The Boundary Element Method for 2D Elastostatics on Graphics Hardware-GPGPU

Josue Labaki, Luiz Otávio Saraiva Ferreira and Euclides Mesquita

Unicamp, State University of Campinas, Campinas-SP, Brazil

Received: April 28, 2011 / Accepted: May 12, 2011 / Published: October 10, 2011.

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 equations 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 is investigated by solving a classical elastostatics problem.

Key words: High performance computing, graphics hardware, boundary elements method.

1. Introduction

Since 2006, high performance computing applications were introduced to a new and promising alternative of implementation, the General Purpose Graphics Processing Units (GPGPUs). These graphics devices used to be dedicated to graphics calculations, especially in the computer game industry. Their astonishing performance on dealing with graphics-related parallel computing attracted early researches about two decades ago who used the devices to implement their non-graphics problems, such as Finite Difference Methods, Particle Based Methods, Lattice Boltzmann Method and also to the Finite Element Method [1]. Oishi and Yoshimura [1] reported that in those cases, the GPU was controlled by the so-called graphics APIs (Application Programming Interfaces), such as OpenGL [2] or DirectX [3]. More recently, the GPU started to be redesigned to perform non-graphics calculations, and a few programming languages were developed to allow the programmers to code the devices in a more convenient, higher-level programming paradigm [4].

Nevertheless, the GPGPU still maintains its high-bandwidth memories and its floating-point operations are still significantly faster than the ordinary CPU’s. These highly threaded devices are especially well-suited for the implementation of problems that possess high arithmetic intensity. The Boundary Element Method (BEM), whose parallel formulations have been exhaustively explored for CPU clusters, is an example of such problems.

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 elastostatic 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

Graphics processing units (GPUs) are specialized computing architectures designed to accelerate the processing of images in a frame buffer for output to a display. The type of calculation they are intended for include pixel shading, vertex transformation and rasterization, which are tasks that require little control and communication, when compared to the volume of calculations. For this reason, GPUs have been designed since its beginning as parallel computing devices, although dedicated to graphics calculations.

In the last years, however, the most important graphics card makers have completely changed their approaches to GPUs by rethinking its design for general-purpose calculations. The resulting General-Purpose GPUs (GPGPUs) possess a combination of hardware and software that allows the use of traditional GPUs to perform extremely demanding, parallelizable computing tasks. 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) [5].

For example, a typical card launched in the end of 2009 is the NVidia GeForce GTX 280 model. This graphics card is organized as an array of 240 highly threaded streaming processors (SPs), distributed among 30 streaming multiprocessors (SMs). Its architecture of cooperative many-core computing units resembles the one found in clusters of CPUs, but it is confined in a single hardware device. Because of this architecture, GPGPUs require a single instruction multiple thread programming paradigm (SIMT).

It is important to point out that Graphics Processing Units are not suitable for every type of application. Some applications benefit from its use more than others do. GPUs are very efficient when it comes to matrix arithmetic and other parallel data operations. In everyday computing tasks such as running an operating system or word processor, which involve recursive, adaptive, and interdependent calculations and demand a large amount of the computation resources to be dedicated to communication of data and control, ordinary Central Processing Units (CPUs) are still a better choice than GPUs. The bottom line is that a good candidate for GPU implementation is a problem that requires little control and communication, when compared to the volume of calculations. (i.e., it possesses a high arithmetic intensity) [5]. In many cases satisfying this requirement, a superior computational performance has been obtained by these cards, when compared to regular CPUs [6-11].

In order to provide the programmers with a more convenient programming paradigm, NVidia and AMD, the two major graphics card makers developed application programming interfaces (APIs) with which the programmers can code the GPUs in a higher-level environment, compared to the former OpenGL and Direct-X [3]. CUDA (Computer Unified Device Architecture) was provided in this context by NVidia for their devices. It is considered a low-level language because of its requirement that the programmer should explicitly allocate and free memory, declare data copies, choose parameters of parallelism, etc. CUDA is
based on an extension of the C programming language, with extra function and variable type qualifiers, kernel execution directives and additional built-in variables [5].

While programming CUDA, the concepts of thread, thread block and grid will come up. A thread is a virtualized CPU. It is the elementary execution unit which will execute a kernel (a set of instructions) over a single data. A device-dependent large number of threads will perform the same kernel on a large set of different data. Thread blocks are abstractions used to group the threads together and distribute them among the multiprocessors of the card. Grids are used to group the thread blocks in the same way.

The number of threads that can be executed simultaneously by each multiprocessor is called warp. If the thread block associated to a given multiprocessor is larger than the warp, its remaining threads will be scheduled to be executed as soon as the first batch-of-warps-of threads have ended their calculations. The same applies to the card as a whole. If the grid contains more thread blocks than the number of its multiprocessors, the remaining thread blocks will be queued for execution as soon as a multiprocessor is available. It is left to the programmer to decide the sizes of thread blocks and grid for each problem. This decision impacts directly on the efficiency of the program.

A complex memory architecture is observed on graphics devices. The global memory of the card is an ordinary RAM memory of up to 2 GB of space, and it is a common choice to place data that must be accessible by all the threads of a grid. The shared memory of each multiprocessor has only 16 kB of space, but the access to it is up to 600 times faster than to the global memory. The data placed in this memory is accessible by the threads of its multiprocessor only. Furthermore, each thread possesses private local memories and register space. GPUs also have special constant and texture read-only cache memories, devoted to specific purposes in the graphics calculation [5].

In a typical CUDA program, all the vectors and matrices containing the data of the problem 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. Pointers to these arrays are passed as arguments to the kernels. The results of the operation of the threads must be transferred back from the GPU so they can be post-processed. All these memory manipulations expend some processor clock cycles, and a precise and fair benchmark of CPU versus GPU performance should take this fact into consideration.

The following section reports 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 \(H_{ij}\) and \(G_{ij}\), is computationally very demanding, but because of its high arithmetic intensity it is a good candidate for implementation on graphics hardware. 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 are 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 \([G]\).

Fig. 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 \( 2 \times 2 N = 6 \times 6 \). 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, these 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.

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. Fig. 3 shows the elapsed times for values of \( N \) between 4 and 5,000 elements, which mean matrices of size between 8 and 10,000 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.

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 that 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 [12], 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 [13, 14]. 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. [12].

5. Concluding Remarks

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 SIMT 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.

References


