

THE BEM ON GENERAL PURPOSE GRAPHICS PROCESSING UNITS (GPGPU): A STUDY ON THREE DISTINCT IMPLEMENTATIONS

Josué Labaki^{1,a)}, Euclides Mesquita^{2,b)}, Luiz Otávio Saraiva Ferreira^{3,c)}

^{1,2,3)}Department of Computational Mechanics, School of Mechanical Engineering,
State University of Campinas – UNICAMP
. 13083-970. Campinas, SP, Brazil.

^{a)}labaki@fem.unicamp.br, ^{b)}euclides@fem.unicamp.br, ^{c)}lotavio@fem.unicamp.br

Keywords: High Performance Computing, Graphics Hardware, Boundary Elements Method.

Abstract. This work addresses the implementation of the direct version of the Boundary Element Method (BEM) for two-dimensional problems on general-purpose graphics hardware (GPGPU). 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), such as the BEM. The performance of the GPU compared to the CPU is investigated for three different BEM formulations: potential and elastostatic problems with constant elements, and for potential problems with quadratic elements. The arithmetic intensity of these implementations is investigated and related to the GPU performance. Gains of performance from one to two orders of magnitude were observed with the use of graphics hardware.

Introduction

In the last years, graphics cards (GPUs) have been investigated for the implementation of non-graphical programs. Early researches managed to use the GPU for non-graphical purposes through the so-called graphics APIs (Application Programming Interfaces), like OpenGL [1] or DirectX [2]. Around the end of 2006, however, a new technology of graphic devices was launched, the general-purpose GPU (GPGPU). This new generation of devices is not only dedicated to graphics computation, but it is also capable of performing general-purpose data processing. Along with this technology a new API called CUDA (Compute Unified Device Architecture) has been launched by the NVidia™ Corporation for this new generation of GPUs [3]. CUDA is a programming language that allows the programmer to code the GPGPU in a higher level paradigm, compared to the former graphics-dedicated APIs such as OpenGL and DirectX [3,4].

Graphics hardware was born as a parallel computation device. Its high-bandwidth memories and its floating-point operations are significantly faster than ordinary CPUs and have called attention of the scientific community. Large scale computational tasks, whose parallel formulations have been explored for CPU clusters, now find in general-purpose GPU a new and promising alternative of implementation.

One example of a large-scale computational task is the Boundary Element Method (BEM). 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 describes the implementation of the Boundary Element Method on graphics hardware. The present article describes a 2D potential problem using constant and quadratic boundary elements as well as a 2D elastostatic problem using constant elements.

The paper begins describing a formulation of BEM for general types of problem and order of elements. Next, the new technology of GPGPU is briefly described. The fourth section shows how the BEM was approached in order to comply with the GPGPU philosophy. Finally, the presented implementations are used to solve representative potential and elastostatic problems with constant and quadratic elements. Their performance is compared with an ordinary CPU serial code.

The Boundary Element Method

The Boundary Element Method (BEM) is a consolidated numerical method for solution of differential equations. The method is based on the Boundary Integral Equation expressed by Eq. (1).

$$C(\mathbf{x}_0)u(\mathbf{x}_0) = \int_{\Gamma_b} u^*(\mathbf{x}, \mathbf{x}_0)q(\mathbf{x}) d\Gamma(\mathbf{x}) - \int_{\Gamma_b} u(\mathbf{x})q^*(\mathbf{x}, \mathbf{x}_0) d\Gamma(\mathbf{x}) \quad (1)$$

In Eq. (1), $u^*(\mathbf{x}, \mathbf{x}_0)$ and $q^*(\mathbf{x}, \mathbf{x}_0)$ are fundamental solutions which depends on the type of problem that are being analyzed [5]. For potential problems, the quantities $u(\mathbf{x})$ and $q(\mathbf{x})$ represent the potential and flux over the boundary of the domain. For elastostatic problems these terms represent displacement and tractions, respectively.

Consider a problem discretized by N boundary elements, or M nodes, and D degrees of freedom per node. For 2D elastostatics $D=2$ and for potential problems, $D=1$. A discretized version of Eq. (1) can be written as:

$$\sum_{j=1}^M \left(\delta_{ij}C_i + \int_{\Gamma_j} \phi_k q^* d\Gamma_j \right) \cdot u^j = \sum_{j=1}^M \left(\int_{\Gamma_j} \phi_k u^* d\Gamma_j \right) \cdot q^j \quad (2)$$

In Eq. (2), ϕ_k are shape functions with $k=1,m$, in which m is the number of nodes per element. For constant elements $m=1$ and for quadratic elements $m=3$. Equation (2) can be rewritten as $[H]\{u\}=[G]\{q\}$, in which $[H]$ and $[G]$ are known influence matrices [5]. In a simple implementation, according to the boundary conditions of the problem, the columns of $[H]$ and $[G]$ are passed from one side to the other of the equation $[H]\{u\}=[G]\{q\}$. Once all unknowns are passed to the left-hand side, the system $[A]\{x\}=[B]\{b'\}$ is obtained. In this system, $\{x\}$ contains all the unknowns of the problem and $\{b'\}$ contains the boundary conditions. The matrix $[B]$ and the vector $\{b'\}$ are multiplied to obtain the following final system of algebraic equations:

$$[A]\{x\} = \{b\} \quad (3)$$

Equation (3) is solved to determine the unknowns of the problem at the prescribed boundary nodes. Once the quantities u^j and q^j are determined for every node j , it is possible to determine the quantity u for any internal point of the domain [5].

Parallel Computing on GPGPUs

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, the architecture of a graphics card (GPU) dedicates most of the chip's area to calculation, and only a small area for control and communication. A modern NVidia GPU presents a single instruction unit and is divided in terms of streaming multiprocessors (SMs), each composed by a number of streaming processor cores (SPs). This particular architecture makes the GPUs specially well-suited to address problems that can be expressed as data-parallel computations with high arithmetic intensity [6].

CUDA (Computer Unified Device Architecture) is an API with which NVidia™ graphics cards can be programmed to perform non-graphics tasks. It requires the programmer to explicitly handle memory allocations, to declare data copies and to choose parameters of parallelism, for example. It comprises function type qualifiers, variable type qualifiers, kernel execution directives and some additional built-in variables specially designed to the GPU's many-cores programming paradigm.

A CUDA application experiments a CPU-GPU complimentary execution, as part of its code is run in the CPU ("host") and part in the GPU ("device"). It consists of a serial host program containing memory manipulation between the CPU and the GPU global memories and one or more *kernel* launches, which are Single Instruction Multiple Threads (SIMT) algorithms.

In CUDA programming, the concepts of thread, thread block and grid are fundamental. A thread is one of the many components responsible for executing a given instruction (the kernel) over a single data. Multiple threads work in parallel executing the same kernel over a set of different data. Threads are divided in thread blocks, each of which is run by each SM of the GPU. Threads within a block are able to share data through the SM's shared memory, and they can be synchronized at a certain point of their execution. Thread blocks are grouped in grids, which spread them among all the SMs of the GPU. Threads from different blocks cannot communicate nor be synchronized, but they all have common access to the GPU global memory. Moreover, threads possess private local memories and register space.

A thread block can be organized as 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. Analogously, the grids may be one-, two- or three-dimensional arrays of blocks. The thread blocks within the grids may also be identified by means of indices. An optimal performance can be achieved by the correct set up of blocks and grids.

The following section reports how the programming concepts of CUDA programming were approached in the present parallel implementations of the BEM.

Implementation of BEM on the GPGPU

Constant elements. The solution is divided in four main steps. The first step is the assembly of matrices [H] and [G]. The second step is the introduction of the boundary conditions. The third step is the solution of the linear equation system. The fourth step is the calculation of values at internal points. Steps 1,2 and 4 are parallelized and performed in the GPU. Step 3 is performed in the CPU as a serial procedure.

The algorithms used to the calculation of the matrices [H] and [G] using constant elements allow a straightforward parallel implementation, because all the terms inside these matrices are independent of each other. The determination of each element is assigned univocally to independent threads that execute in parallel. Experiments have shown that the ideal dimensions for thread blocks are 22×22 threads for potential problems and 4×4 threads for elastostatic problems. In both cases, the size of a two-dimensional grid is calculated so as

to contain as many blocks as needed to accommodate the terms of [H] and [G]. In this paper, four-node Gaussian Quadrature is adopted to integrate the influence coefficients. The four terms loop referring to the Gaussian Quadrature is performed sequentially by each thread.

A kernel dedicated to introduce the boundary conditions, that is, to perform the transition between $[H]\{u\}=[G]\{q\}$ and $[A]\{x\}=[B]\{b'\}$ was written. The remaining calculations, as the multiplication of matrices by vectors and the solution of the linear system $[A]\{x\}=\{b\}$ are performed in serial execution by the CPU. There are initiatives to implement methods for solution of linear systems in CUDA, but the present available implementations are still immature or ill-documented.

Implementation of continuous quadratic elements. For continuous elements, a given term H_{ij} can be composed by the sum of two terms of influence, h_{im} and h_{in} . If two different units of calculation – threads – of indices im and in are assigned to calculate h_{im} and h_{in} , they will not be able to write their answers in the address H_{ij} of [H] at the same time and this issue ought to be worked out carefully. In the present implementation it is assumed that every term of the matrix [G] is independent of each other.

This work uses a simple approach to handle the problem of two threads, which are required to write simultaneously in the same address of matrix [H]. Consider two terms of influence h_{im} and h_{in} , supposed to be written in the address H_{ij} of matrix [H]. The index i ($i=1, 2N$) is the index of a line of [H], and j is associated to a column of [H]. As $H_{ij} = h_{im} + h_{in}$, is true for any index i , that is for every line, although not for ever column in the line, it can be seen that this summation is independent of the index i , and therefore this operation can be performed in parallel for different values of i . In terms of matrix notation, this means that the calculation of an entire column of [H] (several different lines with different indices i) can be performed in parallel.

Consider a reduced example in which, $N = 2$, and therefore [H] and [G] are of size $2N \times 2N = 4 \times 4$ and $2N \times 3N = 4 \times 6$. For this example the matrices [H] and [G] are:

$$[H] = \begin{bmatrix} h_{11} + h_{13} & h_{12} & h_{14} + h_{16} & h_{15} \\ h_{21} + h_{23} & h_{22} & h_{24} + h_{26} & h_{25} \\ h_{31} + h_{33} & h_{32} & h_{34} + h_{36} & h_{35} \\ h_{41} + h_{43} & h_{42} & h_{44} + h_{46} & h_{45} \end{bmatrix}_{2N \times 2N} \quad [G] = \begin{bmatrix} g_{11} & g_{12} & g_{13} & g_{14} & g_{15} & g_{16} \\ g_{21} & g_{22} & g_{23} & g_{24} & g_{25} & g_{26} \\ g_{31} & g_{32} & g_{33} & g_{34} & g_{35} & g_{36} \\ g_{41} & g_{42} & g_{43} & g_{44} & g_{45} & g_{46} \end{bmatrix}_{2N \times 3N} \quad (4a,b)$$

Notice that the indices m and n of h_{im} and h_{in} go from 1 to $3N = 6$, while $i=1, 2N=4$. This matrices, in particular, could be calculated in a loop of $3N = 6$ indices. In each step, all the terms of a column can be calculated simultaneously.

Figure 1 depicts this example. Darkened cells represent the terms that are being added to [H] and [G] in the present calculation step. Observe that, in this approach, more that one value is added to the same address of [H], but not concurrently. For instance, in the first iteration, the term h_{11} is added to H_{11} , and in the third iteration, the term h_{13} is added to the same address. In the present implementation, the matrices [H] and [G] are allocated as vectors of size $2N \times 2N$ and $2N \times 3N$ 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. As the level of parallelization is one-dimensional (i. e., only the terms of a single column are to be calculated simultaneously), these threads are distributed among one-dimensional thread blocks of 64

threads. The literature indicates that 64 is the ideal number of threads for most problems when one-dimensional blocks can be employed [6]. The size of a one-dimensional grid is calculated so as to contain as many blocks as needed to accommodate at the $2N$ terms that a column of $[H]$ or $[G]$ presents.

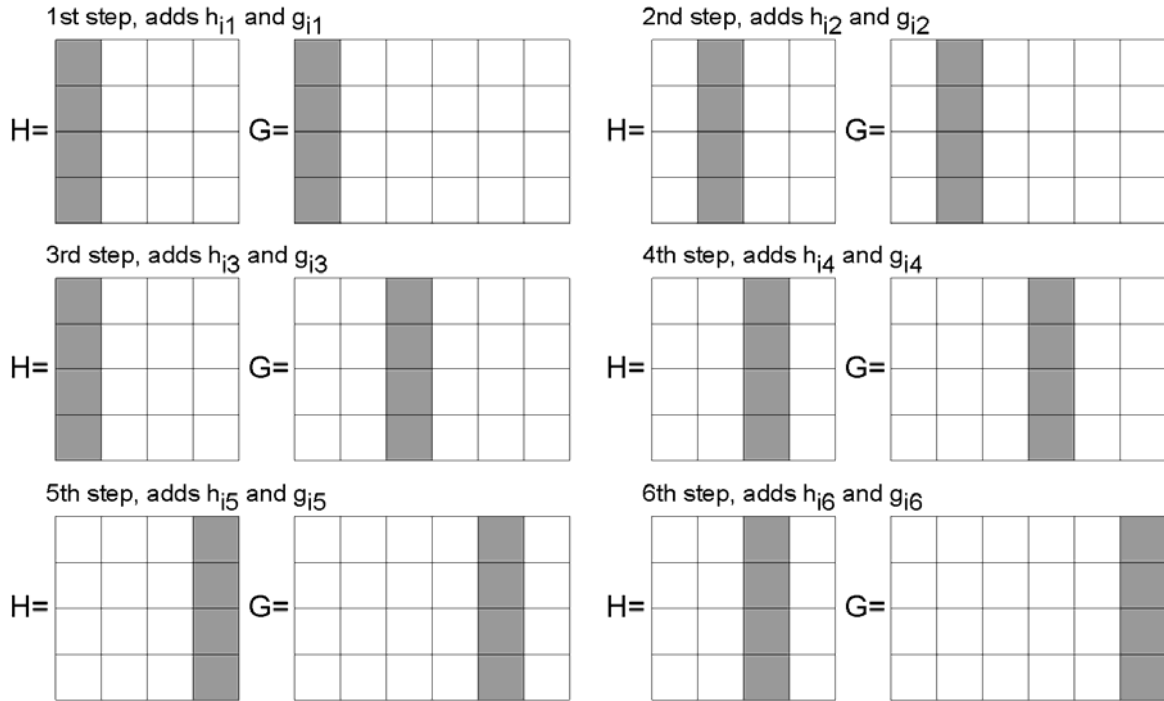


Figure 1. Reduced example of the parallel calculation of matrices $[H]$ and $[G]$.

Single-precision floating-point operations were used throughout the present implementations. But it should be mentioned that present graphics cards are capable of dealing with double-precision floating-point operations [6].

Numerical Results

Problem description. The methodologies described above are applied to a test problem depicted in figure 2. It consists of a square domain with unit length and shown boundary conditions. For potential, scalar problems, consider only the first of the indicated boundary conditions. For the vector elastostatic problems, both boundary conditions apply. The hollow circles denote nodes that only appear when quadratic elements are used. Each edge of the domain is discretized by $N/4$ elements of same length. In the example of Fig. 2, $N=8$ elements are depicted. The problems were executed in a CPU-GPU configuration and for comparisons also on the classical serial CPU. The CPU is the same for both cases.

The time consumed to calculate the matrices $[H]$ and $[G]$ for an increasing number of elements N has been determined. In the CPU-GPU system, this time corresponds to the time spent by the GPU to calculate these matrices. The resulting time is compared to the time required by a serial code written in pure C language running only on the CPU.

Assembly of influence coefficients. Figures 3 and 4 show a comparison of the amount of elapsed time required to assembly the coefficient matrices $[H]$ and $[G]$ for potential and elastostatic problems discretized by constant elements, respectively. For the largest case potential problems considered, with $N=10000$ constant elements (matrix $[G]$ with 10^8 terms)

the GPU outperforms the CPU by a factor of 56.8, as can be seen in figure 3. Analogously, in the case of the elastostatic problem with constant elements, for the larger case ($N=10000$, matrix $[G]$ with $4 \cdot 10^8$ terms) the GPU was able to run 39.6 faster than the CPU.

For the case of the potential problem with quadratic elements for the largest considered case ($N=4000$ elements, G with 96×10^6 terms) the GPU could outperform the CPU by a factor of 71.3, see figure 5.

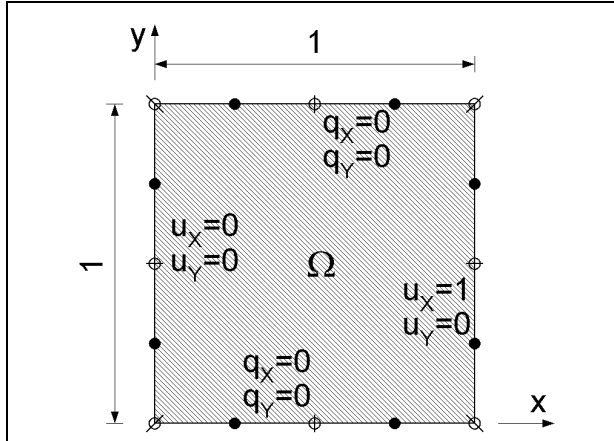


Figure 2: Square domain of unitary length and boundary conditions.

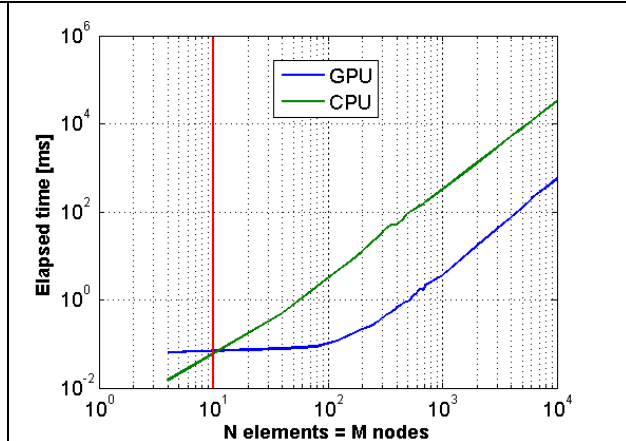


Figure 3: Time spent by the GPU and the CPU to determine $[H]$ and $[G]$ for a potential problem discretized by constant elements.

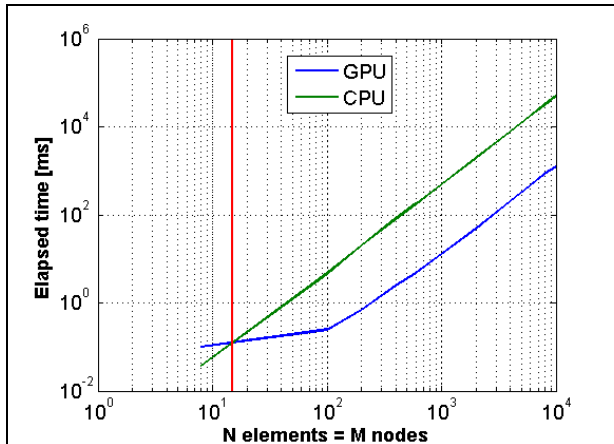


Figure 4: Time spent by the GPU and the CPU to determine $[H]$ and $[G]$ for an elastostatic problem discretized by constant elements

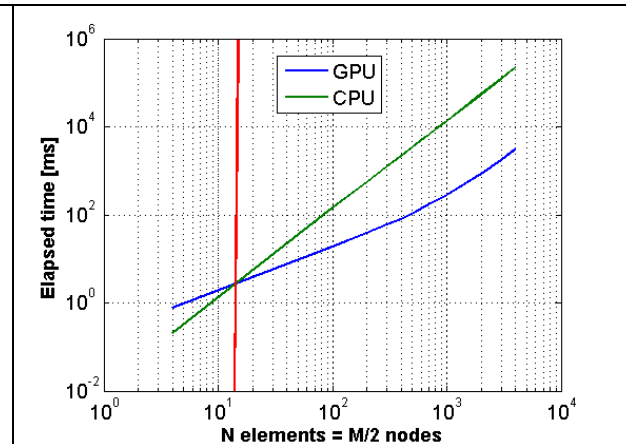


Figure 5: Time spent by the GPU and the CPU to determine $[H]$ and $[G]$ for a potential problem discretized by quadratic elements

The vertical red line in these figures marks the number of elements from which the GPU starts to outperform the CPU. For numbers of elements below this line, the volume of data to be transferred is too large compared to the volume of calculation (i.e., the problem presents low arithmetic intensity). In problems with high arithmetic intensity, that are ideally suited to GPUs, the red line is shifted to the left side.

Introduction of the boundary conditions. Figure 4 shows the time spent by the GPU and the CPU to switch the columns of matrices $[H]$ and $[G]$ and terms of the vectors $\{u\}$ and $\{q\}$, according to the boundary conditions, to form the system $[A]\{x\}=[B]\{b'\}$. The potential

problem is considered and constant elements are used. Notice that, because the amount of data to be transferred is large compared to the trivial computation involved in switching columns of $[H]$ and $[G]$, the red line is moved to the right side of the graphic, meaning that this particular problem presents low arithmetic intensity. For the case in which $N=10,000$ elements were used, the GPU was still 56.8 times faster than the CPU.

Internal points. Conversely, Fig. 7 shows the time spent to determine the potential u for N_p internal points evenly spaced in the domain shown in Fig. 2. The red line is not present, which means that even for the smallest number of elements considered, the GPU outperforms the CPU. For the case in which $N_p=3,600$ internal points were considered, the GPU was 141.6 times faster than the CPU.

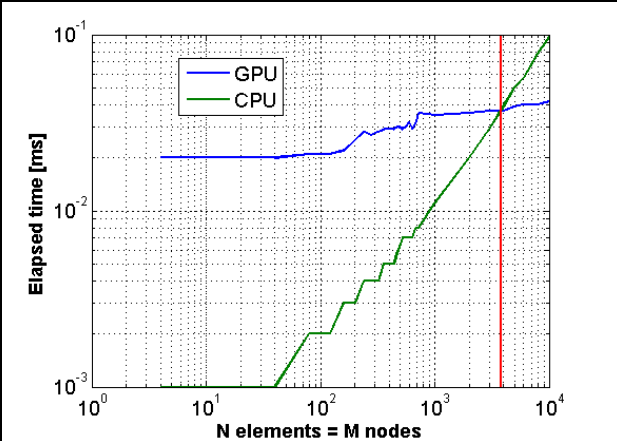


Figure 6: Time spent to switch the equation $[H]\{u\}=[G]\{q\}$ into $[A]\{x\}=[B]\{b'\}$.

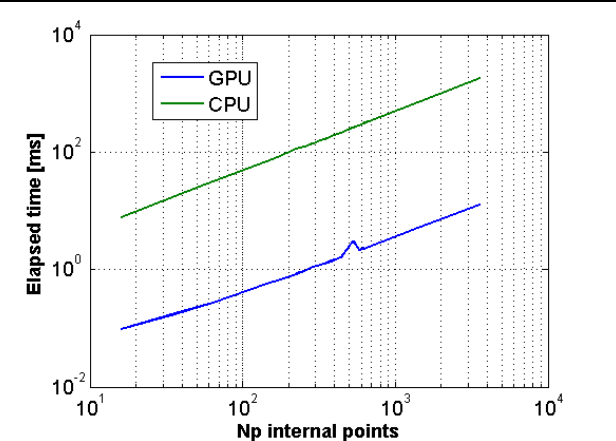


Figure 7: Time spent to calculate the solution of u for N_p evenly spaced internal points.

Solution of the complete problem. Finally, a complete problem was solved. The potential problem is considered; N constant boundary elements are used. The solution goes from calculating $[H]$ and $[G]$ and ends at the calculation of the potential u at $N_p=16$ internal points. The solution of the linear system and the matrix multiplications were performed by classical serial algorithms in the CPU. Once this time is the same in both the present GPU and CPU implementations, it is not shown in the next results.

Figure 8 shows the execution time for values of N between 4 and 10,000 elements. In the final experiment, in which $N=10,000$ elements were used, the GPU was 13 times faster than the CPU.

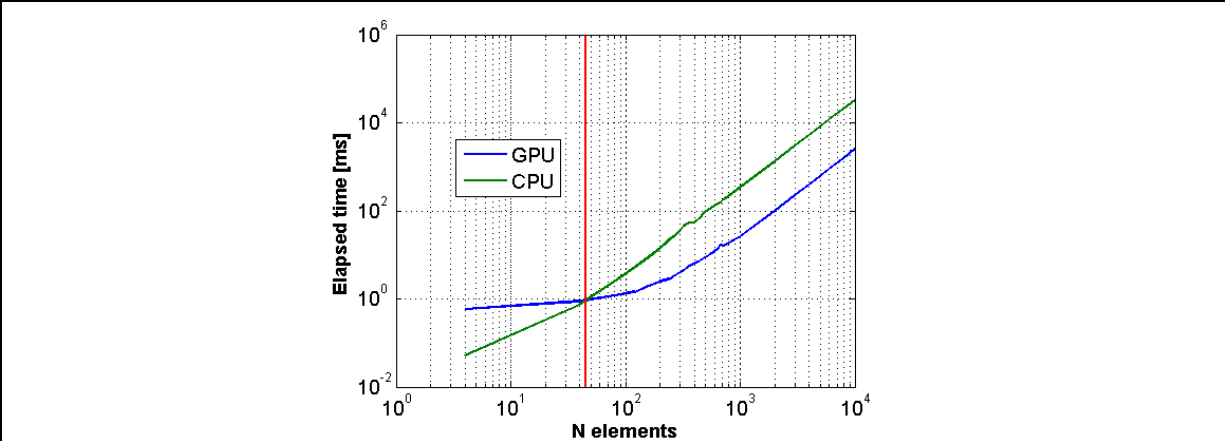


Figure 8: Execution times to solve a complete problem.

Concluding Remarks

This paper described the implementation of the Boundary Element Method for two-dimensional problems on graphics processing devices. Three different formulations are analyzed: for potential problems with constant and quadratic elements, and for elastostatic problems with constant elements. The respective classical serial implementations were rewritten under the SIMT 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 introduction of boundary conditions, that is, the rearrangement of these matrices in the form of a system of equations $[A]\{x\}=\{b\}$, and in the calculation of values at 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 numerically more efficient than the CPU with increasing number of elements and internal points.

The very simple problems discussed in this article indicate that systems based on a CPU-GPGPU architecture are very promising alternatives to increase the computational efficiency of the Boundary Element Method.

Acknowledgements

The research leading to this article has been funded by Capes, CNPq, Fapesp and Faepex/Unicamp. This is gratefully acknowledged.

References

- [1] Shreiner, D.; Woo, M.; Neider, J.; Davies, T. (eds.), *OpenGL Programming Guide*, 5th ed. Addison-Wesley (2005).
- [2] Jones, W., *Beginning DirectX9*, Premior Press (2004).
- [3] CUDA, Developer's Zone. http://www.nvidia.com/object/cuda_home.html (2008).
- [4] Owens, J. D; Luebke, D.; Govindaraju, N.; Harris, M.; Krüger, J.; Lefohn, A. E.; Purcell, T. J., 2007. *A survey of general-purpose computation on graphics hardware*. Computer Graphics Forum; 26 : 80–113.
- [5] F. Aliabadi, *The Boundary Element Method: Applications in Solids and Structures*, Wiley (2001)
- [6] NVidia., “*NVidia CUDA – Compute Unified Device Architecture – Programming Guide*”. NVidia Corporation, Santa Clara (2008).