products applying the expression shown in (10). For the electric field integral equation (EFIE) case, the coupling between element j and element i can be computed using:

$$A_{\mathrm{ji}} = \int V_{\mathrm{mj}}^{\mathrm{AGG}}(\hat{k}) \tau_{\mathrm{mm}'}(\hat{k}, \boldsymbol{r}_{\mathrm{mm}'}) V_{m'i}^{\mathrm{DIS}}(\hat{k}) d^2\hat{k} \tag{10}$$

where $V_{\mathrm{mj}}^{\mathrm{AGG}}(\hat{k})$ represents the aggregation term for the j-subdomain to the centre of the cube m, expressed as follows:

$$V_{\mathrm{mj}}^{\mathrm{AGG}}(\hat{k}) = \int_u \int_v \mathrm{e}^{-\mathrm{j}\hat{k}\cdot\boldsymbol{r}_{\mathrm{jm}}} (\bar{I} - \hat{k}\hat{k}) T_j(u, v) \mathrm{d}u\, \mathrm{d}v \tag{11}$$

where $\boldsymbol{r}_{j,m}$ represents the vector that extends from the sampling point to the aggregation point, and $T_j(u, v)$ represents the basis function associated to the source element j.

Analogously, the disaggregation term can be computed as:

$$V_{m'i}^{\mathrm{DIS}}(\hat{k}) = \int_u \int_v \mathrm{e}^{\mathrm{j}\hat{k}\cdot\boldsymbol{r}_{\mathrm{im}'}} (\bar{I} - \hat{k}\hat{k}) R_i(u, v) \mathrm{d}u\, \mathrm{d}v \tag{12}$$

where $\boldsymbol{r}_{i,m'}$ represents the vector that extends from the sampling point to the disaggregation point, and $R_i(u, v)$ represents the testing function associated to the element i.

$V_{\mathrm{mj}}^{\mathrm{AGG}}(\hat{k})$ and $V_{\mathrm{mj}}^{\mathrm{DIS}}(\hat{k})$ only have θ and ϕ components ($V_{m'i}^{\mathrm{AGG}}(\hat{k}) = V_{\theta m'i}^{\mathrm{AGG}}\hat{\theta} + V_{\varphi m'i}^{\mathrm{AGG}}\hat{\varphi}$). Finally, the translation term between point m and m' is obtained by:

$$\tau_{\mathrm{mm}'}(\hat{k}, \boldsymbol{r}_{\mathrm{mm}'}) = \frac{jk}{4\pi} \sum_{l=0}^{L} j^l (2l+1) h_l^{(1)}(\mathrm{kr}_{\mathrm{mm}'}) P_l(\hat{r}_{\mathrm{mm}'} \cdot \hat{k}) \tag{13}$$

where $h_l^{(1)}(x)$ is a spherical Hankel function of the first kind and $P_l(x)$ is a Legendre polynomial. The integral in (10) is defined over all the directions of the unit sphere, but this integral can be truncated to a number of points N_k that provide a non-significant error, as explained in [4].

The previous expressions allow the use of an efficient iterative solver to address the system solution considering the coupling terms between distant elements. In this work, we have used the stabilised biconjugated gradient (BiCGSTAB) and the generalised minimal residue (GMRES) methods.

This numerical approach involves two separate phases [26]. First, at a pre-processing stage, we compute the contribution of every single j-subdomain to the aggregation (and disaggregation, analogously) to its first-level cube m. These contributions are the $V_{\mathrm{mj}}^{\mathrm{AGG}}$ and $V_{\mathrm{mj}}^{\mathrm{DIS}}$ terms in expressions (11) and (12), respectively, and they need to be applied to each subdomain of the scenario and to each direction sample of the unit sphere. The coupling terms between geometrically close elements are also obtained at this stage using expression (2).

The second stage applies to the equation solving process, in which the matrix–vector product computation is efficiently obtained. For any first-level cube of the problem, the addition of the aggregation term of the subdomains contained is calculated. If there is more than one level in the problem considered, the aggregation for any cube at level x is calculated by adding the aggregation terms of all the $x-1$ level cubes it contains. Once the aggregation terms are computed for every cube, they are translated to the corresponding cubes of the same level. Finally, the contributions are disaggregated from the high-level cubes to the first-level cubes, and subsequently to every subdomain contained. The contribution of geometrically close elements to the matrix–vector product is computed by using the stored terms.

Note that if in the election of the MoM basis and testing functions $T_j(u, v)$ and $R_i(u, v)$ the same functions are chosen, which is known as the Galerkin method, the aggregation and disaggregation terms are computed using the same expression.

The aggregation expression can be interpreted as the radiation pattern associated to each first-level cube, because it represents how the current over the basis functions contained in the cube radiates in a number of directions. If the aggregation and disaggregation terms are the same, it can be concluded that the way in which a set of basis functions radiate is the same as how it receives the external radiation. This is a consequence of the reciprocity theorem, which leads to relevant properties in the analysis of electromagnetic problems.

From a computational point of view, this election involves a significant reduction of the memory requirements because only one term associated to the MLFMA must be computed and stored and, as it will be shown in the next sections, its impact in the total memory requirements is remarkable. It is also important to note that the main burden in the use of GPUs is the limited amount of available memory, so the reduction obtained due to this choice is a relevant advantage.

The MLFMA has been widely used in combination with the MoM and its computational benefits have been illustrated in several publications [4–6].

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 MLFMA terms and solve the system equation using an iterative solver. These tasks still take up a considerable amount of time and memory, and, to reduce these, some parallel computation methods can be taken into account.

Here, we present one of these methods, which makes use of a recent innovation in high performance computing: CUDA [27]. CUDA is a technology that allows the use of GPUs for general purpose computing. GPUs have traditionally been used for graphics applications but can also be used to perform scientific calculations. GPUs are designed with a large number of processing devices, which makes them well suited to execute highly parallel scientific code. CUDA often requires a significant redesign of the algorithms, but provides considerable speedups. The CUDA programming paradigm and system architecture are defined to a greater extent in [28]. In this section, we present a brief description of the underlying framework.

CUDA allows to access hardware resources of the GPUs that basically are processing cores and memory. Cores are very simple processing units. They are physically grouped into clusters called streaming multiprocessors (SMs). A SM has several processing cores as well as registers and shared memory. A GPU typically has several SMs. The number of cores per SM depends on the architecture. The GPU memory is organised in a hierarchy, ranging from a relatively slow global memory, accessible to all the SMs, to very fast shared memories, visible 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 Fig. 1.

Due to the availability of the parallel resources of the GPU, it is particularly suitable for applications with a large amount of data parallelism, or those applications which repeatedly perform the same operations on large data sets. In order to use the resources made available by CUDA, the application must be designed with a special structure from the point of view of its programming. The application is organised as a succession of *kernel* calls. A *kernel* is a special kind of function or procedure, which runs on the GPU. It is executed in single instruction, multiple data (SIMD) mode, which means that multiple instances of the sequence of instructions that make up the function are executed in parallel. These instances are called *threads*. Each thread works over a different piece of the

© The Institution of Engineering and Technology 2019

17518733, 2019, 10, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/iet-map.2018.5568 by Universidad De Alcala, Wiley Online Library on [22/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

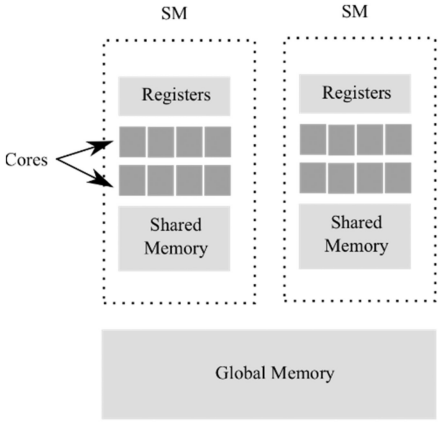

Fig. 1 *GPU architecture*

data, and runs on one core. As the cores work in parallel, the overall effect is that several portions of the data are processed at the same time.

Threads in a kernel are grouped into logical units called thread-blocks for easier handling inside the GPU. One thread-block is completely (i.e. 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. Since this number of threads may not be sufficient for regular applications, a kernel consists of several thread-blocks. Thread-blocks are organised in logical grids.

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). 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 one or two 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 of all, it is necessary to decide which parts will be executed on the GPU and which ones will run on the CPU. Typically, those parts exhibiting a fair amount of data parallelism will run on the GPU, and the rest will run on the CPU. The next step is to organise the parts assigned to the GPU into kernels. Sometimes it is possible to fit each part in one kernel, and sometimes it is necessary to divide it into several, depending on the logical structure of the application. Finally, the disposition of the data in the GPU memory must be decided, and the necessary data transfers must be included. The data must be transferred to 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 keep 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, depending 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 should be avoided whenever possible. The cost of a conditional branch that 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 [21] and made the necessary modifications so that the desired part of the code will run on the GPU. Most of this code has been transformed into custom-made CUDA kernels. The calculation of the matrix and multipole terms have been adapted in the pre-process stage, and the MLFMA matrix–vector product has also been modified. These parts of the code are the most decisive when analysing the computational cost of the method.

In the matrix computation stage, the parallelisation strategy was the definition of two CUDA kernels, each of then computing (3) and (4). These codes are applied to any coupling term between subdomains, so the theoretical number of threads that could run these kernels concurrently is limited to the number of pairs of coupled subdomains of the problem. Due to the use of the MLFMA, only the near-field coupling terms corresponding to the nearby subdomains are stored. Therefore, if we assume that each subdomain could have a reduced set of subdomains near it, there is an almost linear dependence between the coupling terms and the number of subdomains of the problem. As a consequence, the same linear dependence is obtained with respect to the size of the computed $\boldsymbol{Z}$ matrix.

```
→Copy geometrical description of the subdomains to GPU global memory
→Run a CUDA kernel to obtain geometrical parameters (one thread per subdomain)
→Run a CUDA kernel to obtain inductive term (3) (one thread per pair of coupled subdomains)
→Run a CUDA kernel to obtain capacitive term (4) (one thread per pair of coupled subdomains)
→Copy Z matrix to CPU memory
```

The next computation step is the calculation of the multipole terms. These terms are obtained by computing (11) and (12) for all the subdomains. A kernel implementing each equation has been created. There is no dependency between them, so, assuming only software limits, the maximum number of instances that run these kernels could be the number of subdomains of the problem.

```
→Run a CUDA kernel to obtain geometrical parameters (one thread per subdomain)
→Run a CUDA kernel to obtain aggregation term (17) (one thread per subdomain)
→Run a CUDA kernel to obtain disaggregation term (one thread per subdomain)
→Copy matrices to CPU memory
```

In the iterative process, a kernel has been created for each of the phases of the MLFMA matrix–vector product. Assuming a N-level problem, the phases are:

- *First-level aggregation:* The CUDA kernel code in this stage collects the contribution of each subdomain to the centre of its first-level parent cube. The aggregation term for cube m is computed as shown in (14):

$$V_m^{\mathrm{AGG}}(\hat{k}) = \sum_{j=1}^{\mathrm{Nj}} \int_u \int_v \mathrm{e}^{-\mathrm{j}\hat{k}\cdot \boldsymbol{r}_{\mathrm{jm}}} (\bar{I} - \hat{k}\hat{k}) T_j(u,v)\,\mathrm{d}u\,\mathrm{d}v = \sum_{j=1}^{\mathrm{Nj}} V_{\mathrm{mj}}^{\mathrm{AGG}}(\hat{k}) \tag{14}$$

  where N_j is the number of subdomains contained in the cube m.

  Every thread of the GPU computes the contribution of one subdomain for one direction of the spectrum and a single polarisation. Then, considering that the spectrum size at this level is 12×23, a maximum number of threads of Ns*12*23*2 can be calculated simultaneously. Note that several subdomains belong to the same first-level cube, so in order to prevent concurrency problems an atomic operation is required for their storage.
- *Aggregation from the second to the Nth level:* At each level, a parent-cube adds the contribution of its child-cubes from the previous level. A kernel is implemented to perform this part of the algorithm. This kernel computes the aggregation term of all the cubes in a single level, so it must be executed $N-1$ times for each matrix–vector product. Every thread of the GPU calculates, for any parent-cube in the level, the contribution to one direction of the spectrum and to one polarisation created by one of its

© The Institution of Engineering and Technology 2019

17518733, 2019, 10, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/iet-map.2018.5568 by Universidad De Alcala, Wiley Online Library on [22/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

child-cubes. The number of the spectrum directions increases when computing higher levels, while the number of empty cubes decreases.

It is important to note that at any level, there is no dependence between the data used for each thread. As in the previous phase, an atomic operation must be included to consider the computation of several threads that access to the same parent-cube. In addition, the computation of one level depends on the results of its previous level, so a synchronising procedure between levels needs to be established.

- *Translation and disaggregation from the Nth to the second level:* In this step, an i-level cube collects the contribution of the same level cubes (translation at i level) and the contribution of its parent at the $(i+1)$ level. Finally, it spreads that contribution to its child cubes at the $(i-1)$-level. Two kernels perform these actions until first-level cubes contributions are reached. The first kernel obtains the contribution for all the cubes in the level due to the cubes of the same level. Each thread in the GPU calculates the contribution to a single direction and a single polarisation created by a single cube of the level.

  The second kernel performs the distribution of the aggregation term of a parent-cube, computed in the previous kernel, to its child-cubes.

Both kernels handle the translation-disaggregation of a single level, so they must also be executed $N-1$ times at each matrix–vector product, and there is also dependency between levels, as in the previous phase.

- *First-level disaggregation:* Terms computing the interactions of distant elements for each subdomain are obtained by disaggregating the contributions of their first-level cube. A kernel has been developed for this phase, similarly to that of the first-level aggregation, and in consequence, the same conclusion can be applied.
- *Near-field matrix–vector product computation:* The kernel uses the stored $[\boldsymbol{Z}]$ matrix. Each GPU-thread computes the contribution to a single subdomain of all nearby subdomains. There is no dependence of the data involved, which leads to good performance in this computation.

Note that this scheme presents a computational burden. The main restriction in the use of GPUs to solve MoM-based moderate-size problems is its memory limitation. The memory size of the GPU is usually smaller than the memory of the CPU. When the traditional MoM is applied, the $[\boldsymbol{Z}]$ matrix is the term that consumes the most memory and, as a consequence, it conditions the electrical size of the problems that can be handled. The transfer of data from GPU to CPU memory is a solution for computing larger problems, but the performance is strongly affected, as shown in [14].

In the MLFMA, the most memory-expensive terms are those of aggregation and disaggregation, computed as shown in (11) and (12). These terms have a linear dependence with the number of unknowns, an also a linear dependence with the number of angular samples N_k at the first level defined by the MLFMA. When the size of the problem increases, the storage of these terms becomes the main burden for the computation using a GPU card.

The objective of this work is to overcome this restriction by introducing an algorithm to avoid the storage of these terms. It is based in a different manner of retaining the terms of the first-level aggregation and disaggregation, which reduces the need for MLFMA memory at the cost of increasing the number of operations to be performed. The computation of these operations, however, presents a good data parallelism, which makes them suitable to be carried out on GPU cards, obtaining efficient results.

As previously shown, the aggregation term is calculated using expression (11). For its numerical computation, a number of angular samples over the unit sphere are obtained:

$$V_{\theta m' i}^{\mathrm{AGG}}(N_k) = \int_u \int_v \mathrm{e}^{-\mathrm{j}\hat{k}_k \cdot \boldsymbol{r}_{\mathrm{jm}}} T_i(u,v)\mathrm{d}u\,\mathrm{d}v\hat{\theta} \tag{15}$$

where $\hat{k}_k$ is the unit vector over the unit sphere of the sample k. Using the Gaussian quadrature method described in (6), the integrals can be computed as a weighted addition of points, so (15) can be approximated as

$$V_{\theta m' i}^{\mathrm{AGG}}(N_k) = \sum_{\mathrm{pu}=1}^{\mathrm{Nu}} w_u \sum_{\mathrm{pv}=1}^{\mathrm{Nv}} w_v \mathrm{e}^{-\mathrm{j}\hat{k}_k \cdot \boldsymbol{r}(\boldsymbol{p}_u, \boldsymbol{p}_v)_{\mathrm{im}}} T_i(p_u, p_v)\hat{\theta} \tag{16}$$

where w_u and w_v are the coefficients of the Gauss quadrature method at points p_u and p_v, respectively, $T_i(p_u, p_v)$ is the value of the Basis Function of the subdomain i at point (p_u, p_v), and $\boldsymbol{r}_i(p_u, p_v)$ is the position vector of that point. Since the only dependence with the sample k is on the exponential, we can define

$$Vs_{\theta m' i}^{\mathrm{AGG}}(Nu, Nv) = \sum_{pu=1}^{Nu} w_u \sum_{pv=1}^{Nv} w_v T_i(p_u, p_v)\hat{\theta} \tag{17}$$

Note that $Vs_{\theta m' i}^{\mathrm{AGG}}(Nu, Nv)$ has no dependence on samples k. As $N_u{*}N_v$ is smaller than N_k, the memory needed to store this term is reduced, and the computational burden is overcome. For this purpose, the Vs term is stored and applied in the iterative process as shown in (18), instead of using (14).

$$\begin{aligned} V_{\theta m}^{\mathrm{AGG}}(N_k) &= \sum_{i=1}^{Ni} \sum_{pu=1}^{Nu} w_u \sum_{pv=1}^{Nv} w_v \mathrm{e}^{-\mathrm{j}\hat{k}_k \cdot \boldsymbol{r}(\boldsymbol{p}_u, \boldsymbol{p}_v)_{im}} T_i(p_u, p_v)\hat{\theta} \\ &= \sum_{i=1}^{Ni} \sum_{pu=1}^{Nu} \sum_{pv=1}^{Nv} \mathrm{e}^{-\mathrm{j}\hat{k}_k \cdot \boldsymbol{r}(\boldsymbol{p}_u, \boldsymbol{p}_v)_{im}} Vs_{\theta m' i}^{\mathrm{AGG}}(p_u, p_v)\hat{\theta} \end{aligned} \tag{18}$$

According to our experience, a value of 2 for N_u and N_v is sufficient to render an accurate computation of the coupling, since the MLFMA is only applied to compute coupling terms between distant elements. In addition, when choosing a first-level cube size over $\lambda/4$, the number of angular samples N_k that provides accurate results should be about two hundred, so the memory reduction with the modification presented on $V_{m'i}^{\mathrm{AGG}}$ and $V_{m'i}^{\mathrm{DIS}}$ is ∼50 times, and allows the solution of electrically larger problems.

The use of this technique presents an inconvenient: the first-level aggregation and disaggregation stages in the iterative process must perform more operations, since they need to calculate the operations shown in (18) instead of (11). The consequence is that for an only-thread code, the CPU-time spent in these steps increases, which slows down the resolution of the problem. Fortunately, the kind of operations shown in (18) are suitable to be accelerated using a machine with many threads that run simple operations, and that is the case of the use of GPUs.

We can conclude that the computational algorithm presented in this section reduces the memory need of the MLFMA method (increasing the electrical size of the problems that can be solved). The CPU-time increment associated with the algorithm in the solution process is compensated by using parallelised code executed on the GPUs.

There are other implementations of the MLFMA designed for GPUs. In [16], the authors define the OpenMP-CUDA-MLFMA method. The parallelisation for a GPU device with the application of the OpenMP paradigm to be executed in the CPU is the same simulator. In addition, they describe a strategy to adapt the code to multi-GPU platforms. All of these strategies can be combined with the presented approach to increment its performance.

The main difference between both strategies is that due to the application of the computational algorithm shown here, our approach reduces the memory requirements of the method without CPU-time penalty, which allows performing matrix–vector products without transferring data between CPU and GPU. In [16], the option to solve large problems relies on the use of several GPUs, due to the large amount of memory needed to store the matrices. This can be avoided with the approach presented in this work due to the reduction of the memory described, which allows the solution using a single GPU.

© The Institution of Engineering and Technology 2019

17518733, 2019, 10, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/iet-map.2018.5568 by Universidad De Alcala, Wiley Online Library on [22/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

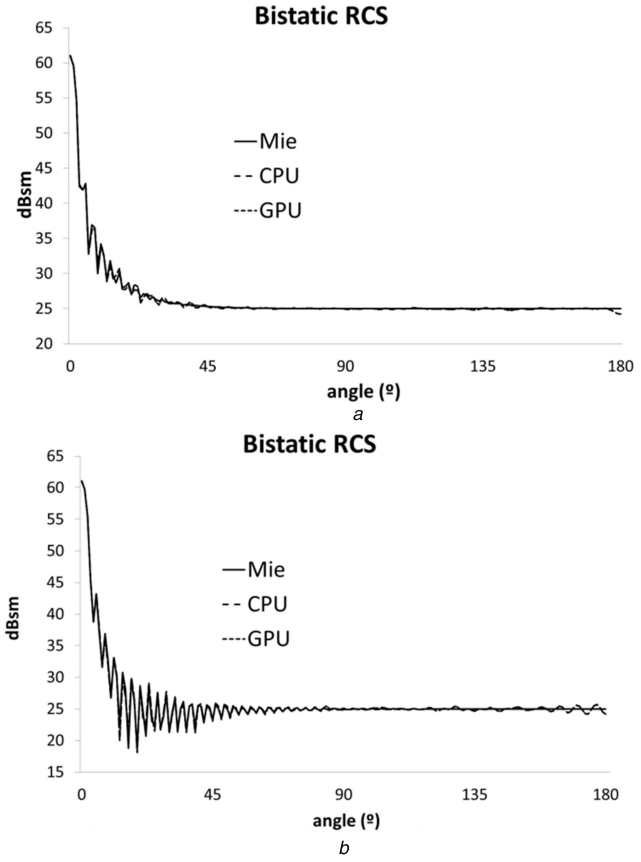

Fig. 2 *Bistatic RCS of a sphere*
(a) H-plane, ***(b)*** E-plane

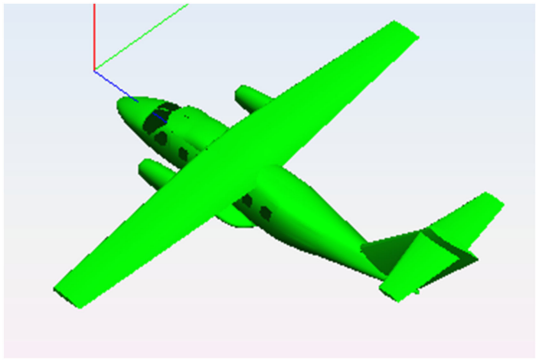

Fig. 3 *Airplane geometry*

With respect to the strategy for the allocation of the data in the architecture of the GPU and the implementation of the kernels, both strategies are similar. As the memory requirements are reduced in our approach, most data are stored in the global memory of the device instead of using pinned memory, which prevents the transfer of data between the GPU and the CPU.

4 Results

This section is organised as follows: first, we validate the version of the application which uses the GPU by comparing its results with those of the version that runs only on the CPU for different test cases, proving that the results obtained by using the GPU are accurate enough to be accepted. Next, we measure the speedup obtained by using the GPU at any of the stages: the calculation of the MLFMA terms and coupling matrix at pre-process, and the computation of a single iteration in the solution process of the system of equations.

The computer considered for the validation has an Intel i7 processor with 8 GB of RAM, and a TESLA C2075 GPU with 6 GB of RAM. It runs a Windows 7, 64-bit license. We use the 3.1 CUDA toolkit. Currently, newer GPU cards are available in the market with an improved architecture, which also contain more parallel blocks and RAM. Recently, cards with 24 GB of RAM have been made available. In addition, the CUDA 9.0 version is currently available, which is suitable for those more powerful cards. It would be interesting to test the algorithm presented with these cards and toolkit, although the authors do not have these tools provided by NVIDIA at their disposal at the moment. Since the main contribution of this paper is the algorithm that allows the computation of large test cases on the GPUs, we can expect that with the hardware improvements of the new cards the speedup increases over that shown in this section.

4.1 Validation

The first test case is the computation of the bistatic Radar cross-section of a 10λ radius sphere. A comparison between the values of analytical, CPU-code and GPU-code version is shown in Fig. 2. The incident angle is $\varphi = 0°$ and $\theta = 180°$ and we consider a θ-cut at $\varphi = 0°$ plane. The number of low-level basis functions obtained has been 251,956, and the numerical results have been obtained with a residual error of 0.01 in the iterative solver.

In order to estimate the differences between the approaches, we have computed the error between the CPU and GPU versions with respect to the analytical values, following expression (19):

$$\text{Error}_{\text{version}} = \sum_{i=1}^{N} \frac{|\text{RCS}i_{\text{MIE}} - \text{RCS}i_{\text{version}}|}{|\text{RCS}_{\text{Mie}}|} \tag{19}$$

where N is the number of observation directions, and $\text{RCS}i_{\text{MIE}}$ and $\text{RCS}i_{\text{version}}$ are the RCS value at the i direction when applying the analytical solution and our algorithm, respectively. $\text{RCS}i_{\text{version}}$ is calculated with both CPU-code and GPU-code. $\text{Error}_{\text{version}}$ in the CPU-code version has been 0.011016 and 0.005110 for the E-plane and H-plane series, respectively, while in the GPU-code the error has been 0.011018 and 0.005113. When comparing numerical results of CPU-code version with GPU-code version, we can see that the error is minimal, and given by the numerical errors due to finite precision of the data represented in simple precision. Therefore, the results clearly validate the adaptation of the algorithm presented to CUDA

The next geometry considered is the airplane model shown in Fig. 3. We have considered three frequencies to show the variation of the speedup with the electrical size of the problem. These frequencies are 225, 500 and 1450 MHz, that give rise to problems containing 19,693, 85,373, and 694,142 unknowns, respectively.

We have computed the monostatic radar cross-section (RCS) and compared the results obtained using the GPU and the CPU algorithms. Figs. 4*a–c* show the RCS for the θ-polarisation and for the $\phi = 180$ angular cut with θ ranging from 0 to 180° in steps of 1° for each frequency. As in the previous test cases, for all the incident directions, the results have been practically identical.

Another test case presented for validation is shown in Fig. 5*a*. It is another airplane model, and its monostatic RCS is computed for the theta polarisation, considering the $\phi = 90$ angular cut and θ from 0 to 180° in steps of one degree at 1125 MHz. The results obtained are shown in Fig. 5*b*.

The next test case presented for validation consists of the computation of the monostatic RCS of a 9 inch ogive at a frequency of 15 GHz, considering the θ–θ polarisation. The ogive (Fig. 6*a*) is a symmetrical case that has a half-angle of 22.62° and a maximum radius of 1 inch. The results obtained using the CPU and GPU code versions are compared with measurements in Fig. 6*b*, which shows a good agreement.

The mathematical description of this geometry is

$$\begin{cases} -4.5 \le x \le 4.5 \\ -\pi \le \phi \le \pi \end{cases}$$

© The Institution of Engineering and Technology 2019

17518733, 2019, 10, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/iet-map.2018.5568 by Universidad De Alcala, Wiley Online Library on [22/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

$$f(x) = \sqrt{1 - \left(\frac{x}{4.5}\sin(22.62°)\right)^2} - \cos(22.62°)$$
$$y = \frac{f(x) \cdot \cos(\phi)}{1 - \cos(22.62°)} \quad (20)$$
$$z = \frac{f(x) \cdot \sin(\phi)}{1 - \cos(22.62°)}$$

The last test case considered in this work is the monostatic RCS of the Cobra geometry at a frequency of 17 GHz, described in Fig. 7*a*. The analysis has been performed for the θ-polarisation in the $\phi = 0$ angular cut and with θ ranging from 0 to 180. A comparison of the results obtained by the CPU and GPU versions with measurements is illustrated in Fig. 7*b*. Very good agreement is shown between the numerical methods and the measurements.

4.2 Computational analysis

The next step in the validation of the proposed approach is the comparison of execution times for the CPU and the GPU approaches. First, we compare the execution times for the test case shown in Fig. 3. On one hand, it is interesting to know how much faster the GPU version is, compared to the CPU version, when calculating the MoM coupling matrix and the MLFMA data, and when solving the linear system of equations using the MLFMA. On the other hand, the memory requirements need to be compared as well. Both versions have been fully implemented by the authors without the need of external linear algebra libraries. As a reference for the comparisons, we consider the time spent by the CPU of the Intel i7 processor with 8 GB of RAM, so, a speedup of 10 means that when using TESLA C2075 GPU with 6 GB of RAM the time is only tenth part of the CPU time. Table 2 shows for both methods the time spent to calculate the $[\boldsymbol{Z}]$ matrix and the MLFMA data. The results shown correspond to the airplane at three different frequencies. Similar results have been obtained for other geometries. The results for the CPU version are obtained using a single CPU thread, since we need a reference value for comparison purposes. As expected, the improvement provided by the GPU grows with the size of the problem. There are several tasks that require an amount of time that is almost independent of the problem size, such as memory initialisation, transferring geometry data to the GPU etc. These tasks have a proportionately greater influence on the total execution time for smaller problems. As problems get larger, their impact is less noticeable, and the speedup increases. The pre-processing calculations are suitable to be performed in parallel, where the calculation of each element is independent from the rest. This is not the case for the subsequent steps of the algorithm, since the process to solve the system of equations has a lower degree of parallelism.

As mentioned above, the MLFMA should be applied with an iterative solver. Table 3 shows the CPU-time spent in one iteration when solving the problem. The analysis is performed for the three frequencies of the first test case. We obtain a remarkable speed-up because with the presented algorithm all the data can be stored in

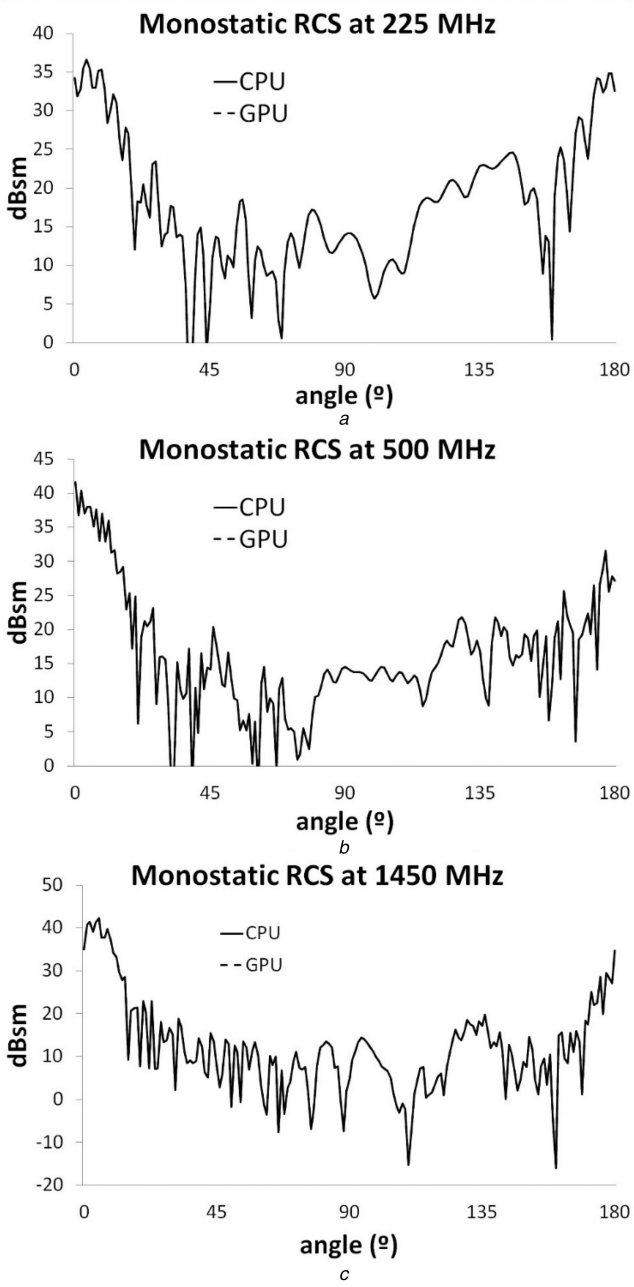

Fig. 4 *Monostatic RCS for the geometry of the airplane of Fig. 3 at* ***(a)*** 225 MHz, ***(b)*** 500 MHz, ***(c)*** 1450 MHz

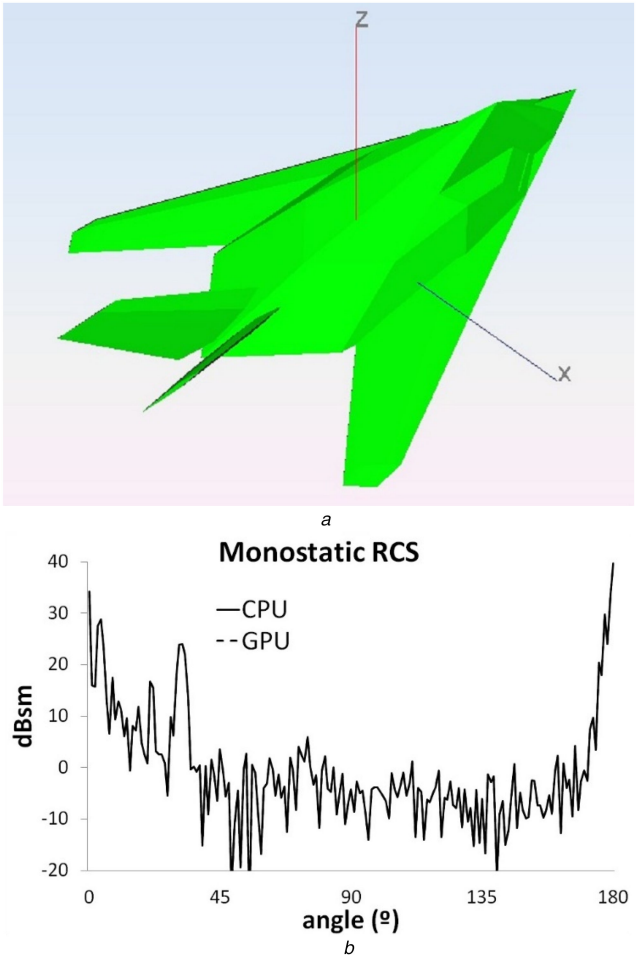

Fig. 5 *Test case*
(a) Airplane geometry and ***(b)*** Monostatic RCS at 1125 MHz for the geometry of airplane

© The Institution of Engineering and Technology 2019

17518733, 2019, 10, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/iet-map.2018.5568 by Universidad De Alcala, Wiley Online Library on [22/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

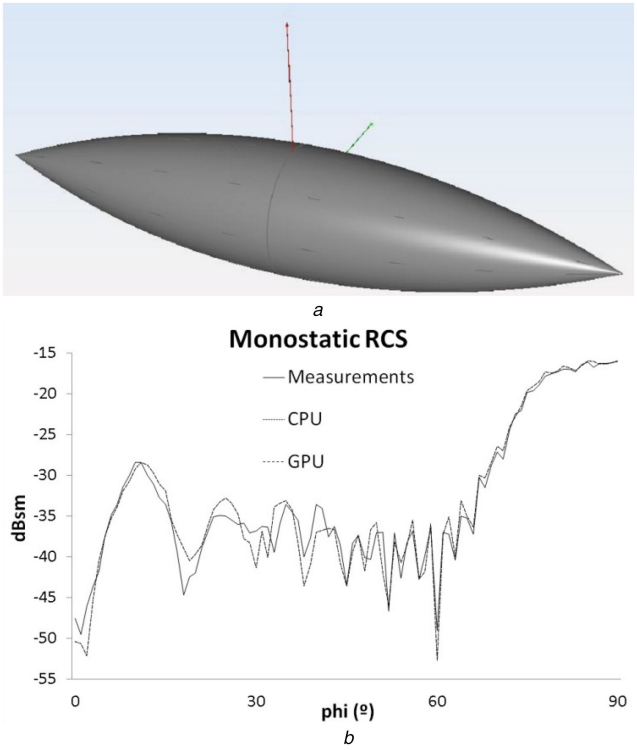

Fig. 6. *Test case*
(a) Geometry of the ogive, ***(b)*** Monostatic RCS of the ogive at 15 GHz

the GPU memory ($\boldsymbol{Z}$ matrix and Vs terms) and, for one iteration, only the input vector is transferred from the CPU to the GPU memory, and only the result vector is transferred from GPU to CPU memory.

Several conclusions can be obtained from these results. First, the improvement in this case is lower than in the previous one. This is because the dependencies within the MLFMA algorithm. For example, the x-level aggregation cannot be computed until the $(x-1)$-level aggregation has been performed. As a result, the total improvement is reduced.

On the other hand, we can see that the improvement is reduced as the problem size grows. To find the reason for this, we perform a detailed analysis of the performance of the different kernels of the iterative solution process for the test case 1. The results of this analysis are shown in Tables 4–6.

We can conclude that there are significant differences between the performances of the different kernels. The first-level aggregation, the second to N-level aggregation and first-level disaggregation kernels have good improvements while those for the rest are quite poor. This is because there are very few elements in these kernels to operate with (the number of cubes at upper levels is small) and, as a consequence, the ratio between the operations to be performed by the kernels and the access to GPU memory leads to this lack of performance. As the problem size grows, the number of levels also increases, and the less efficient kernels become more important, so the improvement is reduced. However, even considering this reduction, the use of the presented algorithm offers very interesting results for large problems compared to CPU versions.

Finally, a comparison of the storage requirements of both versions is presented for the test case shown in Fig. 3. Table 7 shows the memory spent for the CPU version. Note that the more memory consuming terms of the code are the $[\boldsymbol{Z}]$ matrix, and aggregation and disaggregation ones. The geometrical description and other MLFMA data complete the total memory spent.

The memory requirements of the GPU-version code are shown in Table 8. We can see that we need the same memory to store the $\boldsymbol{Z}$ matrix, but there is a huge reduction of the aggregation and disaggregation terms, due to the application of the presented approach. This allows the use of the GPUs for the analysis of cases with moderate electromagnetic size.

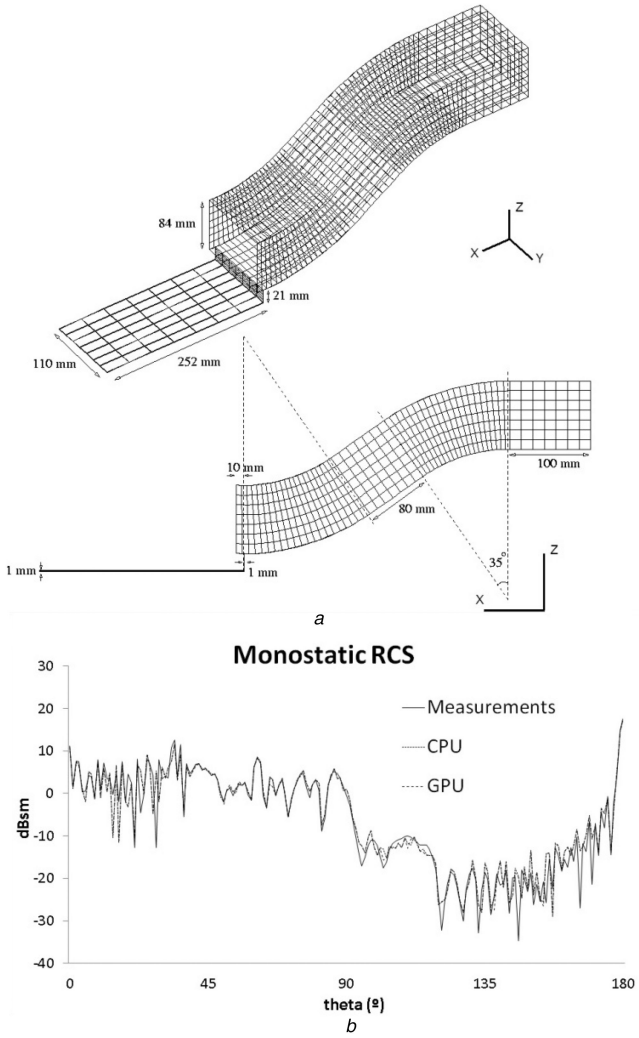

Fig. 7. *Test case*
(a) Geometry of the cobra cavity, ***(b)*** Monostatic RCS of the Cobra cavity at 17 GHz

Table 2 Time in seconds for the pre-processing computation for GPU and CPU based methods

	GPU	CPU	Speedup GPU/CPU
19,693 unknowns	2.3	329.5	143.26
85,373 unknowns	8.43	1346	159.66
694,142 unknowns	86.75	16,472	189.87

Table 3 Time in seconds for the iterative process computation for GPU and CPU based methods

	GPU	CPU	Speedup GPU/CPU
19,693 unknowns	22	2160	98.18
85,373 unknowns	72	6480	90
694,142 unknowns	240	20,160	84

In the GPU-code version, the aggregation terms are computed and stored in a pre-processing stage as described in (17), instead of applying expression (11), which is used in the CPU-code version. This leads to the difference in the memory requirements between both versions. Next, in the iterative stage, the first-level aggregation in the GPU-code version is obtained as described in (18) instead of (14) for each matrix–vector product performed by the iterative solver, which introduces more operations to execute. Since these operations are very suitable to be developed in the GPU, both memory and CPU-time are reduced using this approach on a GPU.

© The Institution of Engineering and Technology 2019

17518733, 2019, 10, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/iet-map.2018.5568 by Universidad De Alcala, Wiley Online Library on [22/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Table 4 Time in seconds for different steps of the iterative process

	GPU	CPU	Speedup GPU/CPU
first-level aggregation	0.031	7.31	235.80
second to *N* level aggregation	0.015	2.44	163.28
n to 2 level translation-disaggregation	0.057	0.28	4.91
first-level translation	0.027	0.078	2.88
first-level disaggregation	0.036	3.728	103.55

Frequency 225 MHz.

Table 5 Time in seconds for different steps of the iterative process

	GPU	CPU	Speedup GPU/CPU
first-level aggregation	0.14	31.09	222.07
second to *N* level aggregation	0.064	11.07	172.96
n to 2 level translation-disaggregation	0.2494	1.16	4.65
first-level translation	0.113	0.296	2.61
first-level disaggregation	0.151	15.33	101.52

Frequency 500 MHz.

Table 6 Time in seconds for different steps of the iterative process

	GPU	CPU	Speedup GPU/CPU
first-level aggregation	1.83	326.99	178.68
second to *N* level aggregation	1.17	183.20	156.58
n to 2 level translation-disaggregation	7.01	24.07	3.43
first-level translation	3.54	7.44	2.10
first-level disaggregation	2.01	163.62	81.40

Frequency 1450 MHz.

Table 7 Memory requirements in Mbytes CPU method

	Z matrix	Aggregation and disaggregation terms	Total memory
19,693 unknowns	31.4	165.1	262.2
85,373 unknowns	104.6	719.2	1006.9
694,142 unknowns	691.7	5846.7	7997.4

Table 8 Memory requirements in Mbytes GPU method

	Z matrix	Aggregation and disaggregation terms	Total memory
19,693 unknowns	31.4	3.6	102.1
85,373 unknowns	104.6	15.7	305.4
694,142 unknowns	691.7	127.1	2281.4

It is also important to note that similar conclusions can be obtained for the analysis of the other test cases presented in the validation section.

5 Conclusions

We have presented a procedure to apply the MLFMA that takes advantage of the current GPUs and overcomes one of the most important limitations suffered by this novel computing system, namely, the limitation of GPU memory. We validate the use of a GPU in terms of accuracy of the results, and we have shown that the speedups obtained are remarkable, which agrees with other results available in the literature.

In order to deal with large geometries, we have adapted the existing MLFMA to reduce the memory requirement of the algorithm. This modification leads to an increased computational cost in the execution of the iterative process, which is overcome due to the parallel nature of the algorithm, amenable to be executed on GPUs. In future works we will try to solve this problem.

6 Acknowledgments

This work has been supported in part by the Spanish Ministerio de Economía y Competividad, Projects TEC 2013-46587-R, PTQ-12-05099 and PTQ-14-07060. The authors declare that there is no conflict of interest regarding the publication of this paper.

7 References

[1] Harrington, R.F.: '*Field computation by moment methods*' (McMillan, New York, 1968)

[2] Prakash, V.V.S., Mittra, R.: 'Characteristic basis function method: a new technique for efficient solution of method of moments matrix equation', *Microw. Opt. Technol. Lett.*, 2003, **36**, (2), pp. 95–100

[3] Engheta, N., Murphy, W.D., Rokhlin, V., *et al.*: 'The fast multipole method (FMM) for electromagnetic scattering problems', *IEEE Trans. Antennas Propag.*, 1992, **40**, (6), pp. 634–641

[4] Chew, W.C., Jin, J., Michielssen, E., Song, J. (Eds.): '*Fast and efficient algorithms in computational electromagnetics*' (Artech House, Norwood, MA USA, 2001)

[5] Ding, D.Z., Fan, Z.H., Tao, S.F., *et al.*: 'Complex source beam method for EM scattering from PEC objects', *IEEE Antennas Wirel. Propag. Lett.*, 2015, **14**, pp. 346–349

[6] Ding, D.Z., Chen, G.S., Chen, R., *et al.*: 'An efficient algorithm for surface integral equation based on meshfree scheme', *IEEE Antennas Wirel. Propag. Lett.*, 2014, **13**, pp. 1541–1544

[7] Ludick, D.J., Davidson, D.B.: 'Investigating efficient parallelization techniques for the characteristic basis function method (CBFM)'. 2009 Int. Conf. Electromagnetics in Advanced Applications (ICEAA 09), Torino, Italy, 2009

[8] García, E., Lozano, L., Algar, M.J., *et al.*: ' 'A study of the efficiency of the parallelization of a high frequency electromagnetic approach for the computation of radiation and scattering considering multiple bounces', *Comput. Phys. Commun.*, 2013, **184**, (1), pp. 45–50

[9] Lezar, E., Davidson, D.B.: 'GPU-accelerated method of moments by example: monostatic scattering', *IEEE Trans. Antennas Propag.*, 2010, **52**, (6), pp. 120–135

[10] Zoric, D.P., Olcan, D.I., Kolundzija, B.M.: 'GPU accelerated computation of radar cross sections with multiple excitations'. 2013 European Conf. Antennas and Propagation (EUCAP 2013), 2013

[11] Michiels, B., Fostier, J., Bogaert, I., *et al.*: 'Full-wave simulations of electromagnetic scattering problems with billions of unknowns', *IEEE Trans. Antennas Propag.*, 2015, **63**, (2), pp. 796–799

[12] Gurel, L., Ergul, O.: 'Hierarchical parallelization of the multilevel fast multipole algorithm (MLFMA)', *Proc. IEEE*, 2013, **101**, (2), pp. 332–341

[13] Pan, X.M., Pi, W.C., Yang, M.L., *et al.*: 'Solving problems with over one billion unknowns by the MLFMA', *IEEE Trans. Antennas Propag.*, 2012, **60**, (5), pp. 2571–2574

[14] Lezar, E., Davidson, D.B.: 'GPU acceleration of method of moments matrix assembly using Rao-Wilton-Glisson basis functions'. 2010 Int. Conf. Electronics and Information Engineering (ICEIE 2010), Kyoto, Japan, 2010

[15] Zoric, D.P., Olcan, D.I., Kolundzija, B.M.: 'Solving electrically large EM problems by using out-of-core solver accelerated with multiple graphical processing units'. 2011 IEEE Int. Symp. Antennas and Propagation and USNC/URSI National Radio Science Meeting (APSURSI 2011), Rome, Italy, 2011

[16] Peng, S., Nie, Z.: 'Acceleration of the method of moments calculations by using graphics processing units', *IEEE Trans. Antennas Propag.*, 2008, **56**, (7), pp. 2130–2133

[17] Topa, T., Karwowski, A., Noga, A.: 'Using GPU with CUDA to accelerate MoM-based electromagnetic simulation of wire-grid models', *IEEE Antennas Wirel. Propag. Lett.*, 2011, **10**, pp. 342–345

[18] Guan, J., Yan, S., Ming Jin, J.: 'An accurate and efficient finite element-boundary integral method with GPU acceleration for 3-D electromagnetic analysis', *IEEE Trans. Antennas Propag.*, 2014, **62**, (12), pp. 6325–6336

[19] Guan, J., Yan, S., Jin, J.-M.: 'An OpenMP-CUDA implementation of multilevel fast multipole algorithm for electromagnetic simulation on multi-GPU computing systems', *IEEE Trans. Antennas Propag.*, 2013, **61**, (7), pp. 3607–3616

[20] Xu, K., Ding, D.Z., Fan, Z.H., *et al.*: 'Multilevel fast multipole algorithm enhanced by GPU parallel technique for electromagnetic scattering problems', *Microw. Opt. Technol. Lett.*, 2010, **52**, (3), pp. 502–507

[21] Rivas, F., Valle, L., Cátedra, M.F.: 'A moment method formulation for the analysis of wire antennas attached to arbitrary conducting bodies defined by

© The Institution of Engineering and Technology 2019

17518733, 2019, 10, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/iet-map.2018.5568 by Universidad De Alcala, Wiley Online Library on [22/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

parametric surfaces', *Appl. Comput. Electromagn. Soc. J.*, 1996, **11**, (2), pp. 32–39
[22]]Valle, L., Rivas, F., Cátedra, M.F.: 'A moment method approach using frequency independent parametric meshes', *IEEE Trans. Antennas Propag.*, 1997, **45**, (10), pp. 1567–1568
[23] Cátedra, M.F., Rivas, F., Valle, L.: 'Combining the moment method with geometrical modelling by NURBS surfaces and Bezier patches', *IEEE Trans. Antennas Propag.*, 1994, **42**, (3), pp. 373–381
[24] Press, W.H., Flannery, B.P., Teukolsky, S.A*., et al.*: '*Numerical recipes in FORTRAN: the art of scientific computing*' (Cambridge University Press, Cambridge, 1992, 2nd edn.)
[25] Velamparambil, S., Chew, W.C., Song, J.: '10 million unknowns: is it that big?', *IEEE Antennas Propag. Mag.*, 2003, **45**, (2), pp. 43–58
[26] González, I., García, E., Sáez De Adana, F*., et al.*: 'Monurbs: a parallelized fast multipole multilevel code for analysing complex bodies modelled by NURBS surfaces', *Appl. Comput. Electromagn. Soc. J.*, 2008, **23**, (2), pp. 134–142
[27] Nickolls, J., Buck, I., Garland, M*., et al.*: 'Scalable parallel programming', *ACM. Queue.*, 2008, **6**, (2), pp. 40–53
[28] NVIDIA Corporation: '*CUDA programming guide*' (NVIDIA, Santa Clara, CA, 2013). Available at http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

© The Institution of Engineering and Technology 2019

17518733, 2019, 10, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/iet-map.2018.5568 by Universidad De Alcala, Wiley Online Library on [22/01/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License