

## IMPLEMENTATION OF THE BOUNDARY ELEMENTS METHOD FOR 2D ELASTOSTATICS 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.** *Due to its architecture, the graphics processing unit (GPU) is specially well-suited to address problems that can be expressed as data-parallel computations with high arithmetic intensity. One example of such problem is the Boundary Elements Method (BEM). This work addresses the implementation of the direct version of BEM for 2D elastostatics. 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 is also parallelized. 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 Intel Core2Duo CPU. The efficiency of the implemented strategies are investigated by solving a classical elastostatics 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 elastostatics problems on graphics hardware. The paper begins with an overview of the new technology of GPGPU. 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 third 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 elastostatics problem. Its performance is compared with an ordinary CPU serial code.

### 2. 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 vectorial 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  $(30 \text{ multiprocessors}) \times (8 \text{ thread blocks per multiprocessor}) \times (32 \text{ threads of the warp}) = 7680$  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 this 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.

### 3. IMPLEMENTATION

The part of the Boundary Elements Method's algorithm referring to the calculation of the matrices [H] and [G], i. e., the calculation of the influence coefficients  $H^j$  and  $G^j$ , is one of the easiest cases to be coded in a parallel algorithm, if the formulation of discontinuous elements is adopted. If a two-dimensional elastostatics problem is discretized in N constant boundary elements, these matrices will have dimensions  $2N \times 2N$ , because the number of degrees of freedom for this kind of problem is twice the number of elements.

In the present implementation, the matrices [H] and [G] are allocated as vectors of size  $4N^2$  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  $4 \times 4$  threads. The size of a two-dimensional grid is calculated so as to contain as many blocks as needed to accommodate the  $4N^2$  terms of  $[H]$  and  $[G]$ .

Figure 1 illustrates the sizes of grids and blocks for a reduced example. In this example, in which  $N = 3$  boundary elements were used, matrices  $[H]$  and  $[G]$  will have dimensions of  $2N \times 2N = 6 \times 6$ . The thread blocks were defined as containing  $4 \times 4$  threads. From Fig. 1, it is observed that the grid will then be calculated to contain  $2 \times 2$  blocks, in a total of  $8 \times 8 = 64$  threads. Even so, only  $6 \times 6 = 36$  out of the 64 threads will perform the calculations of  $H^{ij}$  and  $G^{ij}$ . The darkened cells in Fig. 1 represent the terms that will perform some calculation, while the blank cells represent the threads that were created, but left inactive.

Two  $4 \times 4$  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 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 of its respective pair  $H^{ij}/G^{ij}$ . 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.

The present implementation was applied to calculate the influence matrices of an elementary elastostatics problem by BEM, and the results are reported in the next section.

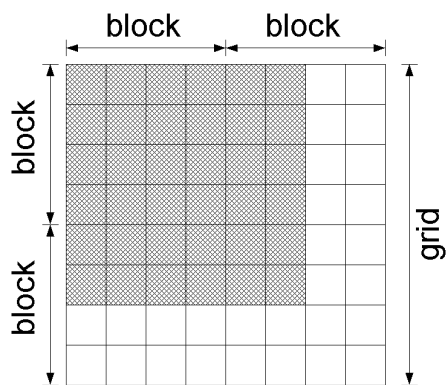


Figure 1. Reduced example of a grid of thread blocks.

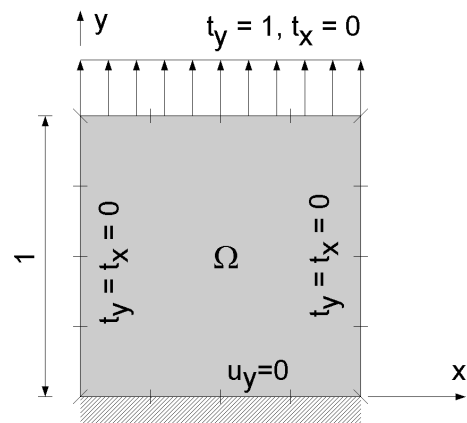


Figure 2. Two-dimensional square plate of unitary edge.

#### 4. RESULTS

The traction-displacement problem depicted by Fig. 2 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 and right borders have zero traction in both  $x$  and  $y$  directions, the upper border has unitary traction in the  $y$  direction and zero in the  $x$  direction, and the lower border is clamped in the  $y$  direction and free to move in the  $x$  direction.

The time consumed to fill the matrices  $[H]$  and  $[G]$  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 3 shows the elapsed times for values of  $N$  between 4 and 5000 elements, which means matrices of size between 8 and 10000 terms.

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. Once 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 5,000 elements was considered, the GPU obtained the matrices  $[H]$  and  $[G]$  (of size  $10,000 \times 10,000$ ) in a time 39.6 times shorter than the CPU.

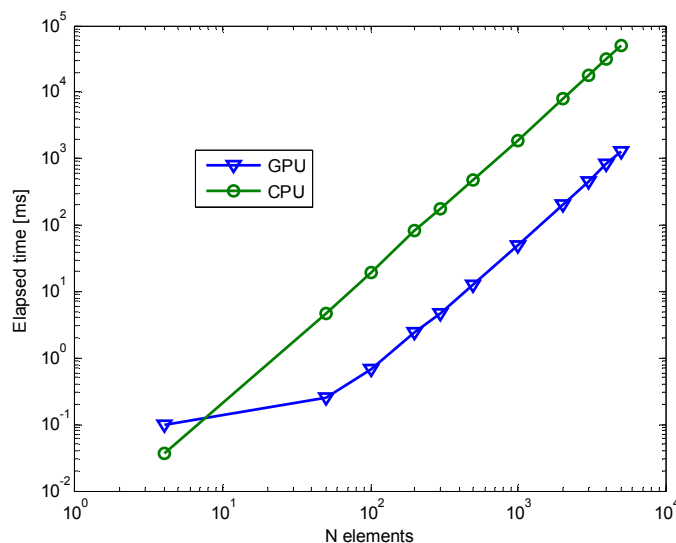


Figure 3. Time spent by the GPU and the CPU to calculate  $[H]$  and  $[G]$ .

In the formulation of BEM for 2D elastostatics, the calculation of a given term  $H^{ij}$  or  $G^{ij}$  depends on whether the term refers to the  $x$  or the  $y$  directions of the displacement/traction of the element. In the present implementation, some control structures (if... then... else) were introduced so the program could select the proper integration expression according to these directions. These control structures cause the flow of data to break. The ideal flow of data for a parallel implementation on GPU is the so called *stream processing*, in which all the threads execute the same procedure over different data. Whenever a group of threads have to follow different paths of computation, executing different procedures over the data, the stream of computation is broken and the performance is harmed.

For example, in a previous work (Labaki et al., 2009), the formulation of BEM for potential problems was implemented on the GPGPU. In that formulation, no break of flow was needed, and therefore the calculation of  $[H]$  and  $[G]$  of size  $10,000 \times 10,000$  were performed by the GPU 56.8 times faster than by the CPU.

After the calculation of the matrices of influence, their columns have to be switched according to the boundary conditions, and a linear system of equation is created. The solution of this system of equation leads to the solution of the problem: the unknown values of displacements and tractions (Kane, 1994; Brebbia, 1978). All these procedures have algebraic nature: none of them depends on whether a potential or elastostatics problem is being treated. The superiority of performance of the GPU over the CPU on dealing with these procedures was already shown by Labaki et al. (2009).

## 5. CONCLUSION

This paper has described the implementation of the Boundary Elements Method for two-dimensional elastostatics 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 the most significant step of BEM: the calculation of the influence matrices. It was observed that the point from which the GPU presents better performance than the CPU is function of the arithmetic intensity of the problem. 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. One of them is the reformulation of the algorithm according to the *stream processing* philosophy, which should avoid breaks in the flow of data and therefore allow an improvement of performance of the GPU.

## 6. REFERENCES

- Brebbia, C. A., 1978, "The Boundary Element Method". Pentech Press, London.
- Cooperman, G., Kaeli, D., 2008, "GPGPU Programming – Syllabus". 28 November 2008. <<http://www.ccs.neu.edu/course/csu610/#syllabus>>
- Kane, J. H., 1994, "Boundary Element Analysis in Engineering Continuum Mechanics". Prentice Hall Englewood Cliffs.
- Klößner, A., Hesthaven, J. S., 2008, "Metaprogramming Graphics Processors from High-Level Languages". 28 November 2008 <<http://mathematician.de/entry/dam>>

- Labaki, J., Ferreira, L. O. S., Mesquita, E. “Implementation of Constant Boundary Elements For 2D Potential Problems on Graphics Hardware – GPU”, Proceedings of the 20th International Congress of Mechanical Engineering, Gramado, Brazil (submitted).
- 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> >

## 7. RESPONSIBILITY NOTICE

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