

## Chapter 2

# Parallelization Strategies for the Characteristic Basis Function Method

Eliseo García, Juan I. Pérez, José A. de Frutos, Felipe Cátedra,  
and Raj Mittra

### 2.1 Implementation of Characteristic Basis Function

In recent years, NewFasant has emerged as one of the commercial tools which has implemented the CBFM [1–5] to improve the traditional MoM for the analysis of complex electromagnetic problems in a efficient manner. NewFasant includes both high-frequency and rigorous techniques, so as the combination with the Multi Level Fast Multipole Algorithm (MLFMA) [6, 7] to take the computational advantages of these methods.

We begin by describing how the Characteristic Basis Functions (CBFs) is incorporated in this tool. In previous works on CEM techniques developed by the authors, each block was identified with each one of the Non-Uniform Rational Bi-Splines (NURBS) [8] patches that define the body under analysis. However, this block definition criterion may not be well suited for achieving the best performance of the method since the surfaces are delimited in concordance with the geometrical design parameters, and some of the resulting patches can be so large as to notably increase the CPU-time required by the numerical tool.

---

E. García (✉) • J.I. Pérez • J.A. de Frutos  
Automatics Department, University of Alcalá, Madrid, Alcalá de Henares, Spain  
e-mail: [eliseo.garcia@uah.es](mailto:eliseo.garcia@uah.es); [nacho.perez@uah.es](mailto:nacho.perez@uah.es); [jose.frutos@uah.es](mailto:jose.frutos@uah.es)

F. Cátedra  
Computer Science Department, University of Alcalá, Madrid, Alcalá de Henares, Spain  
e-mail: [felipe.catedra@uah.es](mailto:felipe.catedra@uah.es)

R. Mittra  
Electromagnetic Communication Laboratory, 319 EEE, Pennsylvania State University,  
University Park, 16802 PA, USA  
Electrical Engineering Department, Pennsylvania State University, University Park,  
16802, PA, USA  
e-mail: [mittra@engr.psu.edu](mailto:mittra@engr.psu.edu)

In the conventional application of the MLFMA methodology, the original geometry is divided into regions, generating a multilevel 3D grid. Consequently, in order to obtain a homogeneous block size which is independent herein of the geometrical size of the surfaces, a new block partitioning method is proposed, defining every block in terms of the MLFMA regions. So, a region could include several blocks, and each one of them can belong to different surfaces. On the other hand, a surface that is defined over several regions would be also described using several blocks. The electrical continuity between different blocks belonging to the same original surface is assured via the use of the connections that comprise of the low-level basis functions.

The first step in the procedure is to define an electrical block size. In our experience, the best computational results are obtained when this size is the MLFMM group size (first level cube size). Once the problem has been discretized with the low-level basis functions, these are grouped into blocks. Each of the CBFs in this above blocks is then assigned to the region which contains the block. As a result, there will be a number of CBFs associated to each region.

Once the size of the CBFs is defined, we proceed to generate of them by using the method described below. Although there are several options available to us, we use the Singular Value Decomposition of the current induced by a spectrum of planar waves. These currents could be computed by using one of the several available techniques, e.g. the Physic Optics (PO) [9] or the Method of Moments (MoM). PO can obtain the desired induced current with little investment in the computational cost and it is best suited for smooth problems. However, MoM is useful for generating accurate result for blocks with complex geometries. In view of this, we suggest using PO for large and smooth blocks, and the MoM for the remainder of the blocks.

## 2.2 Parallelization of a CBFM-MLFMA Code

It is always desirable to set up the analysis of electromagnetic problems such that it achieves a good accuracy without consuming an excessive amount of computation time. Although, a simple approach to achieving this goal is to utilize a powerful machine, it becomes an increasingly expensive proposition to obtain digital equipment with the desired features when solving large and complex electromagnetic problems.

A commonly used strategy for achieving this aim is to use machines with multiple nodes. The strategy pursued is to divide the total computing task between these nodes, so that the computing task per node is lower than if were to solve it on a single node.

When implementing the parallelization paradigm with a given code, we can divide the machines into two categories: Shared-memory and distributed-memory machines. What differentiates these two types is how they exchange data between different nodes which perform the analysis. In the first type of machine, it is normally done through read/write memory operations, while in the second by

sending/receiving messages through a communication network. Since the communication networks are almost always slower than memory accesses, the shared-memory machines typically provide a superior performance over the distributed-memory counterparts. However, the problem is that these machines are usually more expensive and less scalable. As a consequence of this, in recent years we have witnessed increasing use of computer clusters for the analysis of electromagnetic problems.

Once the machine has been selected, it is necessary to modify the code to be run over multiple nodes, taking into account the characteristics of the machine. The most widespread solution for the parallelization of the code is to implement the code functions that allow the division of tasks between nodes and their intercommunications. These functions are defined according to the functionality of the language used and are compiled by following a programming paradigm. The most widespread paradigm for parallelization is the Message Passing Interface (MPI) [10].

## 2.3 An MPI Strategy

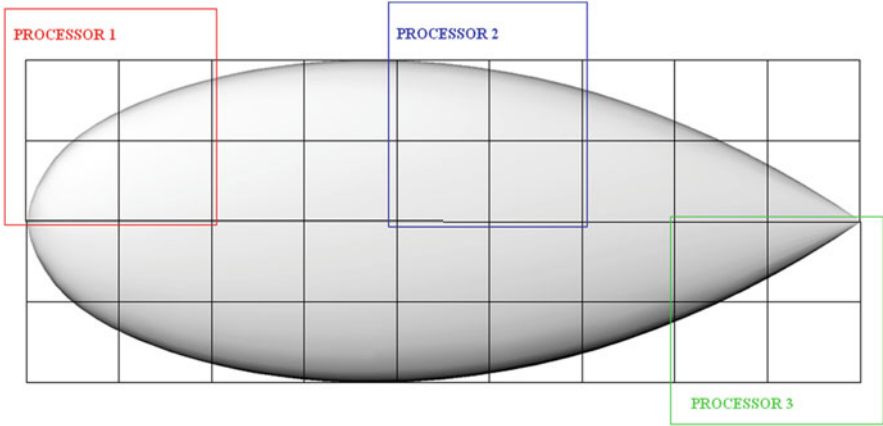
MPI provides the objects and procedures to divide the load work and to communicate between the nodes, to be added to a general programming language. It was created for use in distributed memory machines, but it can also be used in shared memory ones. Next, we describe the application of this paradigm in the CBFM-MLFMA algorithm which runs on several nodes to achieve the computational efficiency.

Generally, the main goal of paralleling a code is to ensure that all nodes are running the code at all time during the life of the application. If a node is sitting idle, it is considered that the node is wasting its time.

The main reason why a node is not running its code is because it is expecting data or synchronization from another process (note that usually it will be waiting for the arrival of a data from the network, which is slow) or because it has completed its work while the rest of the nodes are continuing to work on their tasks. Consequently to achieve a maximum performance of the parallel application, it is good practice to design it to satisfy the following two objectives. The first of these to minimize communications, and, secondly, balance the computation time of all nodes by ensuring the nodes have the same workload (load balancing). Hence, it is important to know, in each application, the work load to be distributed among the nodes.

### 2.3.1 *Distribution Unit*

This section describes how the goals discussed above can be achieved in a CBFM-MLFMA code. We begin by discussing the issues of load distribution. In this work, we consider as “level 1”, the level in which the group size of the FMM is the smallest, and “maximum level”, the level in which an only group includes all the geometry.



**Fig. 2.1** Block distribution among the nodes

In the MLFMA algorithm, the matrix–vector products are computed by using the aggregation, translation and disaggregation scheme. In the first step of the aggregation and disaggregation process, information from each basis function multipole terms is expanded into the lower level groups. From our experience, the time spent on this process is smaller than that needed in the translation procedure.

In the translation process, a multilevel scheme is considered by using groups, and never basis functions. To balance the work among the computer nodes, we distribute all the groups among the nodes. This distribution is made in every level by taking into account the groups of each level; hence, we can conclude that the distribution unit of this application is the group.

In the pre-processing stage all the regions are distributed among the nodes for (Fig. 2.1). Thus, each node will have a set of regions, and it should fetch the influence of all the regions over them. Some part of these it could be computed locally, but others would be received from other nodes [11, 12].

The phases in the computation for every node are as follows:

- Step 1: Each node computes the aggregation terms for all its regions in the first level.
- Step 2: The aggregation terms are computed for its FMM cubes in the rest of levels
- Step 3: The node sends its aggregated-cubes to other node that need them.
- Step 4: It receives the aggregated cubes from other nodes.
- Step 5: The translation process is implemented by using its own and other-nodes aggregated-cubes.
- Step 6: The node disaggregates the translated coupling terms computed earlier.
- Step 7: This final step deals with the evaluation of the rigorous terms of the coupling matrix that are not amenable for evaluation by using the MLFMA.

After following all of these steps, every node would have the influence of all the geometry over its own regions and, as a consequence, all the coupling terms would be accounted.

Once the distribution unit has been set, the lower levels will have many groups, when we are considering the case of intermediate electrical size, and the number of groups will be relatively small at higher levels. As a result, it is very difficult to find a balanced distribution of groups at these higher levels, so this unit is not best suited for this case.

If we take a close look at the algorithm, we see that when we increase the level, the number of groups decreases, however, their size increases and, hence, so does the number of angular samples handled by the MLFMA algorithm. Thus, when we increase the level, we obtain a large number of samples, and a balanced distribution of these between the nodes can be achieved relatively easily. This implies that at these levels all nodes must compute a set of samples from all the groups of that level.

Consequently, the distribution unit will be groups at lower levels, while the MLFMA angular samples at the higher ones. Using these units, a similar load work and computation time can be obtained for all nodes, and a more efficient parallelization could be realized, as a result.

We must define a level at which we implement a change of the type of distribution unit, and this level is referred to the translation level. This level should have the feature that both the number of blocks per node and the number of samples per node must be balanced.

### 2.3.2 *Distribution Algorithm*

At the lower levels, the distribution unit should be the MLFMM groups. However, it is also important to set the criteria for distribution of these groups across the nodes. When setting the criterion for doing this, we must consider several aspects. First, we should realize that the main goal for the distribution is to balance the CPU-time for each node. But it is also very important balance their memory requirements, such that if we set the serial code memory need of  $X$  bytes, when using  $p$  nodes, the memory of each one of them must be close to  $X/p$  bytes. A memory analysis reveals that there is a direct relationship between this and the number of basis functions of the problem. So, given this relationship and this requirement, a good criterion for distribution is to spread the groups among the nodes following the rule that each one must have a balanced number of basis functions.

Additionally, since it is also desirable to minimize the communication, it is recommended that no message will be sent through the network during the process of aggregation and disaggregation. In other words, if a group in the level  $i$  is assigned to the node  $p_0$ , all their children groups at the lower levels and its parents at the upper levels should be assigned to the same node  $p_0$ .

With the same goal of reducing communication through the network, it is very convenient to assign a group to the node  $p_0$ , and assure that all the groups which have multipole coupling with it also belong to the same node  $p_0$ . In this case, the multipole coupling computation would be done by using memory accesses instead of network communications. As a consequence, it could be achieved without any message exchange.

When analyzing these procedures, we realize that the maximum separation between groups that are coupled in the same level should be four groups, because it would be more efficient to calculate them in the upper levels if their separation is larger. So, when performing the distribution, it is desirable that the groups have a distance lower than four groups belong to the same node, because this will serve to reduce the communication during the translation process, and thereby serve to improve the efficiency of the program. So in the distribution, it is also desirable that the groups assigned to each node are as close as possible, provided there is a balance between the number of basis function per node.

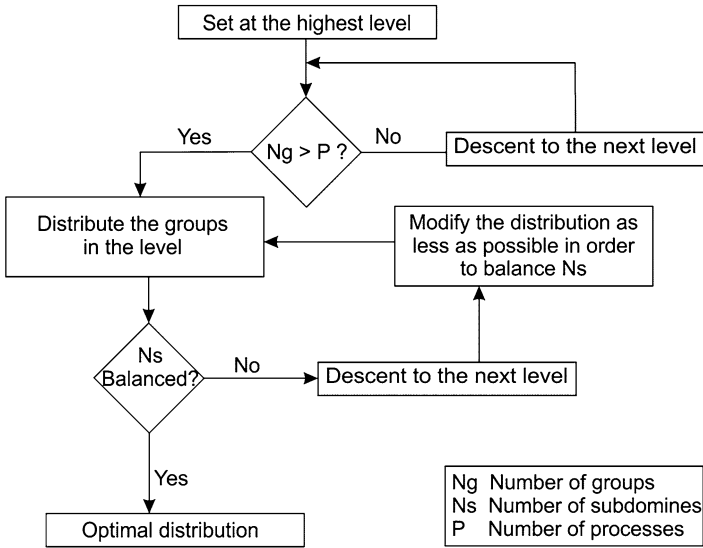
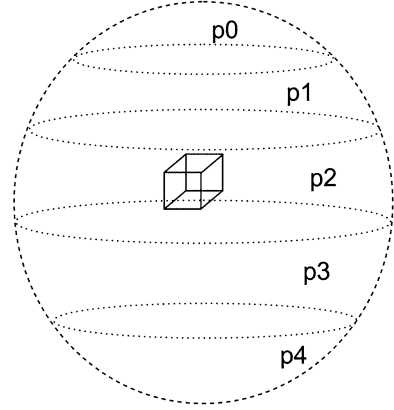
By considering all of the ideas discussed above, the recommended algorithm used for the distribution is as following. First, we set at the highest level and then we look if the number of groups in this level is smaller than the number of nodes. If this is the case, then we descend to the next level; otherwise, we look for a distribution of all its groups among the nodes, searching a balance in number of basis functions. If we succeed, each group of this level will belong to a node, and then all their descendants groups in the lower levels would be assigned to the same node. We achieved several objectives by following this recipe. First, the memory would be balanced because it also will be the number of basis functions per node. Second, it facilitates the aggregation and disaggregation to be performed to this level without exchanging messages and, finally, the groups assigned to each node in the lower levels could be all very close together, which will reduce the message passing on the translation at these levels.

It is possible that we are not able to achieve a good distribution at this level of the groups between nodes, because we cannot balance the number of basis functions is not possible. In this case, we use the distribution that performs the best balance and it is assigned the same node for all its group descendants in the following level to a particular group. Then we descend the lower level and with the data from the distribution at the upper level, we seek a balance at this level in the number of subdomains by introducing slight variation in the assignments of the groups of this level.

For example, if we have a group  $x$  at level  $i$  assigned to the node  $p_0$ , and if the algorithm has not been able to obtain an optimal allocation at this level, then we store this non-optimal distribution. Next, all descendant groups from  $x$  in the level  $i-1$  are also assigned to  $p_0$ , and in this level, namely  $i-1$ , we try to find a balance in the number of basis functions per node by modifying the assignment of the smallest possible number of descendants. This procedure will lead to a balanced distribution at this level  $i-1$  with high number of adjacent groups assigned to the same node.

If it turns out that we are still unable to achieve the desired balance at this level even after modifying the allocation of all the groups to the nodes, we repeat the implementation of the algorithm in the next lower level, and so on (Fig. 2.3).

**Fig. 2.2** Distribution of angular samples among the nodes



**Fig. 2.3** Flowchart of the distributing algorithm

Finally, as mentioned earlier, the criterion for the distribution at higher levels is that a set of adjacent samples are assigned to every node to facilitate interpolations, using a minimum number exchanges of the messages. For easier implementation, the number of samples in theta are distributed among the nodes, to obtain a distribution in slices, as shown in Fig. 2.2. It is possible that when we perform the interpolation of the field at a point located close to the border between two nodes, the node would need adjacent samples that belong to other nodes. In this event, a message will be sent between the nodes in order to obtain the required data.

### 2.3.3 *Strategy for Minimizing Communications*

The sending and receiving of messages through the network is the slowest task in the parallel execution process; consequently, we must find ways to ensure that the number of transmitted messages is minimized. As explained in the preceding paragraphs, it is necessary to assign the lower level groups to each node that are close proximity of each other, since this allows that aggregation, disaggregation and translation could be performed with a minimum amount of communication.

But, even if an optimal group distribution has been achieved at one level, it is possible that at higher levels, their parents groups need to exchange messages in the node of aggregation. Let us imagine that the balance has been achieved at level  $i$ , and at that level we have a group A which has been assigned to the node  $p_0$  and a contiguous group B that has been assigned to  $p_1$ . Let us say that both groups have the same father group C at level  $i + 1$ , and that they have been assigned to the node  $p_0$ . Then, when performing aggregation at level  $i + 1$  we need to obtain the data of group A (located at the same node) and group B (located at a different node). Consequently,  $p_0$  will need to receive a message from  $p_1$  with the group B data. Then, we will need send/receive a message to compute the aggregation of C.

One way to avoid sending/receiving data to/from the group C in the aggregation computation at level  $i + 1$  is to duplicate this group. Thus, we may define, at level  $i + 1$ , a group C0 at node  $p_0$ , which contains the aggregation in the group C of all the descendants groups of C at level  $i$  that are assigned to the same node  $p_0$ , and a group C1 that has the aggregation of groups descending from group C at level  $i$  that are in the node  $p_1$ , and so on. Aggregation of group C will be the addition of the aggregations of group C0, C1, and so on. This same strategy can be achieved for disaggregation. The implementation of this algorithm decreases the need for communication between the nodes associated with aggregation and disaggregation.

There is yet other issue which we need to consider when using this strategy. If there is a group in the node  $p_2$  that is coupled with the group C, and we perform the translation of the group C to the node  $p_2$ , it must send two messages for this group C, one from the node  $p_0$  and the other from node  $p_1$ , each of which contain the partial results of aggregation in group C. Note that if we do not duplicate the groups, we would only need to send one message to  $p_2$ .

## 2.4 Results

We now proceed to present some results of the task distribution algorithm in this Section. First, we show the results for a sphere whose radius is 1 m, and whose centre is located at the origin. It is illuminated by a vertical dipole at the point (0.0, 0.0, 1.5) and operating at 2 GHz. We employ 10 samples per wavelength, which results in a total of 112,144 low-level basis functions and 39,872 CBFs functions (unknowns). The block size for the generation of the CBFs is chosen to be  $1\lambda \times 1\lambda$ .



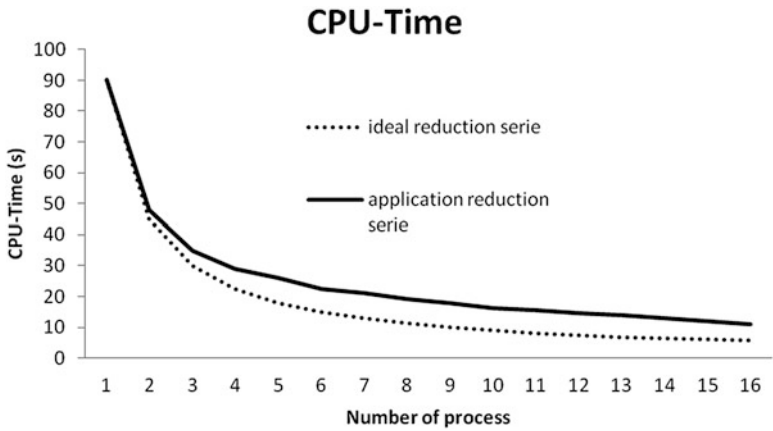


Fig. 2.4 CPU-time reduction with increase in the number of processors

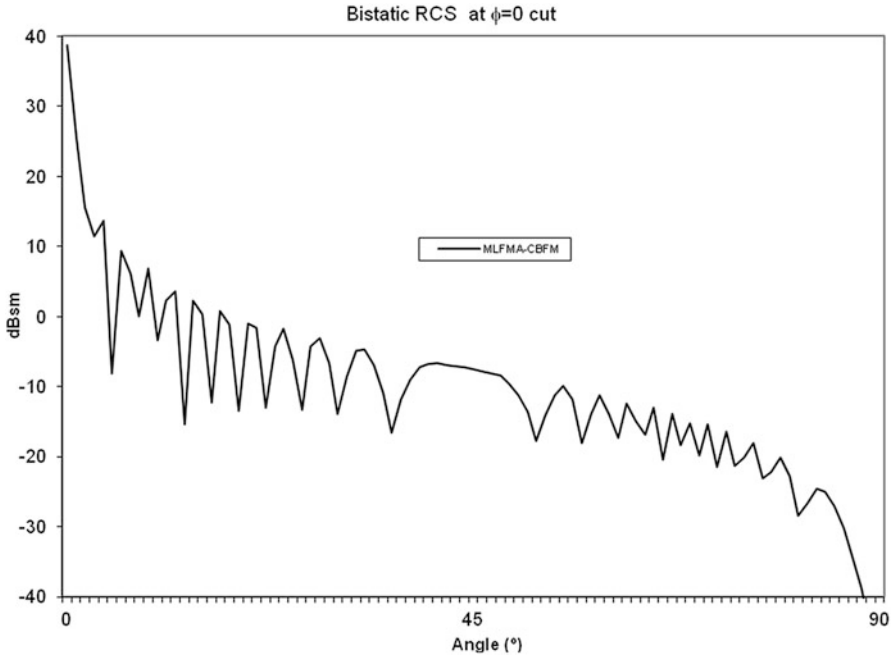
Table 2.1 Algorithm levels for different number of nodes

N of nodes	Distribution level	Translation level
1	7	2
2	6	2
3	6	2
4	6	2
5	6	2
6	6	2
7	5	2
8	5	2
9	5	2
10	5	3
11	5	3
12	5	3
13	5	3
14	5	3
15	5	4
16	4	4

Figure 2.4 shows the CPU-Time per iteration versus the number of processors. The reduction of the CPU-Time of the application compared to the ideal reduction is defined as:

$$\text{CPU} - \text{Time}_{n \text{ node}} = \text{CPU} - \text{Time}_{1 \text{ node}} / n \tag{2.1}$$

The maximum MLFMA-CBFM level in the analysis is 7. On this level, a single cube includes all the geometry. Table 2.1 shows the distribution level and the translation level set for the analysis in these cases. We can conclude that the number of groups per node in a level decreases with an increase in the number of nodes.



**Fig. 2.5** Bistatic RCS for the  $\phi = 0$  cut

Also it becomes more difficult to work with the distribution at the highest levels. As a result, we must descend to lower levels in order to find a proper distribution. As a consequence, the reduction of the CPU-time diverges from that of the ideal case with an increase in the number of nodes.

We can also note that when increasing from 9 nodes to 10 nodes, the translation level increases from 2 to 3. This is because the number of angular samples per node at level 2 is optimum with 9 processors, but this relation is not good when we increase the number of nodes to 10. In this case, we increase the translation level at 3, and the number of angular samples also increases. As a consequence, an optimum relation can be obtained.

The second case we consider is a flat plate which is  $80\lambda \times 80\lambda$  size. The plate requires, when sampled every  $\lambda/10$ , a total of 1,278,400 low-level basis functions and 396,281 CBFs. Bistatic RCS for a normally incident and a  $\theta$ -polarized wave is computed for two different cuts, namely  $\phi = 0$  and  $\phi = 90$ , and are plotted in Figs. 2.5 and 2.6, respectively. In this problem, Fig. 2.7 and Table 2.2 show the performances for several different choices of the numbers of nodes (processors).

Next, the bistatic RCS of an airplane, shown in Fig. 2.8, is computed. It is illuminated by a  $\theta$ -polarized plane wave incident from the angle  $\theta = 0$  and  $\phi = 0$ . The number of low level basis functions is 2,134,057. The results for the  $\theta = 90$  cut are shown in Fig. 2.9. Table 2.3 shows the results of computational analysis when using 12 processors.

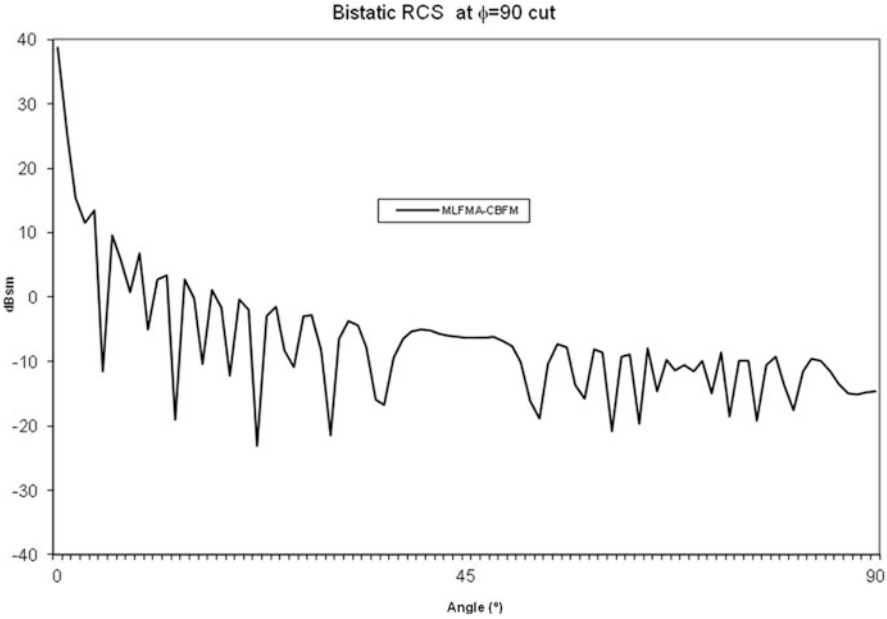


Fig. 2.6 Bistatic RCS for the  $\phi = 90$  cut

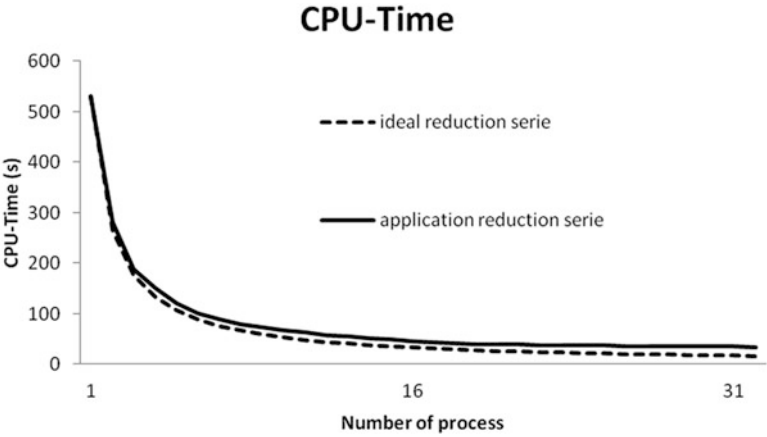
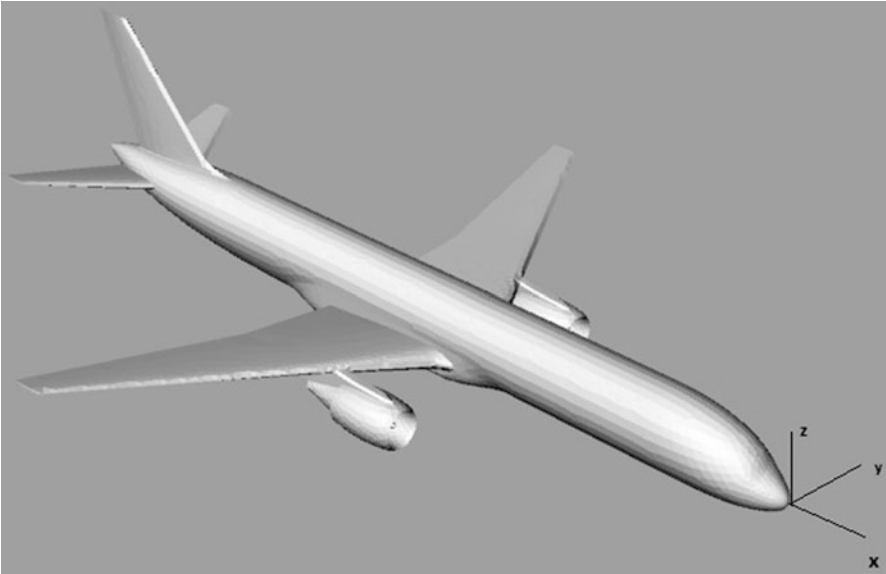


Fig. 2.7 CPU-time reduction with increasing number of processors

Finally, we present the results of bistatic RCS analysis of a ship. Details of the simulation are shown in Fig. 2.10. The structure is illuminated by a  $\theta$ -polarized plane wave incident from  $\theta = 0$  and  $\phi = 0$ . The RCS result for the  $\theta = 90$  cut is shown in Fig. 2.11, and a computational analysis is presented in Table 2.4. The number of

**Table 2.2** Algorithm levels for different number of nodes

N of node	Distribution level	Translation level
1	10	2
2	9	2
4	8	2
8	7	2
16	7	4
32	6	6



**Fig. 2.8** Boeing 757

low level basis function is 2,076,267. The results were obtained by using a parallel version of the tool running on a machine with twelve Intel(R) Xeon(R) 2.0 GHz processors.

**2.5 GPU Strategy**

Rapid growth in demand for video games with realistic graphics, which require a very high computing power, has led to the development of graphic cards with unprecedented capabilities. Because of the nature of the calculations required to generate the 3D game scenes, they can be broken down into numerous short and independent tasks, which can be executed in parallel. For this reason, these cards have been developed with by multiple processing cores, which can run up to millions of simultaneous threads.

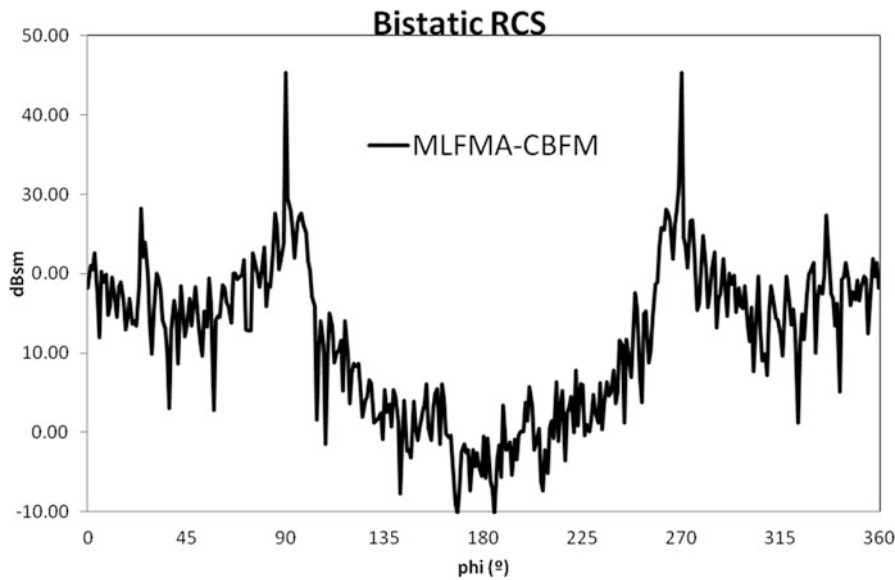


Fig. 2.9 Bistatic RCS for the test case shown in Fig. 2.8

Table 2.3 Computational analysis for the test case shown in Fig. 2.26

Method	N of Unknowns	Group size	Block size	Total time	Pre-process time	Solver time per direction
MLFMA-CBFM	262,376	$0.25 \lambda$	$2.0 \lambda$	14 h 48' 9"	14 h 35' 17"	12' 52"

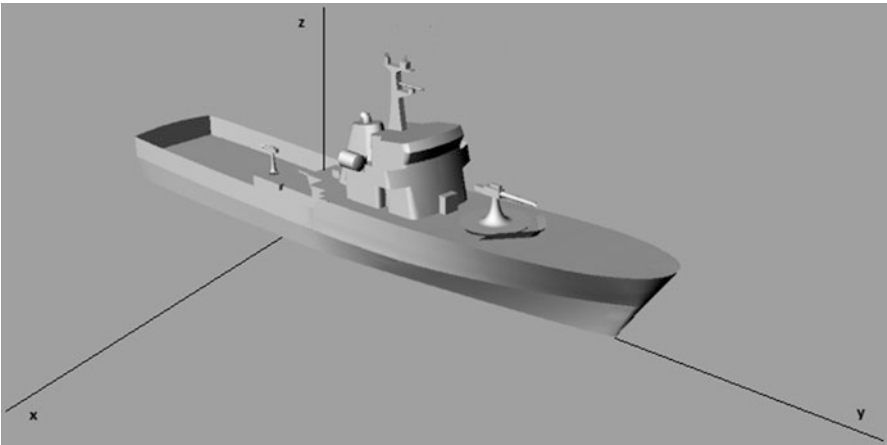


Fig. 2.10 Ship analyzed

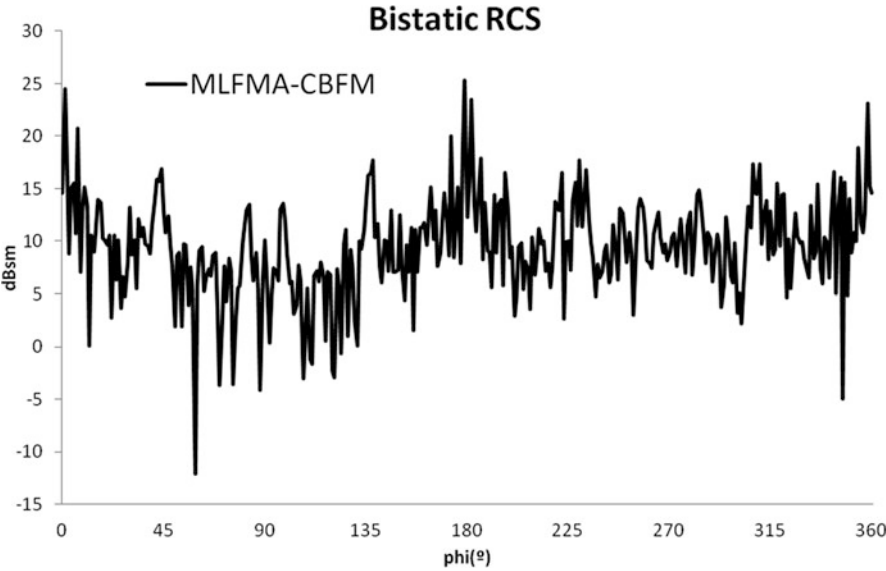


Fig. 2.11 Bistatic RCS computed for the test case shown in Fig. 2.10

Table 2.4 Computational analysis for the test case shown in Fig. 2.10

Method	N of Unknowns	Group size	Block size	Total time	Pre-process time	Solver time
MLFMA-CBFM	249,242	0.25 $\lambda$	2.0 $\lambda$	18 h 33' 41"	16 h 49' 14"	1 h 44' 27"

Initially, GPUs in relative terms were designed primarily for 3D graphics that enabled them to define triangles, textures to be applied, spatial coordinate transformations, etc.

Fortunately, manufacturers of GPUs soon realized the enormous potential that these devices for performing other operations, and not just for 3D rendering. Specifically, a set of extensions have been designed for the most popular languages for running any programs on them. The task of the programmer is now to rewrite their original programs, by using the new extensions of the popular languages to take advantage of the power offered by the GPUs.

We will describe the transformation of the CBFM and MLFMA-CBFM into an implementation, which takes advantage of a relatively new programming paradigm called GPGPU (General Purpose computing on Graphical Processing Units). This new programming framework enables us to use a Graphical Processing Unit (GPU) for general purpose computing, and it is specially suited for scientific work. GPUs, by design, work well with data structures such as matrices and vectors, since they (GPUs) follow the SIMD (Single Instruction Multiple Data) paradigm, where the same operation is repeatedly (and possibly simultaneously) performed on different data sets. Thus, an application which works on matrices and performs the same

operations on a large set of data is a good candidate for being executed on a GPU. This is indeed the case with CBFM.

As mentioned earlier, the main advantage of using a GPU is that it is a parallel machine by nature. If the application allows it, it can deal with a great number of operations at the same time. A GPU such as the NVIDIA TESLA can execute up to 240 operations simultaneously, and can handle millions of them concurrently. The potential for speedup is enormous, but this potential can be limited by the structure of the application. If its operations depend on one with another, then they must be executed sequentially, and the potential of the GPU would be lost. Therefore, great care must be taken when coding the application to exploit all the parallelism that it offers.

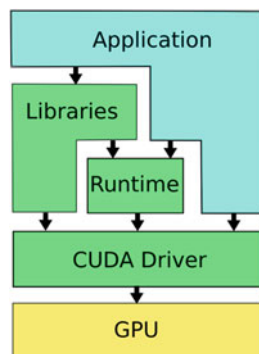
## 2.6 CUDA

Graphics cards are a special kind of hardware, designed to process millions of graphics primitives per second. They do this by processing as many of these primitives as possible in parallel. Primitives are geometrical shapes, and the processing involves geometrical transformations as well as texture mappings. In order to execute many of these operations in parallel, the graphic card requires extensive arithmetic hardware. The idea of GPGPU [13] is to use this hardware, originally intended to perform graphics calculations, to process scientific code, which, in principle, should not be too different from graphic calculations.

One of the frameworks available to an average user today is NVIDIA's CUDA (Compute Unified Driver Architecture) [14] computing platform. CUDA offers a number of software layers, which enable the programmer to access the computing resources of a GPU. The application can do this in one of three ways. In the first of these, one can access the CUDA driver directly. This is a low level approach, in which the driver usually provides more functionality but at the cost of increased complexity. Another way to access the GPU is by means of the slightly higher level runtime software. The CUDA runtime provides a simpler access to the hardware, which is useful for most applications, since they do not need the subtleties provided by the driver. The third way is to use existing third-party libraries that perform certain functions (usually, but not only, mathematical in nature) like those in the BLAS (Basic Linear Algebra Subroutines). The best case scenario is when an application can get all it needs by calling functions from a library, since, in this case, it does not even need to know that it is using the GPU. Unfortunately, this is not usually the case, and some access, at least to the runtime, is necessary. The libraries may themselves access the runtime or use the driver, and the runtime uses the driver to get the job done (Fig. 2.12).

In the most general form, the application will use either the runtime or the driver, and this means that it has to interact with the GPU. For this type of interaction, it uses the GPU like a traditional supporting mathematical co-node, although this is a more sophisticated, -and in terms of programming, more demanding- use. In

**Fig. 2.12** Software layers for accessing a GPU



general, the sequence of events in the application is as follows (more detail later). First, prepare the data for processing in the GPU; then transfer the data to the GPU memory; next, invoke a kernel, which is a special type of function which runs on the GPU. The kernel processes the data and generates the results that are transferred back to the CPU (Central Processing Unit, the traditional computer's processor) for further processing required by the application. Alternatively, the results of a kernel can be used as input for another kernel. The details are provided in the figure below. In it, Kernel 1 processes the data copied into the GPU and the application obtains the results for further processing. This is different in the case of Kernel 2 and Kernel 3. Kernel 3 continues the processing initiated by Kernel 2 and works either on the same data as Kernel 2, on its results, or on both. At any rate, after GPU processing, the application receives the results from the GPU.

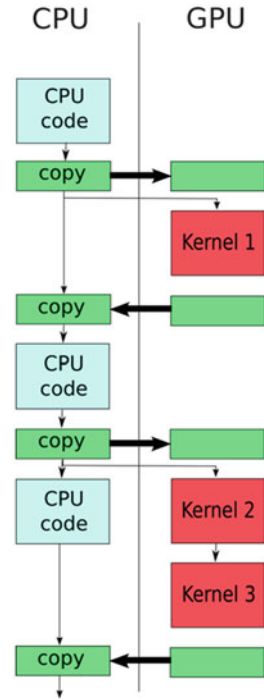
It is important to note that the memory transfers to and from the GPU, as well as the kernel executions, are initiated by the application in the CPU. Only recent versions of the runtime and drivers allow for a kernel to launch another kernel, but in general (and in the case relevant to this chapter), this is not the case. It is possible, however, to have the code running in the CPU simultaneously with a kernel running in the GPU, as is the case for Kernel-2 in Fig. 2.13. Some GPUs even allow for the data for one kernel to be transferred to the GPU while a different kernel is still running. If a code is concurrently running on the CPU, then this makes the maximum use of the available resources of the system.

A CUDA application consists, then, of traditional-style code, which executes on the CPU, kernels, which, as has been said, are special functions that are executed on the GPU, and the code which transfers data to and from the GPU. The compiler knows which code runs on each device, and it generates the machine code accordingly. The CUDA system integrates with existing developing environments, like gcc on Linux and Visual Studio on Windows, and only processes the part of the application destined to the GPU. The part of the application which runs on the CPU it processes with the CPU compiler (gcc or the Visual Studio compiler).

The GPU code follows a programming paradigm called SIMD (Single Instruction, Multiple Data). This means that for every instruction in the code several

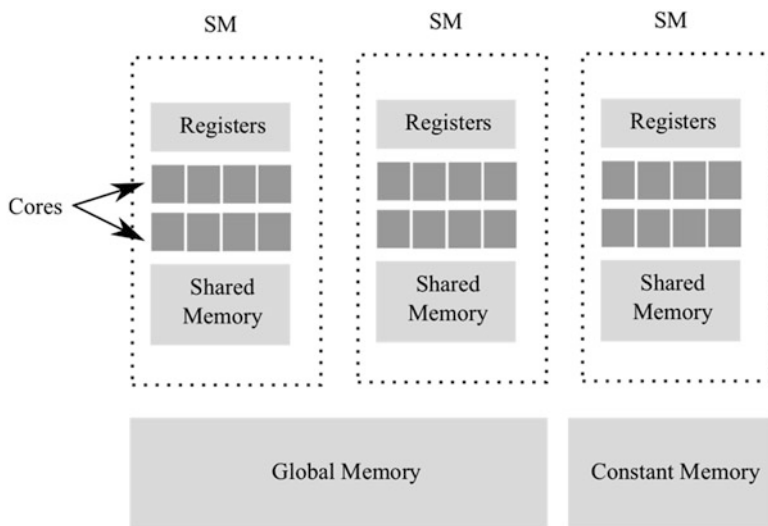


**Fig. 2.13** CBFM-GPU code scheme



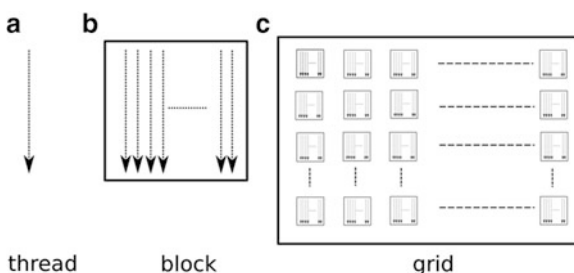
“clones” are made, each one of them working on a different piece of the data. Several copies of the instruction are executed concurrently in the GPU, causing the execution times of the applications to decrease significantly. The drawback is that the application has to contain sufficient data for all the execution units, or the potential for speedup is lost. In other words, only those applications which exhibit a high level of data parallelism (and make it explicit) will benefit from a GPU. Fortunately, scientific code often meets this requirement, for example, when it works with vectors. Still, to take advantage of the computing power of a GPU, the kernels must be coded with the SIMD paradigm in mind.

In order to be able to execute several instances of the same instruction at the same time, the GPU structure is designed in a special way (see Fig. 2.14). Its basic computation unit is called a core, and is essentially an arithmetic-logic unit (ALU). Cores are grouped into Streaming Multiprocessors (SM), which, besides the cores, contain registers for quick data access, and shared memory for common data. A Streaming Multiprocessor has a set of instructions executing and sharing data through a Shared Memory. A GPU has a number of SMs ranging from 1 for low-end products to 30 in high-end products. It must be said that some recent GPUs have reduced the number of SMs while increasing the number of cores in each one, while providing a global increase in the total number of cores. The most recent NVIDIA Kepler model of GPUs has up to 8 SMs, each one consisting of 192 cores, for a total of 1,536 cores.



**Fig. 2.14** Block diagram of a GPU

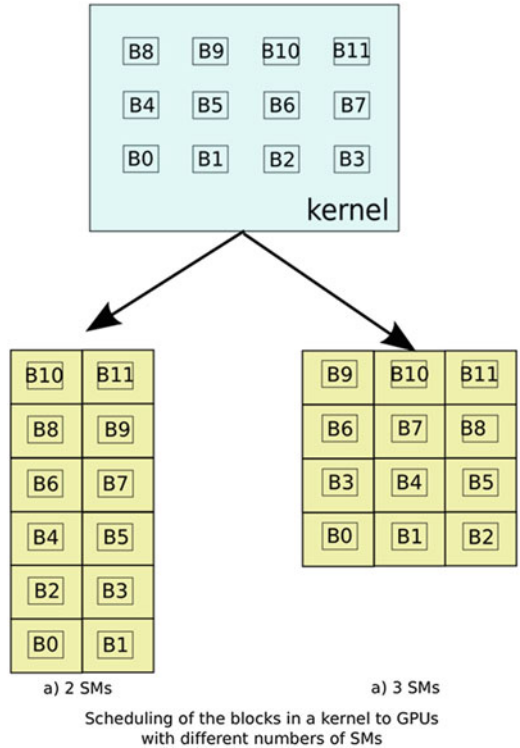
**Fig. 2.15** Logical scheme of a GPU kernel



Given the hardware structure of a GPU, the software must be organized accordingly. Applications with a high level of data parallelism will benefit most from the hardware of the GPU. These applications consist of sequences of operations repeatedly applied to different sets of data. These sequences are chosen to be executed on the GPU, in the form of kernels, one for each sequence.

A GPU kernel is a grid of blocks of threads, as Fig. 2.15 shows. Each thread executes one instance of a sequence of operations on one piece of data, but all threads in the same grid execute the same sequence, only the data is different. Threads are grouped into blocks so they are easier to handle, since, in this way, one control unit is able to handle a whole group of threads. One block executes on a single SM. The blocks in a grid are assigned to SMs dynamically, as it is possible (and advisable) that a kernel has a number of blocks greater than the number of SMs in the GPU. Many blocks can be assigned to the same SM sharing the resources of the SM. Threads in a block are arrayed in a 3D matrix, while the blocks arrayed in a grid in a 2D matrix. The number of threads in a block depends on the resources needed by each thread, in terms of registers and shared memory used. Usually, the

**Fig. 2.16** Block scheduling in two different GPUs



maximum number of threads per block is 512, although newer models allow up to 1,024 threads in one block. A grid can hold as many as  $2^{16} \times 2^{16}$  blocks, which makes more than four billion bocks, or over two trillion threads. This gives an idea of the amount of parallelism that a GPU can handle.

Of course, since a GPU has at most 30 SMs, and a single SM can handle only a few blocks, depending on the resources it needs, a very large grid would not have all of its threads being executed at the same time. The GPU assigns a block to an SM as soon as it is available. The organization of a grid in blocks makes it very easy to adapt the kernels of an application to all types of GPUs. Figure 2.16 shows how the blocks can be assigned to different GPUs, regardless of the number of SMs available.

Transforming a sequential code into GPU code is critical for taking advantage of the possibilities that it offers. So much so, that it is sometimes more advisable to construct the whole application anew with the GPU in mind. Other times, some parts of the old code can be used, parts which would not take advantage of running on a GPU, anyway, and those other parts which would, can be modified in order to take advantage of the GPU. Each data-parallel section in the application will be executed in the GPU, and the rest of the application will run on the CPU. In-between, the data will be transferred into the GPU and the results out to the CPU, as in the sequence shown in Fig. 2.13.

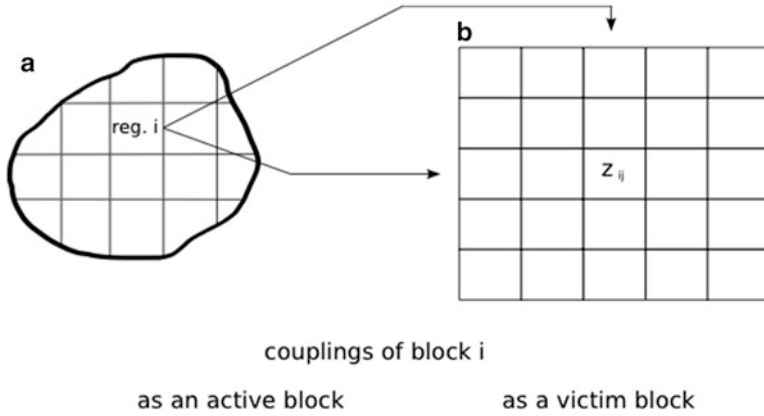
When crafting a GPU code, it is important to keep in mind that the code in the kernel will be executed by every thread. Threads have identifying indexes, and blocks have their own indexes, too, in order that each thread can find which data to work on. It is possible to include control structures in the kernel, so that different threads may do different things depending on the indices. But this is not advisable and, in fact should be avoided. This is because, under these circumstances, the execution model states that when some threads execute, the others are idle and waiting (or other way around), and this results in loss of efficiency.

Each thread can access data in the registers, shared memory, constant memory and global memory. The access time for each storage type is different, and grows in the order in which they are listed. Registers are the fastest of all, and they are where variables local to the kernel are stored. Each thread has one private copy of each local variable, so the number of used registers grows rapidly. Next in line is shared memory, which is accessible to all threads in the same block. It is slower than the registers, but larger, and it is advisable to use it for data that are commonly accessed, or for shared data. Global memory is the most abundant and slowest type of memory in the GPU. It holds all the data that are sent to the GPU from the CPU, and can be accessed by all threads in any block. Since it is slow, data that are accessed repeatedly can be copied into shared memory for repeated access there, which is faster. Results should be written in global memory when the kernel has completed its task, in order to send it from there to the CPU, or to another kernel for later use. Constant memory is a special part of global memory, specifically meant for constants. It is cached, which means that, after it is read by a thread, all others in the same SM will be able to read it faster. CUDA allows the programmer to specify in which kind of memory a variable will be located, which is very important in order to obtain good execution times. The memory levels of the CUDA hierarchy may be found in Fig. 2.14.

The task of the programmer is, to first decide which parts of the application can benefit from execution on the GPU. Then, kernels must be formed from these parts. The sequence of the kernels corresponds roughly to the sequence of parts in the application, although it is possible that a part of the application is divided into two or more kernels. A decision must be made with regard to the disposition of the data on several layers of memory in the GPU, and then the kernels must be coded. Data transfers must be placed conveniently, if possible, so as to avoid redundant transfers. Sometimes it is more time-efficient to recalculate data than to transfer it back and forth from the GPU. In the next section we describe how we did this for the CBFM.

## 2.7 CUDA Version of the CBFM

An overview of the CBFM [15] has been presented in Chap. 1. It was also mentioned that, in the case when the geometry is too large to be processed in the GPU in its entirety, it is necessary to divide it into smaller pieces, called CBF-blocks. When



**Fig. 2.17** Matrix equivalence of the electromagnetic problem

processing a geometry piece-by-piece, the impedance matrix  $Z$  is also obtained in the same manner. This is shown in Fig. 2.17. The sub-matrices in  $Z$  correspond to the couplings of one CBF-block of the geometry with another. If the geometry is divided into  $n$  CBF-blocks, the impedance matrix  $Z$  will consist of  $n \times n$  sub-matrices. The sub-matrix  $Z_{ij}$  corresponds to the coupling of block  $i$ , the victim block, with block  $j$ , the active block.

In the CBFM chapter it was shown how it is possible to obtain the CBF-based impedance matrix  $Z$  from the MoM version of this matrix. The relationship between both matrices was described by (1.7) in the CBFM chapter. The transformation, in matrix form, is:

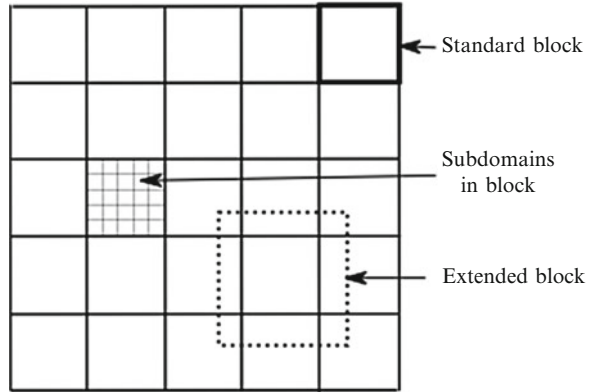
$$Z_{cbf} = U^H * Z_{MoM} * U \quad (2.2)$$

In the rest of the section we refer to  $Z_{cbf}$  as  $Z'$  and to  $Z_{MoM}$  as simply  $Z$ . The transformation matrix  $U$  can be obtained by applying the SVD to the matrix  $J$  of currents induced on the geometry by a set of excitations  $V$  obtained from the Plane Wave Spectrum. This matrix,  $J$ , uses the subdomain-based representation for these currents. When the geometry is subdivided into blocks, the transformation matrices are obtained by applying the SVD to each sub-matrix  $J_i$ , which represents the currents induced into the subdomains of a sub-block by a set of excitations  $E_i$ . To calculate  $J_i$ , using MoM it is necessary to solve the system:

$$V_i = Z_i * J_i, \quad (2.3)$$

where  $Z_i$  is the impedance matrix between the subdomains in block  $i$ . Note that it corresponds to sub-matrix  $(i, i)$  of the global impedance matrix  $Z$  (that is, element  $i$  of the diagonal). One consequence of this is that the transformation matrices  $U_i$  are

**Fig. 2.18** Extension of the block for CBF computation



obtained from the impedance matrices which represent the couplings of each block with themselves. These transformation matrices can later be used to obtain the rest of the CBF-based  $Z$  sub-matrices, where the active block and the victim block are no longer the same. The general expression which describes the transformation of the sub-matrices of the subdomain-based impedance matrix  $Z_{ij}$  into the corresponding CBF-based submatrices is:

$$Z'_{ij} = U_i^H Z_{ij} U_j \quad (2.4)$$

In our implementation, this implies that it is necessary to process the sub-matrices of  $Z$  in the diagonal first, since this provides the transformation matrices  $U_i$ , and the rest of sub-matrices only later.

Another problem which arises when dividing the geometry into blocks, in the context of the MoM, is that of electrical continuity. This problem does not exist if the CBFs are calculated by means of Optical Physics, but this method is not used in this version of the application. If we consider only the subdomains in a block when processing that block, we would be calculating the impedance matrix for that block as if it were isolated from the rest, which would not be the case; consequently, the result would be incorrect. To solve this problem, it is sufficient to include in the block, only for the purpose of processing (that is, without really removing them from the other blocks) those subdomains from other blocks that are close, up to a limit, to the block being processed. This yields an extended block (see Fig. 2.18). The extended block is used, as has been mentioned earlier, to provide electrical continuity, and only as long as the calculation of the induced currents. At that point, only the currents induced in the subdomains of the regular block are retained for further processing, as electrical continuity has already been achieved by then.

With all of this in mind, our algorithm, as will be described in the rest of this section, is organized around two sets of loops. The first set extends through the CBF-blocks in the geometry to calculate the transformation matrix  $U_i$  for each block and the reduced impedance matrix  $Z'_{ii}$ . (Note that, in order to calculate matrix  $Z'_{ii}$ , we

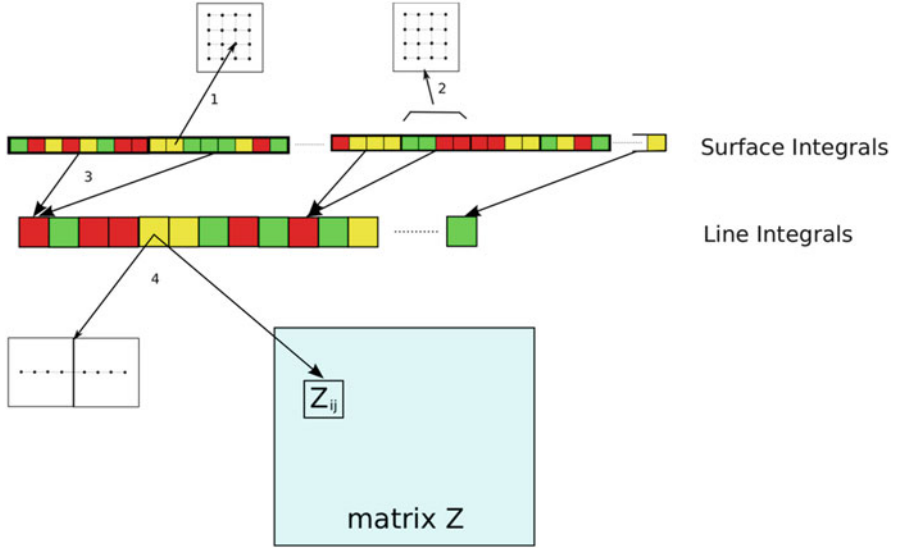
only need  $U_i$ , the transformation matrix for that block). The second loop is really a nested one, where we first calculate, for each extended block as the victim block, and all other extended blocks as the active blocks, the subdomain-based impedance sub-matrix  $Z_{ij}$ , and then the corresponding reduced sub-matrix  $Z'_{ij}$ .

The algorithm starts by analyzing the geometry. It calculates several relevant geometric and electromagnetic data. Also, the set of plane wave excitations is generated, for later use the induced currents are calculated. This part of the application has not been modified from the original implementation, since it does not consume too much time. Part of this data is transferred to the GPU for later use. Then, two kernels are called in sequence. The first one calculates the points necessary in the complete geometry for the razorblade integrals, for every pair of subdomains. The second set of kernels calculates, for every CBF-block, the subdomains that are sufficiently close to it, and, hence, must be included in the corresponding extended CBF-block.

Then, the loop which processes the sub-matrices in the diagonal of  $Z$  is executed. For every CBF-block, it generates a list of the subdomains involved, and includes the points for the razorblade integrals. This information is transferred into the GPU. Then, for every point in each razorblade integral, a kernel calculates how many points are necessary for the surface integral. Also, other data for the integral is calculated in the GPU, such as the coordinates of the points in the razorblade and of the points for the surface integrals. The information generated is retained in the GPU, for later use by other kernels.

After this set of data has been calculated, a function in the CPU classifies surface integrals in three groups, depending on the number of integration points. We use 2, 4 and 10 points in both axes. Since the integral must be calculated on both patches of each rooftop, this makes 8, 32 and 200 evaluations for each integral. This provides three lists of integrals, and each list is processed separately by a different kernel. For instance, the kernel for 4 point-integrals calculates all the surface integrals where 4 points are used. This includes both function evaluations and weighted additions of the values obtained. The additions correspond to the Gaussian Quadrature. The result is a list of values, one for each surface integral, which is retained in the GPU. This is because, following that, three more kernels calculate the results of each linear integral. The number of points in the integral (again 2, 4 and 10) determines which one of three kernel handles it. Each kernel processes all line integrals with the same number of integration points for which it is responsible. The result of each integral is an element of the subdomain impedance matrix  $Z_{ji}$  for the CBF-block under process. The results are again kept in the GPU for the kernels which must process them after these. The calculation of sub-matrices that are not on the diagonal follows this same procedure, as is explained below.

All these processes are illustrated in Fig. 2.19. The boxes labeled ‘Surface Integrals’ correspond to threads which perform function evaluations, given in (1) for the surface integrals. Each thread executes one function evaluation. They are grouped into thread-blocks, and each thread-block accumulates (following the method of Gaussian Quadrature) the results of the function evaluations into a single value (2), the result of the integral. The different colors represent the



**Fig. 2.19** Computation scheme for the integrals in the MoM matrix

different numbers of integration points in the surface integral. The next set of kernel invocations, labeled ‘Line Integrals’, corresponds to the line integrals. In this case, each thread in a thread-block accumulates the results of those surface integrals, given in (3), which correspond to the line integral for which it is responsible. One of these threads processes the results of several thread-blocks from the kernels of the ‘Surface Integrals’, given in (4). Thus, each line integral produces one element of the subdomain matrix  $Z$ , in (4). Again, a separate kernel handles the integrals for each possible number of integration points, yielding three kernels. A flow-chart-like representation of this process is shown in Fig. 2.20.

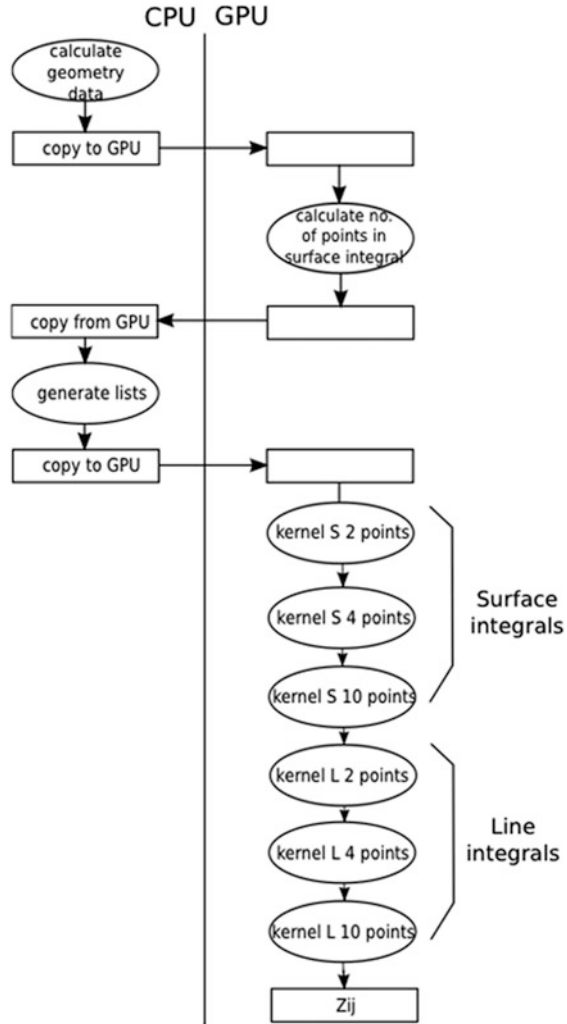
Once the subdomain impedance matrix for the CBF-block has been computed, the matrix  $U_i$  for this block must be obtained. At this point, the excitation matrix  $V_i$  for the subdomains in the current extended block is extracted from the excitation matrix for the whole geometry  $V$ . It must be noted that both  $V$  and  $V_i$  are sets of excitations, which implies that they are matrices. Each column of the matrices corresponds to an excitation, and each row to a subdomain, and  $V_i$  is a subset of the elements of  $V$ . The matrix  $V_i$  is then sent to the GPU in order to solve the system of equations

$$V_i = Z_i * J_i. \quad (2.5)$$

The function `cgesv` from the CULA library is used for the purpose of solving the linear system of equations. CULA is an adaptation of the LAPACK (Linear Algebra PACKage) library to CUDA. It is provided by EM Photonics, and has a free version, which is the one we used.



**Fig. 2.20** Flowchart of the computation of the MoM matrix



The result of the previous step is vector  $J_i$ . It contains values for all subdomains in the extended block, and the next step is to strip it of the values for all the subdomains that are not in the standard block. This is also done by a kernel in the GPU. The results are kept in the GPU, because they must be processed by function `cgesvd`, also from the CULA library, which applies the SVD to the matrix of induced currents for the standard block. This yields the transformation matrix  $U_i$  and a list of Singular Values in decreasing order. Each column of  $U_i$  is a characteristic basis function, and corresponds to a singular value. The list is transferred to the CPU for analysis, discarding those that fall below a specific threshold. This threshold is set at  $1/500$  of the value of the largest singular value. All the columns of  $U_i$  which correspond to values below this threshold are discarded as well. This is shown in

**Fig. 2.21** Result matrix after the SVD process

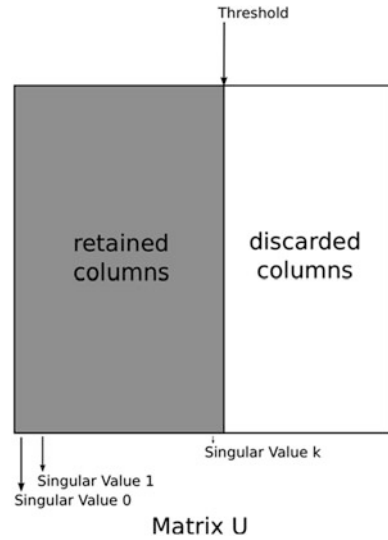


Fig. 2.21. The resulting matrix, which we will also refer to as  $U_i$  -because the complete matrix is never really used-, is the desired transformation matrix. It is transferred from the GPU to the CPU to make room for processing of the remaining blocks. Finally, the reduced impedance matrix for the block is obtained from:

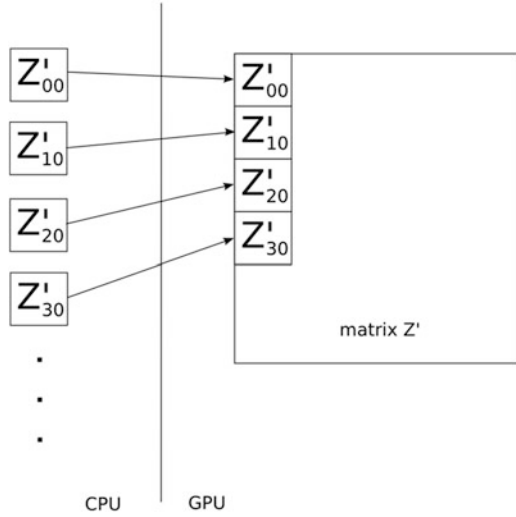
$$Z'_{ii} = U_i^H * Z_{ii} * U_i. \quad (2.6)$$

The matrix multiplications are executed in the GPU with function `cgemm` from cuBLAS, an adaptation of the BLAS (Basic Linear Algebra Subroutines) library to CUDA, provided by NVIDIA. The reduced matrix is removed from the GPU, also to make room for the remaining calculations.

Once the sub-matrices in the diagonal of  $Z$  have been processed, the rest of the sub-matrices can be, too. In the case of these sub-matrices, the processing is shorter, for reasons that have been explained above. First the impedance matrix is calculated, then the reduced impedance matrix is obtained from it, using the corresponding transformation matrices  $U_i$  that have been calculated before. Again, the function `cgemm` is used for this task. The resulting reduced matrix is removed from the GPU and stored in the CPU memory.

The last step involves putting together all the sub-matrices from the reduced impedance matrix  $Z'$ , that have been stored in CPU memory. This is shown in Fig. 2.22. For this, they are returned to the GPU, and a kernel is invoked, which assembles them together. The resulting matrix is kept in the GPU since, although the process for generating it has been completed at this point, it is only the first step for other processes which will subsequently use the matrix to analyze the geometry. It may appear to be a waste of time to take the sub-matrices of  $Z'$  from the GPU to the CPU and then back to the GPU, but it is necessary because the calculation of the

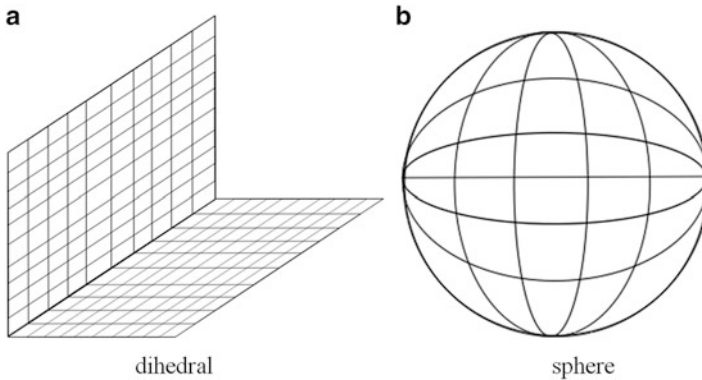
**Fig. 2.22** Extended matrix computation



impedance matrices takes up a lot of space in the GPU. In fact, the more free space is available for these calculations, the larger the blocks in the geometry can be, and this is really time-efficient. In fact, the best option is to process the geometry in one piece, if possible, and only do it piece-by-piece if this is not.

The arrangement of the kernels in this implementation has gone through an evolution based on empirical results of efficiency obtained. Initially, the kernels which calculate the surface integrals used global memory for temporary storage. Shared memory was found to be a more efficient alternative, since these data were shared by threads in the same thread-block. Also, using the register memory was not feasible for these data, because of its large size, and because the compiler does not allocate arrays of data (the form used for these temporary values) in register memory.

Also, we found that the surface and line integrals are performed much better with a separate kernel for each number of integration points. Initially, one kernel performed all of the surface integrals. Each thread started by selecting which section of the code from the kernel to execute based on the number of integration points it had to use. This was inefficient, since it involved several successive branches (a selection), and this control structure is not recommended in CUDA. The cause of the inefficiency is the way in which CUDA handles the branches. Since all threads in the same thread-block execute the same code (the same instructions at the same time, in SIMD fashion), whenever a branch is found, CUDA temporarily stops threads which must continue along one of the paths of the branch, executing only the others, until the point of convergence. Once there, these threads wait, and it is the turn for the others to execute until they, in turn, arrive at the point of convergence. Control structures with successive branches cause many threads to become inactive, and this results in a loss of efficiency. To avoid these problems, we use separate



**Fig. 2.23** Test cases (a) dihedral, (b) sphere

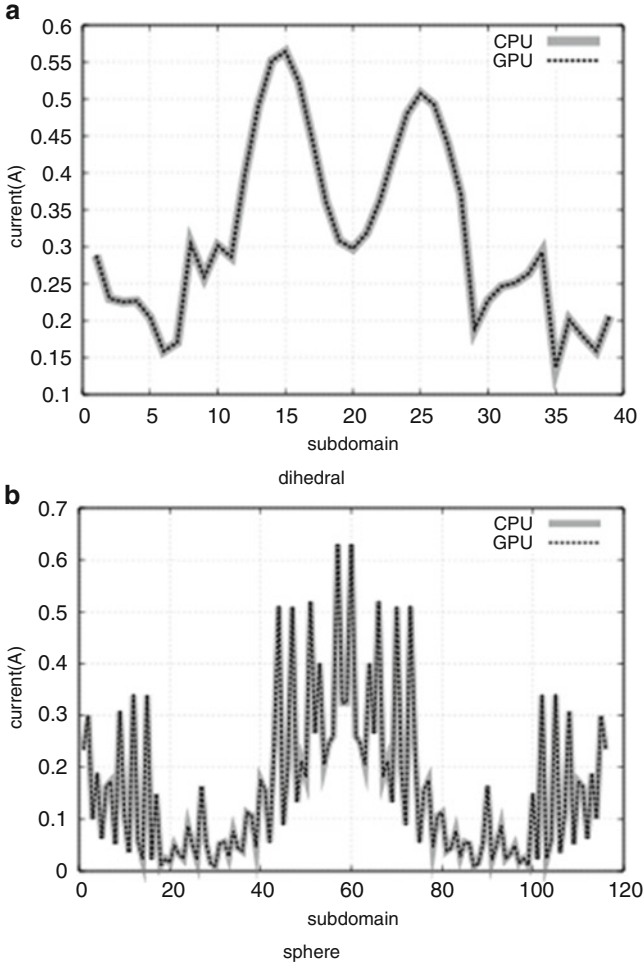
kernels for each set of integration points, both for surface and line integrals. The cost incurred is three times that of the kernel launch overhead, but the benefits greatly outweigh this cost.

### 2.7.1 Numerical Results

We have tested the implementation, described above, on a computer with two 2.27 GHz Intel Xeon E5520 processors, each with 4 cores. The system RAM was 16 GB. The GPU was a Tesla C1060 card with 4 GB of global memory. We considered two test geometries, namely a dihedral and a sphere, as shown in Fig. 2.23. To avoid the generation of the same geometries, but in different sizes, when necessary, we modify the size of the subdomains instead. The number of subdomains is chosen to accommodate the problems that range from very to one which fills up the entire GPU memory.

We compare the results of the original sequential version and the GPU version by analyzing the calculated intensities of current in the subdomains in a cross-section of the geometries. This is shown in Fig. 2.24. As can be seen, the results are almost identical to those obtained by using the sequential version, which is used as a reference.

Next, we analyze the time required to execute both versions. Figures 2.25 and 2.26 plot the speedups achieved by using the GPU implementation. Figure 2.25 presents the results for the dihedral, and Fig. 2.26 those for the sphere. Both work on the geometry in one block. Each figure shows the results for calculating only the subdomain-based impedance matrix  $Z$  and the CBF-based matrix  $Z'$ . The number of subdomains in the geometry grows until it is too large to fit in the GPU in one piece. As can be seen, the speedup grows with the size of the problem. This is due to the fact that there are some tasks that are almost independent of the problem size,



**Fig. 2.24** Comparison of results obtained by using the GPU and sequential (CPU) codes for the induced current (a) dihedral, (b) sphere

and, when the problem is small, they take up a large portion of the execution time. As the problem grows, they become proportionally smaller, hence the increasing speed-up.

Also worth noting is the smaller speed-up achieved when calculating the CBF-based matrix  $Z'$ . This part of the program in the sequential version primarily utilizes Intel's Math Kernel Libraries. These libraries are multithreaded, and they make use of all eight cores of the system. Thus, we are not comparing a GPU with a CPU with sequential execution, but with a multithreaded CPU execution, and the results of the speedup are quite interesting.

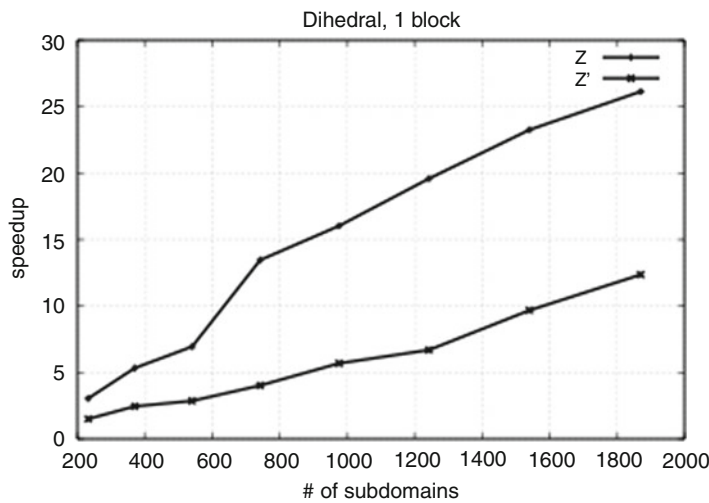


Fig. 2.25 Speedup of GPU version versus sequential version

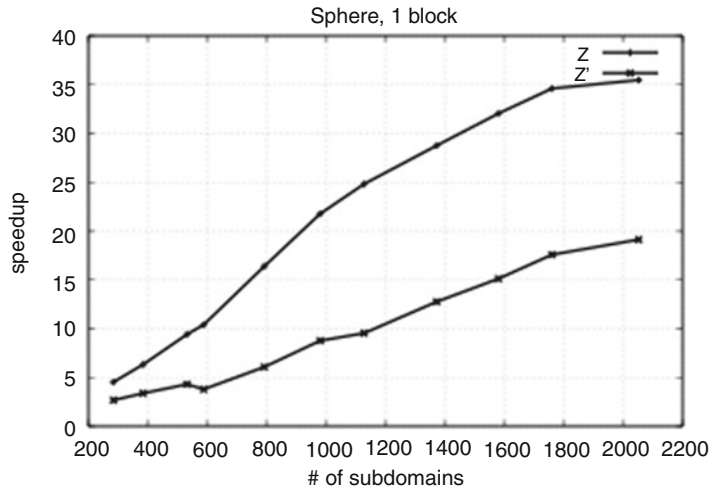
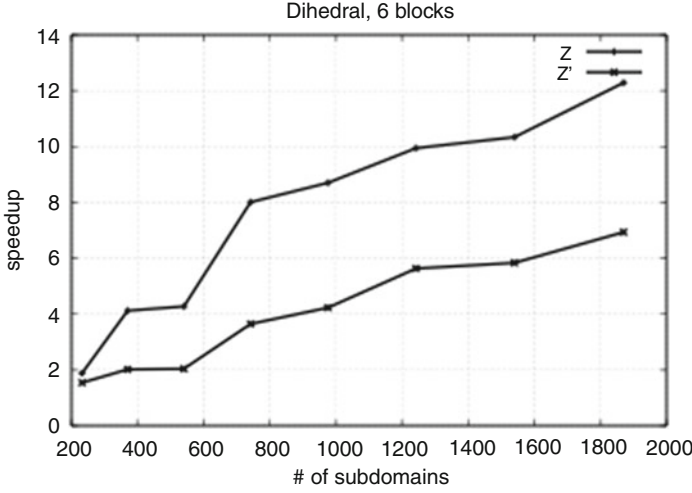


Fig. 2.26 Speedup of GPU version versus sequential version

We are also interested in finding the effect of dealing with geometries that have been subdivided. This is necessary for large geometries, which do not fit in the GPU memory in one piece. The results are shown in Fig. 2.27, when the geometry is, subdivided into six blocks. The speedup results are more modest for this problem. The results shown are only for the dihedral, but those for the sphere are similar. The speedup shown is for the GPU version with six blocks versus that of the sequential version with the entire geometry in one block. One might argue that the comparison is not totally fair, since it is biased in favor of the sequential version,



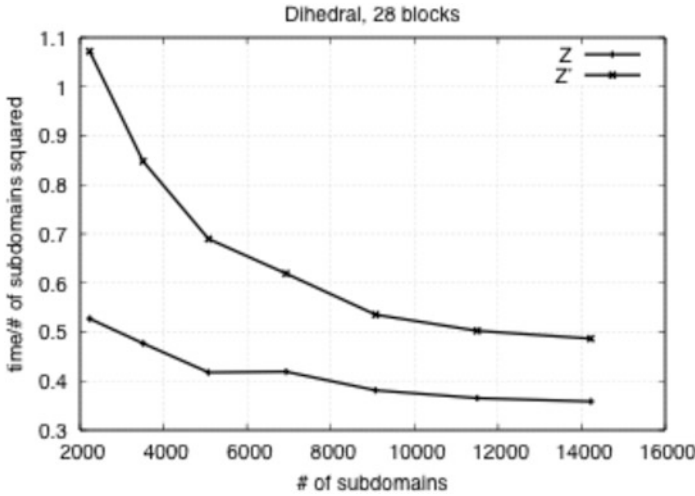
**Fig. 2.27** Speedup of GPU version versus sequential version, six blocks

though it can also be argued that it is not the fault of the CPU that it has a memory large enough to accommodate the entire geometry. Still the GPU is seen to improve on the sequential code, and again the speedup grows with the problem size. The speedup for  $Z'$  is also smaller than it is for  $Z$ . The overall reduced speedup is attributable to the fact that, because of the use of extended blocks, the couplings of many pairs of subdomains are computed several times, as those pairs are present in several extended blocks.

In order to get an idea of what happens as the size of the geometry becomes larger, since, in principle it is possible to work with arbitrarily large geometries by subdividing it in blocks, we let the number of subdomains grow. To handle this problem, we divide the geometry into 28 blocks. Now the problem becomes too large, even for the larger CPU memory, so we cannot compute the speedup. Thus, in Fig. 2.28 we plot the execution time divided by the number of subdomains squared, since the calculation of the impedance matrix  $Z$  has a complexity related to  $O(n^2)$ , if  $n$  is the number of subdomains. We see that, after a large drop, the plot begins to decrease more gently from about 8,000 subdomains, which is an indication that dealing with the geometry in blocks scales well with an increase in the number of blocks.

## 2.8 CUDA Version of the MLFMA-CBFM

As mentioned in Chap. 1, MLFMA-CBFM is a technique which enables us to analyze very large electromagnetic problems. This method entails the storage of the diagonal and near-diagonal terms of the reduced matrix, and the computation of



**Fig. 2.28** Execution time for larger geometries

the rest of the terms by using aggregation, translation and disaggregation schemes. This computation is carried out by using an iterative procedure for the resolution of the system equation. This step may consume a considerable amount of CPU-time; hence, it is a good candidate for parallelization by using the CUDA GPU devices.

The main problem in the algorithm is the dependence between the steps. For instance, before the disaggregation step, the translation process should have been completed, and it is necessary that aggregation be finished before starting the translation step.

So, in the first approximation, we should consider different kernels for every step in the algorithm. But, there is also some dependence inside of every step. For example, we need to ensure that the aggregation of the previous level be completed when aggregating one particular level.

In the aggregation of the first level, a kernel is implemented following the next rule. A set of CBF-blocks is assigned to each GPU block, and the different threads on the block compute all of MLFMA angular sample. As a consequence, all angular samples are computed simultaneously.

A similar strategy is followed in implementing the aggregation of the rest of levels. This step has the unique feature that the aggregation in a certain level needs the results of the aggregation of the previous level. Hence, a kernel has been developed to distribute MLFMA cubes to the GPU blocks, and computing one of the angular samples of every thread. Because of the dependence between levels, this kernel is invoked as many times as levels of the algorithm.

The same problem is faced in the step involving translation/disaggregation, and we implement a top-down scheme for this step. First, the translation of the top level is computed; next, its terms are disaggregated to the previous level. Then, the translation is calculated for this level, and then disaggregated to the previous level,



and so on. As we can infer, there is a similar dependence between the levels as there is for aggregation. So, a kernel has been defined for this step, and it is invoked once per level.

Finally, we compute the level one disaggregation by following the same strategy as in case of level one aggregation. There is a distribution of the set of CBF-blocks to the GPU blocks, and each of the threads compute the MLFMA angular samples in parallel.

### 2.8.1 Numerical Results

In order to demonstrate the computational advantages of the method, we analyze of the monostatic radar cross section of a  $1 \lambda$  plate. The number of low-level basis functions is 760, and this number is reduced to only 207 CBFs. The CPU-time spent in the computation of one vector–matrix product in the serial approximation is 390 ms, but in the parallelized version using CUDA/GPU this time is reduced to only 40 ms; consequently, there is an improvement of 11 times in the solve time of the problem. If we analyze the kernels independently, we can conclude that the best reduction in the time is achieved in the first levels aggregation and disaggregation steps, and the reason is that the calculations are not dependent.

**Acknowledgements** This work has been supported in part by the Comunidad de Madrid Project S-2009/TIC1485, the Castilla-La Mancha Project PPII10-0192-0083 and the Spanish Department of Science, Technology Projects TEC2010-15706, and the Comunidad de Madrid Alcala University Project UAH2011/EXP-015.

## References

1. Laza VA, Matekovits L, Vecchi G (2005) Synthetic-functions decomposition of large complex structures. Antennas and propagation society international symposium, Albuquerque, July 2005
2. Matekovits L, Vecchi G, Dassano D, Orefice M (2001) Synthetic function analysis of large printed structures: the solution space sampling approach. Antennas and propagation society international symposium, Boston, July 2001
3. Prakash VVS, Mittra R (2003) Characteristic basis function method: a new technique for efficient solution of method of moments matrix equation. *Microw Opt Technol Lett* 36(2):95–100
4. Tiberi G, Degiorgi M, Monorchio A, Manara G, Mittra R (2005) A class of physical optics-SVD derived basis functions for solving electromagnetic scattering problems. Antennas and propagation society international symposium, Washington, DC, July 2005
5. Lucente E, Monorchio A, Mittra R (2008) An iteration-free MoM approach based on excitation independent characteristic basis functions for solving large multiscale electromagnetic problems. *IEEE Trans Antenn Propag* 56(4):999–1007
6. Engheta N, Murphy WD, Rokhlin V, Vassiliou MS (1992) The fast multipole method (FMM) for electromagnetic scattering problems. *IEEE Trans Antenn Propag* 40(6):634–641

7. Chew WC, Jin J, Michielssen E, Song J (eds) (2001) Fast and efficient algorithms in computational electromagnetics. Artech House Inc., Boston
8. Farin G (1988) Curves and surfaces for computer aided geometric design: a practical guide. Academic, Boston
9. Delgado C, Mittra R, Cátedra F (2008) Accurate representation of the edge behavior of current when using PO-derived characteristic basis functions. *IEEE Antenn Wireless Propag Lett* 7:43–45
10. Pacheco PS (1997) Parallel programming with MPI. Morgan Kaufmann Publishers, Inc., San Francisco
11. Ergul O, Gurel L (2008) Efficient parallelization of the multilevel fast multipole algorithm for the solution of large-scale scattering problems. *IEEE Trans Antenn Propag* 56(8):2335–2345
12. Ergul O, Gurel L (2009) A hierarchical partitioning strategy for an efficient parallelization of the multilevel fast multipole algorithm. *IEEE Trans Antenn Propag* 57(6):1740–1750
13. Owens JD, Houston M, Luebke D, Green S, Stone JE, Phillips JC (2008) GPU computing. *Proc IEEE* 96(5):879–899
14. Sanders J, Kandrot E (2010) CUDA by example. Addison-Wesley Professional, Boston
15. Delgado C, Cátedra MF, Mitra R (2008) Application of the characteristic basis function method utilizing a class of basis and testing functions defined on NURBS patches. *IEEE Trans Antenn Propag* 56(3):784–790

Computational Electromagnetics  
Recent Advances and Engineering Applications  
Mittra, R. (Ed.)  
2014, VIII, 704 p. 427 illus., 192 illus. in color.,  
Hardcover  
ISBN: 978-1-4614-4381-0