IMPLEMENTATION OF CONSTANT BOUNDARY ELEMENTS FOR 2D POTENTIAL PROBLEMS ON GRAPHICS HARDWARE – GPU

Josué Labaki, labaki@fem.unicamp.br
Luiz Otávio Saraiva Ferreira, lotavio@fem.unicamp.br
Euclides Mesquita, euclides@fem.unicamp.br
Unicamp, State University of Campinas
Mendeleyev St., 200. Campinas – SP/Brazil.

Abstract. *There is a growing trend towards solving problems of computational mechanics by parallelization strategies. The more traditional approach is to implement the parallelization procedures on CPUs based on the MPI or OpenMP paradigms. Recent efforts have been made to implement computational tasks, which are amenable to parallelization on graphics hardware (GPU). Due to its architecture, the GPU is specially well-suited to address problems that can be expressed as data-parallel computations with high arithmetic intensity (the ratio of arithmetic operations to memory operations). One example of such problem is the Boundary Elements Method (BEM). This work addresses the implementation of the direct version of BEM for two-dimensional potential problems. For the present implementation constant boundary elements are used. According to the formulation of BEM, every term of both influence matrices (G_{ij} and H_{ij}) is independent of each other. In classical CPU serial implementations, these terms are calculated in a sequence of two loops: for the field point i and for the source point j. On the other hand, from the point of view of the GPU parallel processing paradigm, the calculation of every one of these terms can be assigned to a thread (GPU's elementary unit of calculation) and calculated simultaneously. The transposition of the influence equation to an algebraic linear system of equation also admits parallelization. The computational efficiency of distinct levels of parallelization is addressed. Standard Gaussian quadrature is applied to integrate each term of influence matrices. The code was developed on a NVidia CUDA programming environment and executed on a GeForce GTX 280 graphics card hosted by a regular AMD dual-core CPU. The accuracy and efficiency of the implemented strategies are investigated by solving a classical potential problem.*

Keywords*: High Performance Computing; Graphics Hardware; Boundary Elements Method*

1. INTRODUCTION

In the last three years, the edges of computing capability have been pushed by the emergence of General Purpose Graphics Processing Units (GPGPU). Around the end of 2006, a new technology of graphic devices was launched. This new generation of devices is not only dedicated to graphics computation, but is also capable of performing general-purpose calculations. Along with this technology, Application Programming Interfaces (APIs) were also launched, allowing the programmer to code the GPGPU in a higher level paradigm (Owens et al, 2007).

Graphics hardware was born as parallel computation hardware. Its high-bandwidth memories and its floating-point operations are significantly faster than ordinary CPUs and have driven attention of the scientific community. Methods of discretization, such as the Boundary Elements Method (BEM), whose parallel formulations have been explored for CPU clusters, now find in general-purpose GPU a new and promising alternative of implementation.

In the process of solution of a problem by BEM, several non-recursive numerical calculations have to be performed, which are good candidates to parallelization on graphics hardware. Many numerical integrations have to be done, a dense linear system has to be solved, and a couple of rectangular and square matrix-vector multiplications has to be performed.

This paper addresses the implementation of the Boundary Elements Method for two-dimensional potential problems on graphics hardware. The paper begins describing the formulation of the BEM. The classical serial implementation is overviewed. Next, the new technology of GPGPU is described in some details. It is shown why the GPU implementation is more efficient than its CPU counterpart and how the coding of non-graphical algorithms is treated. The fourth section shows how the BEM was approached in order to comply with the GPGPU philosophy. Finally, the presented implementation is used to solve a simple potential problem. Its performance is compared with an ordinary CPU serial code.

2. THE BOUNDARY ELEMENTS METHOD

The Boundary Elements Method (BEM) began to be developed by Cruse and Rizzo (1968) and Brebbia (1978), primarily as a discretized formulation of the Integral Equations and the Boundary Integral Equations, from the works of Fredholm and Helmholz (Courant and Hilbert, 1989; Arfken and Weber, 2005).

The BEM is part of the group of numerical methods which involve some discretization, i. e., a great, complex problem is divided into smaller problems which solution is more easily obtained. These solutions are them assembled in

order to form the solution of the greater problem. The nature of the problem being studied can be structural static or dynamic, thermal, electrical and electromagnetical, chemical, and so forth.

The solution of the problems by BEM goes through the following main steps:

(a) the discretization of the domain boundary by elements;

(b) the assignment of boundary parameters in terms of nodal values;

(c) the numerical integration over the elements;

(d) the assembly of an algebraic system of equations which represent the contribution of the solutions over the elements to the solution of the whole problem and

(e) the numerical solution of this final system of equations.

It is also possible to determine the solution at the domain of the problem from this analysis of the boundary. However, while in the Finite Elements Method this is done by interpolation, in BEM it is achieved by a technique of integration similar to the one which led to the solution at the boundary.

2.1 Formulation of BEM for potential problems

Consider a domain Ω_b enclosed by a boundary Γ_b (Fig. 1), in which the behavior of a quantity u($\boldsymbol{x}$) is described by the Laplace homogeneous equation, Eq. (1).

$$\nabla^2 u(\boldsymbol{x}) = 0 \tag{1}$$

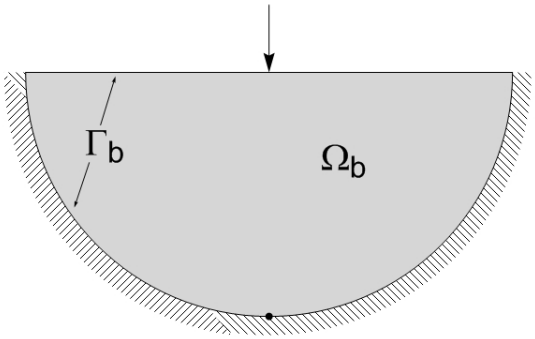

Figure 1. Domain Ω_b enclosed by a boundary Γ_b.

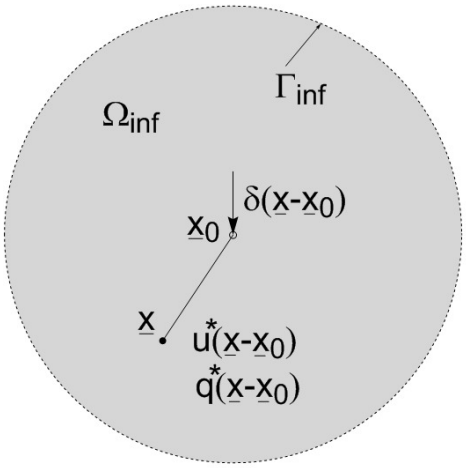

Figure 2. Particular case of auxiliary state: the fundamental solution.

Consider also a relation of reciprocity between two states u*($\boldsymbol{x}$, $\boldsymbol{x_0}$) and u($\boldsymbol{x}$), such as the Green's Second Identity (Brebbia and Dominguez, 1992; Kane, 1994).

$$\int_{\Omega_b} \left(u^*(\boldsymbol{x},\boldsymbol{x_0}) \nabla^2 u(\boldsymbol{x}) - u(\boldsymbol{x}) \nabla^2 u^*(\boldsymbol{x},\boldsymbol{x_0}) \right) d\Omega = \int_{\Gamma_b} \left(u^*(\boldsymbol{x},\boldsymbol{x}) \frac{\partial u(\boldsymbol{x})}{\partial \boldsymbol{n}} - u(\boldsymbol{x}) \frac{\partial u^*(\boldsymbol{x},\boldsymbol{x_0})}{\partial \boldsymbol{n}} \right) d\Gamma(\boldsymbol{x}) \tag{2}$$

In Eq. (2), the state u*($\boldsymbol{x}$, $\boldsymbol{x_0}$) is called *auxiliary state*, and is a fundamental concept in the Boundary Element Method. A classical example is the solution of the Laplace operator, Eq. (3), when the domain Ω is unbounded (Fig. 2):

$$\nabla^2 u^*(\boldsymbol{x},\boldsymbol{x}_0) = -\delta(\boldsymbol{x},\boldsymbol{x}_0) \tag{3}$$

This state refers to the application of a unitary concentrated load (Dirac Delta distribution) at the point $\boldsymbol{x_0}$, subjected to the condition that its effects vanishes at a point $\boldsymbol{x}$ infinitely far from $\boldsymbol{x_0}$. In other words,

$$\lim_{|\boldsymbol{x}-\boldsymbol{x}_0|\to\infty} u^*(\boldsymbol{x},\boldsymbol{x}_0) = 0 \tag{4}$$

The source-point $\boldsymbol{x_0}$, in which the auxiliary state is applied, as well as the field-point $\boldsymbol{x}$, where the effect of this load is measured, is another fundamental concept of BEM, as well as the derivative of the auxiliary state with respect to a normal direction $\boldsymbol{n}$ to be defined later:

$$\frac{\partial u^*(\boldsymbol{x},\boldsymbol{x}_0)}{\partial \boldsymbol{n}} = q^*(\boldsymbol{x},\boldsymbol{x}_0) \tag{5}$$

This particular case of auxiliary state is called the *fundamental solution* of the Laplace operator. The characteristics of this fundamental solution are:

(a) the auxiliary state is the solution of the Laplace operator at the unbounded domain Ω_{inf}, presenting a non-homogeneous term defined by a Dirac Delta and

(b) it satisfies the boundary condition stated by Eq. (4).

The fundamental solution $u^*(\boldsymbol{x}, \boldsymbol{x_0})$ presents the following mathematical property:

$$\int_{\Omega_b} \left(u(\boldsymbol{x}) \nabla^2 u^*(\boldsymbol{x},\boldsymbol{x}_0) \right) d\Omega = \int_{\Omega_b} -\left(u(\boldsymbol{x}) \delta(\boldsymbol{x},\boldsymbol{x}_0) \right) d\Omega = -u(\boldsymbol{x}_0) \tag{6}$$

Replacing (6) and (1) into (2) it is obtained:

$$\int_{\Omega_b} \left(u^*(\boldsymbol{x},\boldsymbol{x}_0) \underbrace{\nabla^2 u(\boldsymbol{x})}_{0} - u(\boldsymbol{x}) \underbrace{\nabla^2 u^*(\boldsymbol{x},\boldsymbol{x}_0)}_{-\delta(\boldsymbol{x}-\boldsymbol{x}_0)} \right) d\Omega = \int_{\Omega_b} \left(-u(\boldsymbol{x}) \left[-\delta(\boldsymbol{x}-\boldsymbol{x}_0) \right] \right) d\Omega = u(\boldsymbol{x}_0) \tag{7}$$

Equation (7) is applied to the right hand side of the Green's Second Identity (Eq. 2) in order to obtain the following Boundary Integral Equation:

$$+u(\boldsymbol{x}_0) = \int_{\Gamma_b} \left(u^*(\boldsymbol{x},\boldsymbol{x}_0) \frac{\partial u(\boldsymbol{x})}{\partial \boldsymbol{n}} - u(\boldsymbol{x}) \frac{\partial u^*(\boldsymbol{x},\boldsymbol{x}_0)}{\partial \boldsymbol{n}} \right) d\Gamma(\boldsymbol{x}) \tag{8}$$

Once the position of the source-point $\boldsymbol{x_0}$ is arbitrary, it can be collocated at the boundary of the problem. In this case, Eq. (8) becomes similar to the Somigliana identity, but for the Laplace operator:

$$C(\boldsymbol{x}_0) u(\boldsymbol{x}_0) = \int_{\Gamma_b} \left(u^*(\boldsymbol{x},\boldsymbol{x}_0) \frac{\partial u(\boldsymbol{x})}{\partial \boldsymbol{n}} - u(\boldsymbol{x}) \frac{\partial u^*(\boldsymbol{x},\boldsymbol{x}_0)}{\partial \boldsymbol{n}} \right) d\Gamma(\boldsymbol{x}) \tag{9}$$

This equation forms the basis of the classical BEM for potential problems and its formulation can be found in many text-books about the Boundary Elements Method. Equation (9) is an exact Boundary Integral Equation in which line integrals must be determined along a boundary like Γ_b in Fig. 1. The formulation of the BEM consists in the discretization of Eq. (9). According to this method, the boundary Γ_b is discretized by boundary elements Γ_e (Fig. 3), each one having normal vectors $\boldsymbol{n_e}$ pointing outwards the domain. The solution over the boundary elements are assumed to vary according to some pre-defined mathematical function $h_i(\boldsymbol{x})$:

$$u(\boldsymbol{x}) = \sum_i u_i h_i(\boldsymbol{x}) \; ; \qquad q(\boldsymbol{x}) = \sum_i q_i h_i(\boldsymbol{x}) \tag{10}$$

Replacing Eq. (10) into (9), results:

$$C(\boldsymbol{x}_0) u(\boldsymbol{x}_0) = \sum_i q_i \int_{\Gamma^e} u^*(\boldsymbol{x}_e,\boldsymbol{x}_0) h_i(\boldsymbol{x}_e) d\Gamma^e(\boldsymbol{x}_e) + \sum_i u_i \int_{\Gamma^e} q^*(\boldsymbol{x}_e,\boldsymbol{x}_0) h_i(\boldsymbol{x}_e) d\Gamma^e(\boldsymbol{x}_e) \tag{11}$$

2.2. Serial implementation of the BEM

Consider a two-dimensional problem discretized by N constant boundary elements, as it can be seen in Fig. 4b. According to the formulation based on constant elements, the central node of the element is taken to represent the whole element, in which the quantities *u* and *q* are taken as constants.

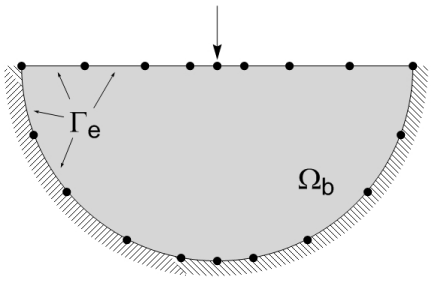

Figure 3. Discretization of the boundary Γ_b.

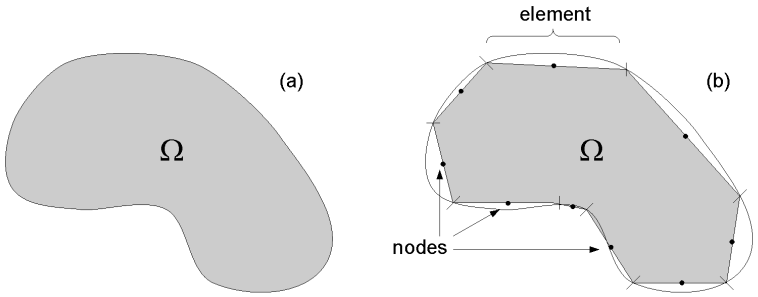

Figure 4. (a) Continuous problem and (b) problem discretized by boundary elements.

The point i (Eq. 11) is an arbitrary element in which the fundamental solution is applied, and Γ_e is the boundary of another element j. For constant elements, the multiplier $C(\boldsymbol{x_0})$ is always 0.5 (Brebbia, 1978). Considering that the quantities u and q are constants along the element j, they can be taken out of the integral. Equation (11) becomes then:

$$\frac{1}{2}u^i + \sum_{j=1}^{N}\left(\int_{\Gamma j} q^* d\Gamma\right)\cdot u^j = \sum_{j=1}^{N}\left(\int_{\Gamma j} u^* d\Gamma\right)\cdot q^j \tag{12}$$

It is usual to denote the integrals of Eq. (12) by influence coefficients, given by:

$$H^{ij} = \begin{cases} \widehat{H}^{ij} = \int_{\Gamma j} q^* d\Gamma;\ i \neq j \\ \int_{\Gamma j} q^* d\Gamma + \frac{1}{2};\quad i = j \end{cases} \quad \text{and} \quad G^{ij} = \int_{\Gamma j} u^* d\Gamma \tag{13}$$

Replacing (13) into (12) yields:

$$\sum_{j=1}^{N} H^{ij} u^j = \sum_{j=1}^{N} G^{ij} q^j \tag{14}$$

If the index i runs through all the N boundary elements, Eq. (14) becomes the system of algebraic equations given in Eq. (15), in which **H** and **G** are matrices with dimensions N × N, and $\boldsymbol{u}$ and $\boldsymbol{q}$ are vectors N × 1:

$$\mathbf{H}\cdot\mathbf{u} = \mathbf{G}\cdot\mathbf{q} \tag{15}$$

In a well-posed problem, each element has a known u and an unknown q or vice-versa. Hence, every problem will have N known variables and N unknowns. Equation (15) has to be rearranged in order to separate the unknowns to the same side of the equation (Eq. 16).

$$\mathbf{A}\cdot\mathbf{x} = \mathbf{B}\cdot\mathbf{b'} \tag{16}$$

In Eq. (16), matrices **A** and **B** are formed by a combination of columns of **H** and **G** according to the problem's boundary conditions, i. e., according to which values of u or q are known in a given element i. The vector $\boldsymbol{x}$ contains the unknowns of the problem and the vector **b'** contains the boundary conditions. The matrix B and the vector **b'** are multiplied to obtain the following final system of algebraic equations:

$$\mathbf{A}\cdot\mathbf{x} = \mathbf{b} \tag{17}$$

Equation (17) is solved to determine the unknowns of the problem at the prescribed boundary. Once determined u^j and q^j for every element j, it is possible to determine the quantities u and q for any internal point $\boldsymbol{p}$ of the domain from Eq. (11). Now that the point $\boldsymbol{p}$ belongs in the domain of the problem, the value of the constant $C(\boldsymbol{p})$ is 1 (Brebbia, 1978). Thus, Eq. (11) becomes:

$$u^p = \sum_{j=1}^{N} G^{pj} q^j - \sum_{j=1}^{N} \widehat{H}^{pj} u^j \tag{18}$$

In the serial implementation, the terms H^{ij} and G^{ij} (Eq. 14) are calculated in a sequence of two loops. The iterator *i* represents the collocation of the source-point on different elements. The iterator *j* varies representing the element over in which the integration is performed. Depending on the method of integration adopted, an additional inner loop, responsible for the numerical integration, will have to be carried out for each pair *i-j*. For example, for the integration by Gaussian Quadrature, an additional loop *k* over the Np integration nodes will be necessary (Davis and Rabinowitz, 2007).

In a very simple programming scheme, once full the matrices **H** and **G** are determined, the transition between Eq. (15) and (16) is performed. A loop of N terms fills the vectors **x** and **b'** with data from **u** and **q** according to the boundary conditions. In this loop, the columns of **A** and **B** are created, with data from **H** and **G**, according to the boundary conditions. In the sequence the linear system of Eq. (17) is solved. A large variety of numerical methods are described by the literature for the solution of this sort of system (Ruggiero and Lopes, 1996).

A new double loop in *p* and *j* fills the new rectangular matrices $\mathbf{H}^{pj}$ and $\mathbf{G}^{pj}$. The multiplication of these matrices by the just determined vectors **u** and **q** results in the solution of *u* for the internal points.

In this section, the Boundary Elements Method for the study of potential problems was displayed. A simple and classical serial implementation was summarized. Next, the technology of computation on graphics hardware will be presented.

3. PARALLEL COMPUTING ON GRAPHICS HARDWARE

Ordinary Central Processing Units (CPUs) must be capable of dealing with a variety of tasks demanded by a computer. Among them, there are recursive, adaptive, and interdependent problems, which demand a large amount of the computation resources to be dedicated to communication of data and control. On the other hand, graphics calculations such as pixel shading, vertex transformation and rasterization are tasks that require little control and communication, when compared to the volume of calculations. Because of that, graphics hardware has been developed since its beginning as parallel computation devices. They are specially well-suited to address problems that can be expressed as data-parallel computations with high arithmetic intensity (the ratio of arithmetic operations to memory operations) (NVidia, 2008).

For example, a typical card launched in the end of 2006 is the NVidia GeForce 8800 model (TechReport, 2006b). This graphics card contains 128 calculation units (called multiprocessors), distributed among 8 vector processors. This architecture is similar to the one found in clusters of CPU, only confined in a single hardware device. Because of its architecture, this family of graphics cards requires a single instruction–multiple data programming paradigm (SIMD). For this family of cards, in regular computations such as matrix multiplication, performances 100 times higher than the CPU have been reported (Cooperman and Kaeli, 2008; Ohshima et al, 2007).

The model GeForce 8800, launched by NVidia along with its CUDA API, is part of the new technology of general-purpose programmable graphics processing units, the GPGPUs. The company ATI was the first to introduce this technology (TechReport, 2006a). Its graphics cards are programmable since the Radeon R600 model launched in May, 2007 (TechReport, 2007). Its respective API is called CTM (Close-to-the-Metal). In the present work, NVidia's API, CUDA, was adopted.

CUDA (Computer Unified Device Architecture) is an API (Application Programming Interface) with which NVidia graphics cards can be programmed to perform non-graphics tasks. It is a low level language, because it requires the programmer to explicitly allocate and free memory, to declare data copies, to chose parameters of parallelism, and so forth. It is essentially an extension of the C programming language, with the addition of function type qualifiers, variable type qualifiers, kernel execution directives and some additional built-in variables. CUDA is multiplatform and can work with all NVidia card architectures (NVidia, 2008).

In CUDA programming, the concepts of thread, thread block and grid are fundamental. Thread is a virtualized CPU, the basic execution unit: it is the component responsible for executing a given instruction (the kernel) over a single data. Multiple threads may work in parallel executing the same kernel over a set of different data. Thread blocks are used to spread the threads between the various multiprocessors of the graphics card. Grids are used to spread the data of the problem among thread blocks. A thread block can be a one-, two- or three-dimensional array of threads, and CUDA offers variables with which the index of every thread inside its block can be recovered. The same applies to grids with respect to their blocks.

A GPU has several multiprocessors, each one of them capable of dealing with many blocks simultaneously. Each thread block, in turn, admits the execution of a limited number of threads at the same time. This number is called *warp*, and use to be of 32 threads. The number of multiprocessors in a GPU, such as the number of thread blocks with which each one of them can deal with and the warp size depends on the card's model. For example, the model GTX 280 has 30 multiprocessors, each one of them capable of dealing with 8 blocks simultaneously. Altogether, this card can execute the same kernel simultaneously over (30 multiprocessors) × (8 thread blocks per multiprocessor) × (32 threads of the warp) = 7680 data.

It is up to the programmer to decide in which way the data of the problem will be divided in terms of blocks and grids. This is a tough decision which implies directly on the efficiency of the program. Recently, an application has

been developed, to determine these parameters by metaprogramming (Klöckner and Hesthaven, 2008).

Graphics hardware holds complex memory architecture. The most important of them, the *global*, may have up to 2 GB of memory, in the newest cards. The data placed in this memory are available to all the threads of a grid. Each thread block has its own *shared memory*, of only 16 kB, but the access time is up to 600 times faster than of the global memory. However, only the threads of the given block are allowed to access their block's shared memory. Furthermore, each thread has its own registers, accessed only by the thread itself. The graphics card also has the *constant* and *texture read-only* cache memories, devoted to specific purposes in the graphics calculation (NVidia, 2008). Despite all this graphics hardware memories, a CUDA program also has to deal with the ordinary CPU RAM memory, as every classical low-medium level program does.

The execution of GPU programs requires a sophisticated manipulation of data between all these memories. All the vectors and matrices that might be accessed by the threads have to be allocated in the RAM memory of the CPU that hosts the graphics card, and also allocated in the GPU's global memory. Only pointers to these vectors are passed as arguments to the kernels. If it is necessary to access a set of data repeatedly inside a block, it might be also necessary to define a space inside its shared memories, or even in the threads' registers.

At the end of the execution of a kernel, the data calculated by the threads are saved in the memory allocated in the GPU. It is necessary to copy back this data to the CPU's memory so that they can be printed, read, saved, etc.

All memory manipulation expends some processor clock cycles. A precise, fair benchmark of processing time between CPU and GPU will be achieved only if it also involves the time the GPU consumes to perform these memories operations. The following section will report how the programming concepts of GPGPU were approached in the present implementation of the Boundary Element Method.

4. IMPLEMENTATION

In section 2.2, a classical serial algorithm of the implementation of BEM was summarized. The part of that algorithm referring to the calculation of the matrices **H** and **G**, i. e., the calculation of the influence coefficients H^{ij} and G^{ij} (Eq. 14), is one of the easiest cases to be coded in a parallel algorithm, if the formulation of discontinuous elements is adopted.

The matrices **H** and **G** are allocated as vectors of size N^2 and passed as argument to the kernel that will perform the calculations of their terms H^{ij} and G^{ij}. The data of the problem, like the coordinates of the nodes and the incidence of the elements are passed as arguments as well.

A number of threads is chosen in order to perform the calculations. In the present implementation, these threads are distributed among two-dimensional thread blocks of 22 × 22 threads. The number 22 is chosen because 22 × 22 is the biggest dimension a square block can have (23 × 23 > 512, maximum number of threads per block). The size of a two-dimensional grid is calculated so as to contain as many blocks as needed to accommodate the N^2 terms of **H** and **G**.

Figure 5 illustrated the sizes of grids and blocks for a reduced example. In this example, matrices **H** and **G** will have dimensions of N × N = 6 × 6. The thread blocks were defined as containing 4 × 4 threads. From Fig. 5, it is observed that the grid will then be calculated to contain 2 × 2 blocks, in a total of 8 × 8 = 64 threads. Even so, only 6 × 6 = 36 out of the 64 threads will perform the calculations of H^{ij} and G^{ij}. The darkened cells in Fig. 5 represent the terms that will perform some calculation, while the blank cells represent the threads that were created, but left inactive.

Two 22 × 22 sub-matrices (of **H** and **G**) are allocated at each thread block's shared memory. The calculation of H^{ij} and G^{ij} performed by these threads are initially stored in these sub-matrices. After all the block's threads have ended their calculations, this data are finally copied to the vectors **H** and **G** allocated at the GPU's global memory.

In the calculation of internal points, a similar procedure is employed. The same size of blocks is used. The difference is that, as the number of internal points might be different of the number of elements, the matrices might present more or less rows than columns, and therefore the grids will also have more or less thread blocks in its "vertical" direction.

In parallel execution, instead of two chain loops, each thread of the whole grid will have its own index *ij*. Based on this index, the threads will be able to univocally determine, from the data of the problem (node coordinates, element incidence, etc.) the parameters needed to perform the integration shown by Eq. (13). In this paper, four-node Gaussian Quadrature is adopted to perform this integration. The four terms loop referring to the Gaussian Quadrature is performed sequentially by each thread.

A kernel dedicated to perform the transition between Eqs. (15) and (16) was written. According to the boundary conditions, the thread of index *ij* switches or not the terms H^{ij} and G^{ij} and assembles the vectors **x** and **b'** from **u** and **q**. Hence, not only several terms of **x** and **b'** are filled in one single execution step, but also several columns of **H** and **G** are switched at the same time.

The same procedure is employed to calculate the matrices $\mathbf{H}^{pj}$ and $\mathbf{G}^{pj}$ of Eq. (18). In this case, only the number of activated threads is different.

The remaining calculations, as the multiplication of matrices by vectors and the solution of the linear system expressed by Eq. 17 are performed in serial execution by the CPU. There are initiatives to implement methods for solution of linear systems in GPGPU, but the present available implementations are still immature or ill-documented.

The present implementation was applied to solve an elementary potential problem by BEM, and the results are reported in the next section.

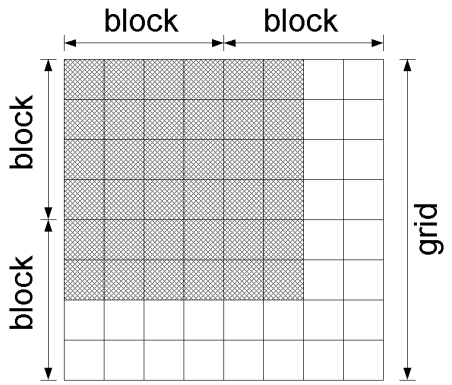

Figure 5. Reduced example of a grid of thread blocks.

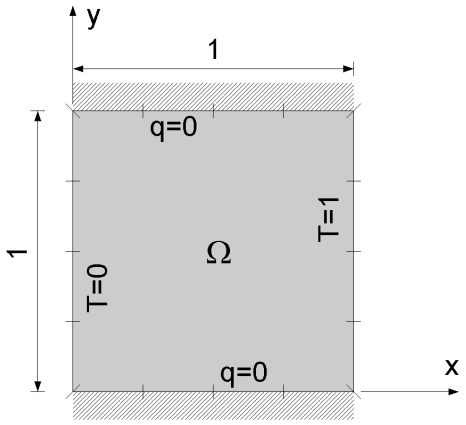

Figure 6. Two-dimensional square plate of unitary edge.

5. RESULTS

The present implementation is capable of dealing with two-dimensional problems, discretized by constant boundary elements. As input data, it must be provided: a vector containing the coordinates (x^i, y^i) of the vertices of the N elements; a vector containing the relationship of incidence of the elements (which nodes belong in each elements); a vector containing the type (*u* or *q*) and the value of the boundary conditions, and a vector containing the coordinates (x^p, y^p) of the internal points.

The thermal problem depicted by Fig. 6 was treated. The problem refers to a square plate of unitary edge. Each edge is discretized by N/4 elements of same length. As boundary conditions, all the elements of the left border have zero temperature, all the elements of the right border have temperature 1, and the remaining borders are insulated (q = 0). This problem has a closed form analytical solution given by T(x) = x, according to the given system of coordinated (Fig. 6).

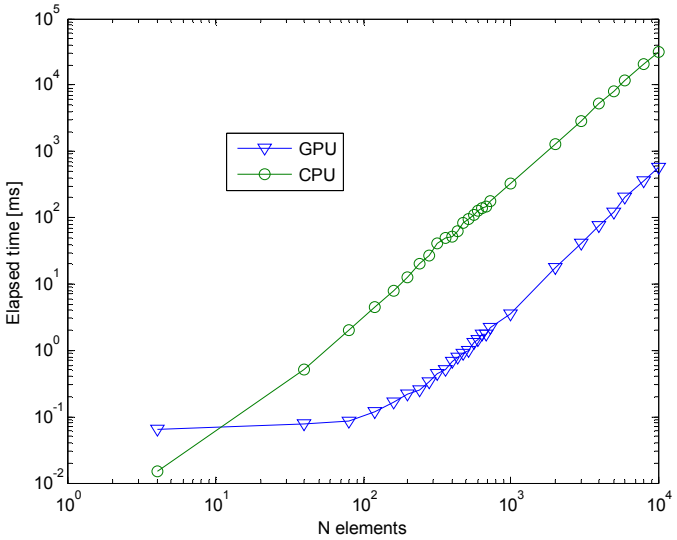

Figure 7. Time spent by the GPU and the CPU to calculate **H** and **G**.

The time consumed to fill the matrices **H** and **G** of Eq. (14) was measured to several numbers of elements N. In the GPU, this time corresponds to the time spent by the specific kernel that calculates these matrices. These times are compared to a serial code written in pure C language. In the CPU, this time corresponds to the time spent by the specific function that performs these calculations. Figure 7 shows the elapsed times for values of N between 4 and 10,000 elements.

At the beginning of the graphic, it can be observed that there is a number of elements before which the use of CPU is more advantageous than the GPU. The reason to that is that, in order to execute the kernel that calculates **H** and **G** on the GPGPU, a few allocations and copies of memory are needed, which are not necessary in the CPU. This allocation time is rather short and depends little on the number of elements N, the increase of N causes it to dissolve in the total execution time of the kernel.

Beyond this point, the superiority of performance of the GPU is observed. In the final experiment, in which a problem of 10,000 elements was considered, the GPU obtained the matrices **H** and **G** in a time 56.8 times shorter than the CPU.

The present study also compared the time consumed to introduce the boundary conditions, that is to map the vectors **u** and **q** of Eq. 15 into the vectors **x** and **b'** of Eq. 16, that is to switch the columns between **H** and **G** according to the boundary conditions. The comparison is shown by Fig. 8.

It is again observed that there is a certain number of elements, from which the use of GPU is worth. The reason is the same: there is memory handling required before any calculation can commence. Because the calculation of the distribution of vectors and transposition of columns of matrices is much simpler than the calculation of the matrices **H** and **G**, this problem has smaller arithmetic intensity than the prior. For this reason, the use of GPU is advantageous only from a number of elements bigger than in the previous problem.

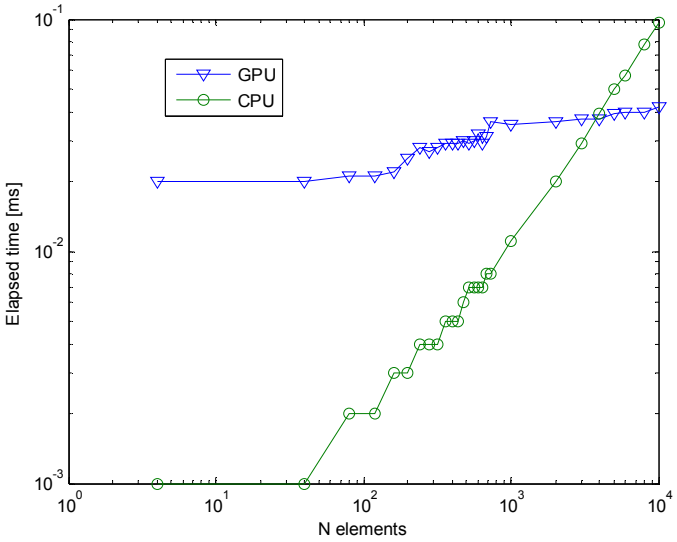

Figure 8. Time spent by the GPU and the CPU to distribute the vectors **u** and **q** among **x** and **b'**, and switch the columns between **H** and **G**.

The performance of GPU versus CPU in the calculation of internal points was also investigated. A mesh of N_{ptint} equally-spaced internal points was spread inside the domain of the same problem of Fig. 8. Figure 9 shows the distribution of the internal points when N_{ptint} is 9.

Figure 10 reports the time spent by the GPU and the CPU to calculate the matrices $\mathbf{H}^{pj}$ and $\mathbf{G}^{pj}$ (Eq. 18) as the number of internal points varies. The number of boundary elements was fixed in 1,600 elements. It is observed a large superiority of the GPU in numerical efficiency in this calculation. In the final experiment, in which 3,600 internal points were considered, the GPU obtained the matrices $\mathbf{H}^{pj}$ and $\mathbf{G}^{pj}$ in a time 141.6 times shorter than the CPU.

Finally, a complete problem was solved. In this experiment, N boundary elements are used. The solution goes from calculating **H** and **G** and ends at the calculation of the temperature at N_{ptint} = 16 internal points. The aim of this experiment is to relate the time spent by the GPU in global memory operations to the time spent with the global processing.

As it was already told, the solution of the linear system and the matrix multiplication in the present implementation were performed by classical serial algorithms, in both the CPU and the GPU. Once this time is the same in both cases, it is not shown in the next results. Figure 11 shows the execution time for values of N between 4 and 10,000 elements.

Unlike the experiments shown in Fig. 7, in which the memory operations of the kernel was compared with the kernel's own calculations, the solution of the present complete problem also involved memory operation between CPU and GPU and argument passing between kernels. Therefore, from the point of view of the complete problem, the present implementation of BEM has reduced arithmetic intensity when compared to the calculation of **H** and **G** alone. For this reason, the use of GPU is more advantageous from a number of elements bigger than what was observed in Fig. 7.

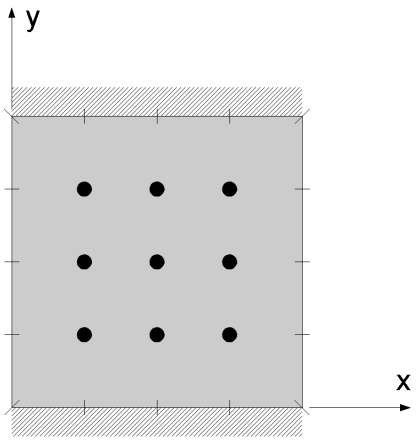

Figure 9. Distribution of internal points inside the domain of the square plate.

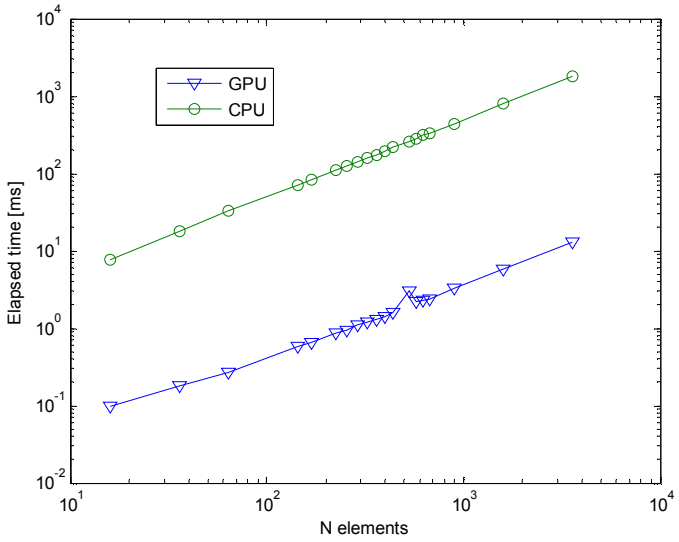

Figure 10. Time spent by the GPU and the CPU to calculate $\mathbf{H}^{pj}$ and $\mathbf{G}^{pj}$.

Even so, a superior performance of the GPU over the CPU is observed. In the final experiment, where 10,000 elements were considered, the graphics hardware solved the problem – both the solution at the boundary and at the 16 internal points – in a time 13 times shorter than the CPU.

A last case of complete problem was solved, with a much larger number of internal points. Ten thousand of boundary elements and internal points were used. The time consumed by the CPU, except to matrix multiplications and solution of the linear system, was 64,776.865 ms. The time consumed by the graphics card to solve the same problem was 3,200.773 ms – 20.24 times lower.

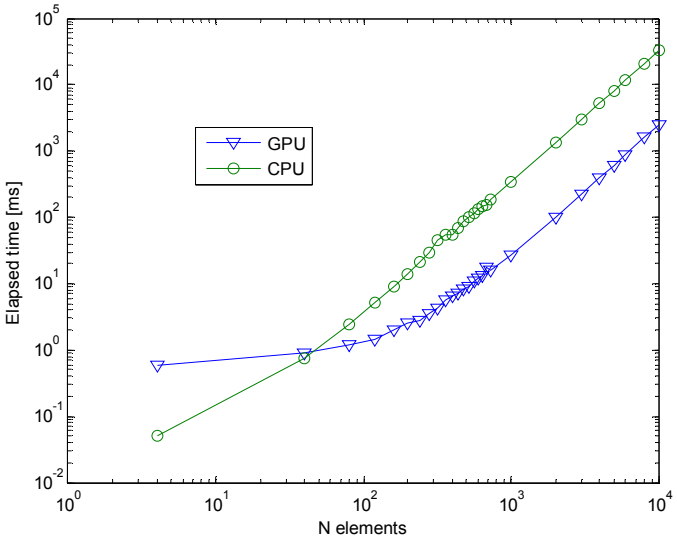

Figure 11. Times of execution to solve a complete problem.

6. CONCLUSION

This paper has described the implementation of the Boundary Elements Method for two-dimensional potential problems on graphics processing devices. A classical serial implementation was rewritten under the SIMD parallel programming paradigm.

The paper reports the performances of GPU and CPU on dealing with three important steps of BEM: the calculation of the influence matrices, the rearrangement of these matrices in the form of a system of equations $\mathbf{A}\cdot\mathbf{x} = \mathbf{b}$, and in the

calculation of the matrices of influence of internal points. It was observed that the point from which the GPU presents better performance than the CPU is function of the arithmetic intensity of each problem. In all the cases, however, the graphics hardware has shown to be more numerically efficient than the CPU with increasing number of elements and internal points.

Many improvements can be added to the present implementation. It is well known that the time spent in the solution of a linear system of algebraic equations varies dramatically with the size of the system. In the last experiment of this work, in which the linear system had dimension of $10^4 \times 10^4$, the time consumed to solve it was 101,839.016 ms: over 30 times the time consumed to perform all the remaining calculations of the problem. Therefore, it is hoped that the development of a reliable CUDA linear systems solver may boost the analysis of problems by BEM in GPU to even more attractive levels.

7. REFERENCES

Arfken, G. B., Weber, H. J., 2005, "Mathematical Methods for Physicists". Academic Press, Orlando, 1200p.

Brebbia, C. A., 1978, "The Boundary Element Method". Pentech Press, London.

Brebbia C. A., Dominguez J., 1992, "Boundary Elements - An Introductory Course". 2nd Edt., Computational Mechanics, Southampton.

Courant, R., Hilbert, D., 1989, "Methods of Mathematical Physics". Wiley-Interscience, New York, 560 p.

Cooperman, G., Kaeli, D., 2008, "GPGPU Programming – Syllabus". 28 November 2008. <http://www.ccs.neu.edu/course/csu610/#syllabus>

Cruse, T. A., Rizzo, F. J., 1968, "A direct formulation and numerical solution of the general transient elatodynamic problem". I. I. J. Math Analysis, vol. 22, pp. 244-259.

Davis, P. J., Rabinowitz, P., 2007, "Methods of Numerical Integration". 2nd Edt. Dover Publications. Mineola.

Kane, J. H., 1994, "Boundary Element Analysis in Engineering Continuum Mechanics". Prentice Hall Englewood Cliffs.

Klöckner, A., Hesthaven, J. S., 2008, "Metaprogramming Graphics Processors from High-Level Languages". 28 November 2008 <http://mathema.tician.de/entry/dam>

NVidia., 2008, "NVidia CUDA – Compute Unified Device Architecture – Programming Guide". NVidia Corporation, Santa Clara.

Ohshima, S., Kise, K., Katagiri, T., Yuba, T. 2007, "Parallel Processing of Matrix Multiplication in a CPU and GPU Heterogeneous Environment". Graduate School of Information Systems The University of Electro-Communications, Tokyo.

Owens, J. D. et al, 2007, "A Survey of General-Purpose Computation on Graphics Hardware", Computer Graphics, 26, 1, pp. 80-113.

TechReport, 2007. "AMD's Radeon HD 2900 XT graphics processor: R600 revealed". 23 May 2009, <http://techreport.com/articles.x/12458>

TechReport, 2006a. "ATI dives into stream computing and makes a splash". 23 May 2009, <http://techreport.com/articles.x/10956>

TechReport, 2006b. "Nvidia's GeForce 8800 graphics processor The green team reinvents its own reality and rattles ours". 23 May 2009, < http://techreport.com/articles.x/11211 >

Ruggiero, M. A. G., Lopes, V. L. R., 1996, "Cálculo Numérico – Aspectos Teóricos e Computacionais", Editora Pearson Education, São Paulo, 410 p.

8. RESPONSIBILITY NOTICE

The authors are the only responsible for the printed material included in this paper.