Optimizing a Boundary Elements Method implementation for Stationary Elastodynamic Problems with Graphical Processing Units (GPU)

Giuliano A. F. Belinassi[1], Rodrigo Siqueira[1], Ronaldo Carrion[2] Alfredo Goldman[1]

[1]Instituto de Matemática e Estatística (IME) – Universidade de São Paulo (USP)
Rua do Matão, 1010 – São Paulo – SP – Brazil

[2]Escola Politécnica (EP) – Universidade de São Paulo (USP)
Avenue Professor Mello Moraes, 2603 – São Paulo – SP – Brazil

Abstract. *The Boundary Element Method requires a geometry discretization to execute simulations, and it can be used to analyze the 3D stationary behavior of wave propagation in the soil. Such discretization involves in generating two high computational power demanding matrices, and this article demonstrates how Graphical Processing Units (GPU) can be used to accelerate this process.*

1. Introduction

Since the computer was proven to be a useful machine, there always has been an interest of building faster versions of it to solve bigger and more complex problems in a matter of time that no human can. One way archiving this is by building more complex sequential CPUs, with more transistors. Recently, as the size of CPU components got smaller, it became difficult to create such faster sequential CPUs [Brock and Moore 2006]. As consequence, CPU manufacturers are investing in multicore CPUs, capable of running more than one task asynchronously.

Parallel computing is hard to define, but intuitively, it is a computation method that allows data to be distributed and processed simultaneously. In Flynn's taxonomy [Pacheco 2011], there are two types of parallel computer archictectures:

1. Single Instruction, Multiple Data (SIMD); a processor that allows a chunk of data to be loaded and a single instruction to be used to process it. As example of SIMD, there is AMD's 3DNow! [AMD 2000] and Graphics Processing Units (GPU).
2. Multiple Instruction, Multiple Data (MIMD); a system consisting of multiple independent processing units, executing asynchronously. Multicore CPUs are a example of MIMD.

Originally, GPU was designed to render graphics in real time and OpenGL and DirectX were the first libraries designed to access GPUs resources and provide graphics. However, researchers realized that GPUs could be used for other applications different from graphics. Consequently, these libraries were used by engineers and scientists in their specific problems, however, they had to convert their problem into a graphical domain.

NVIDIA noticed a new demand for their products and created an API called CUDA to enable the use of GPUs in general purpose situation. CUDA has the concept of kernels, which are functions called from host to be executed in the GPU threads. Kernels are organized into a set of blocks wherein each block is a set of threads that cooperate with each other [Patterson and Hennessy 2007].

GPU's memory is divided into global memory, local memory, and shared memory. First, it is a memory that all threads can access. Second, it is a memory that is private to a thread. Third, it is a low-latency memory that is shared between all threads in the same block. [Patterson and Hennessy 2007].

2. Context

Differential equations governing problems of Mathematical Physics only have analytical solutions in cases in which both the domain geometry and the boundary and initial conditions are reasonably simple. Problems with arbitrary domains and fairly general boundary conditions can only be solved in an approximate way, for example, by numerical techniques. These techniques have experienced strong development due to the presence of increasingly powerful digital electronic computers, allowing the solution of complex mathematical problems. The Boundary Element Method (BEM) is a very efficient alternative for modeling unlimited domains, since it naturally satisfies the Sommerfeld radiation condition, also known as geometric damping. Such method can be used to numerically model the 3D stationary behavior of wave propagation in the soil based on BEM with the objective of creating a computational tool which can assist in analysis such as evaluation of vibrations in the soil rising from machine tools operation or railway lines as well as to understand the role of soil during earthquakes or even in the design of offshore oil platforms.

Wihout addressing details on the BEM formulation, the Bondary Integral Equation for Stationary Elastodynamic Problems can be written as:

$$c_{ij}u_j(\xi,\omega) + \int_S t^*_{ij}(\xi,x,\omega)u_j(x,\omega)\mathrm{d}S(x) = \int_S u^*_{ij}(\xi,x,\omega)t_j(x,\omega)\mathrm{d}S(x) \tag{1}$$

After performing the geometry discretization, Equation (1) can still be represented in matrix form as:

$$[H]\{u\} = [G]\{t\} \tag{2}$$

Algorithmically, equation (1) is implemented into a routine named `Nonsingd`, computing the integral using the Gaussian Quadrature without addressing problems related to singularity. To overcome singularity problems, there is a special routine called `Sigmaec` that uses an artifice known as Regularization of the Singular Integral. By last, `Ghmatecd` is a routine developed to create both H and G matrices described in equation (2).

3. Parallelization Strategies

A parallel implementation of BEM began by analyzing and modifying a sequential code provided by [Carrion 2002]. Gprof [GNU], a profiling tool by GNU, showed that the most time-consuming routine was Nonsingd. Since most calls to `Nonsingd` were from `Ghmatecd`, most of the parallelization effort was focused on that last routine.

3.1. Ghmatecd Parallelization

Algorithm 1 shows the pseudocode of `Ghmatecd` subroutine. Let n be the number of mesh elements and m the number of boundary elements. `Ghmatecd` builds matrices H and G by computing smaller 3×3 matrices returned by `Nonsingd` and `Sigmaec`.

Algorithm 1 Creates $H, G \in \mathbb{C}^{(3m)\times(3n)}$

```
 1: procedure GHMATECD
 2:     for $j := 1, n$ do
 3:         for $i := 1, m$ do
 4:             $ii := 3(i-1)+1; jj := 3(j-1)+1$
 5:             if $i == j$ then
 6:                 $Gelement, Helement \leftarrow$ Sigmaec($i$)    ▷ two $3 \times 3$ complex matrices
 7:             else
 8:                 $Gelement, Helement \leftarrow$ Nonsingd($i, j$)
 9:             $G[ii : ii+2][jj : jj+2] \leftarrow Gelement$
10:             $H[ii : ii+2][jj : jj+2] \leftarrow Helement$
```

There is no interdependency between all iterations in lines 2-3 loops, thus, all iterations can be computed in parallel. Since typically high-end CPUs have 8 cores, even a small number of mesh elements generate enough workload to use all CPUs resources if this strategy alone is used. By another hand, a GPU contain thousands of processors, thus even a considerable large amount of elements may not generate a workload in a way that it consumes all its resources. Since `Nonsingd` is the cause of the high time cost of `Ghmatecd`, the main effort was to implement an optimized version of `Ghmatecd`, called `Ghmatecd_Nonsingd`, that only computes the `Nonsingd` case in the GPU, and leave `Sigmaec` to be computed in the CPU after the computation of `Ghmatecd_Nonsingd` is completed. With this, a new strategy arises when also computing Nonsingd in parallel. Let g be the number of Gauss quadrature points. The pseudocode in Algorithms 2 and 3 pictures this new strategy.

Algorithm 2 Creates $H, G \in \mathbb{C}^{(3m)\times(3n)}$

```
 1: procedure GHMATECD_NONSINGD
 2:     for $j := 1, n$ do
 3:         for $i := 1, m$ do
 4:             $ii := 3(i-1)+1; jj := 3(j-1)+1$
 5:             Allocate Hbuffer and Gbuffer, buffer of matrices $3 \times 3$ of size $g^2$
 6:             if $i \neq j$ then
 7:                 for $y := 1, g$ do
 8:                     for $x := 1, g$ do
 9:                         Hbuffer($x, y$) $\leftarrow$ GenerateMatrixH($i, j, x, y$)
10:                         Gbuffer($x, y$) $\leftarrow$ GenerateMatrixG($i, j, x, y$)
11:             $Gelement \leftarrow$ SumAllMatricesInBuffer(Gbuffer)
12:             $Helement \leftarrow$ SumAllMatricesInBuffer(Hbuffer)
13:             $G[ii : ii+2][jj : jj+2] \leftarrow Gelement$
14:             $H[ii : ii+2][jj : jj+2] \leftarrow Helement$
```

Algorithm 3 Compute Sigmaec part of $H, G \in \mathbb{C}^{(3m)\times(3n)}$

```
1: procedure GHMATECD_SIGMAEC
2:     for $i := 1, m$ do
3:         $ii := 3(i-1)+1$
4:         $Gelement, Helement \leftarrow$ Sigmaec($i$)
5:         $G[ii : ii+2][ii : ii+2] \leftarrow Gelement$
6:         $H[ii : ii+2][ii : ii+2] \leftarrow Helement$
7: procedure GHMATECD
8:     Ghmatecd_Nonsingd()
9:     Ghmatecd_Sigmaec()
```

The `Ghmatecd_Nonsingd` can be implemented as a CUDA kernel. Inside of a CUDA block, create $g \times g$ threads to compute in parallel the two nested loops in lines 2-3 and allocate spaces in the shared memory to keep the buffer of matrices (`Hbuffer` and `Gbuffer`). Since these buffers contain matrices of size 3×3, nine of these $g \times g$ threads can be used to sum all matrices because one thread can be assigned to each matrix entry, unless $g < 3$. Notice that g is also upper-bounded by the amount of shared memory available in the GPU. Launching $m \times n$ blocks to cover the two nested loops in lines 2 to 3 will generate the entire H and G without the `Sigmaec` part. The `Ghmatecd_Sigmaec` can be parallelized with a simple OpenMP Parallel for clause, and it will calculate the remaining H and G.

4. Methodology

In order to check if the final result obtained by the parallel program is numerically compatible with the original, the concept of matrix norms are necessary. Let $A \in \mathbb{C}^{m\times n}$. [Watkins 2004] defines matrix 1-norm as:

$$\|A\|_1 = \max_{1\le j\le n} \sum_{i=1}^{m} |a_{ij}| \tag{3}$$

All norms have the propierty that $\|A\| = 0$ if and only if $A = 0$. Let f and g be two numerical algorithms that solves the same problem, but in a different fashion. Let now y_f be the result computed by f and y_g be the result computed by g. The *error* between those two values can be measured computing $\|y_f - y_g\|$.

The error between CPU and GPU versions of H and G matrices were computated by calulating $\|H_{cpu} - H_{gpu}\|_1$ and $\|G_{cpu} - G_{gpu}\|_1$. An automated test check if this value is bellow 10^{-4}.

For experimenting, there were four data samples as illustrated in Table 1. The application is executed in a computer with an AMD A10-7700K processor aided with a GeForce GTX980 for each one of the samples using the original code (serial implementation), the OpenMP version, and the CUDA and OpenMP together. Before any data collection, a warm up procedure is executed, which consists of running the application with the sample three times without getting any result. Afterward, all experiments were executed 30 times per sample. Each execution produces a file with total time elapsed.

Table 1. Table 1: Data experiment set

Number of Mesh elements	240	960	2160	4000
Number of Bondary elements	100	400	900	1600

GPU total time was computed by the sum of 6 elements: (1) total time to move data to GPU, (2) launch and execute the kernel, (3) elapsed time to compute the result, (4) time to move data back to main memory, (5) time to compute the remaining H and G parts in the CPU. The elapsed time was computed in seconds with the OpenMP library function `OMP_GET_WTIME`. This function calculates the elapsed wall clock time in seconds with double precision. All experiments set the Gauss Quadrature Points to 8.

5. Results

Gfortran 5.4.0 [GFortran 2017] and CUDA 8.0 [NVIDIA 2017] were used to compile the application. The main flags used in Gfortran are `-Ofast -march=native -funroll-loops -flto`. The flags used in CUDA nvcc compiler are: `-use_fast_math -O3 -Xptxas --opt-level=3 -maxrregcount=32 -Xptxas --allow-expensive-optimizations=true`.

The logarithmic scale graphic at figure 1 illustrates the results. All points are the mean of the time in seconds of 30 executions as described in Methodology.

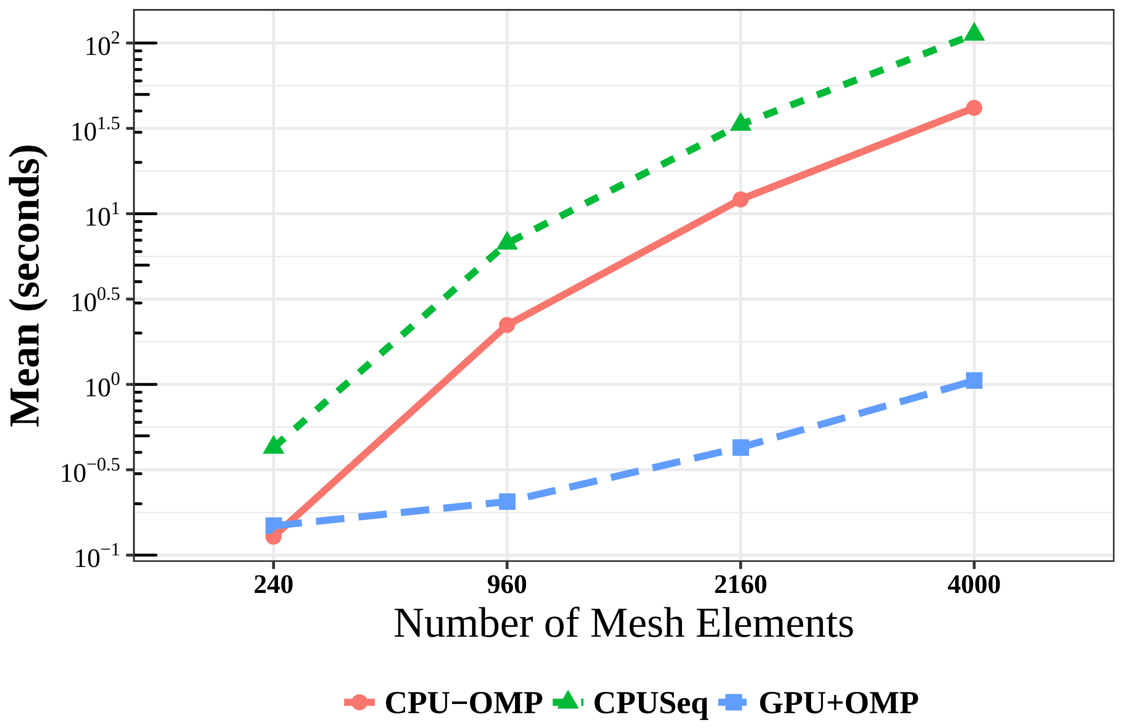

Figure 1. Time elapsed by each implementation in logarithm scale

Since the speedup acquired in the 4000 mesh elements sample with OpenMP and CUDA+OpenMP with respect to the sequential algorithm are 2.68 and 106.95 respectively, the conclusion is that the presented strategy aided with GPUs can be used to accelerate the overall performance of the simulation for a large number of mesh elements.

This is a consequence of parallelizing the construction of both matrices H and G, and the calculations in the `Nonsingd` routine. Notice that there is a performance loss in the 260 sample between OpenMP and CUDA+OpenMP, this is caused by the high latency between CPU-GPU communication, thus the usage of GPUs for smaller data may not be attractive.

6. Future Works

The current implemented code have limitations. First, there is no logic to assemble H and G by blocks by lauching multiple kernels. This strategy would allow larger problems to be solved, since if the number of mesh elements is large enough the application would crash due to insufficient GPU memory. Second, the singular and nonsingular part can be computed independently, so the CPU can be used to compute the singular part while the GPU is computing the nonsingular part. The usage of GPUs in the singular case can also be analyzed.

References

AMD (2000). Amd 3dnow! `http://support.amd.com/TechDocs/21928.pdf`. Accessed: 2017-07-08.

Brock, D. C. and Moore, G. E. (2006). *Understanding Moore's law: four decades of innovation*. Chemical Heritage Foundation.

Carrion, R. (2002). *Uma Implementação do Método dos Elementos de Contorno para problemas Viscoelastodinâmicos Estacionários Tridimensionais em Domínios Abertos e Fechados*. PhD thesis, Universidade Estadual de Campinas.

GFortran, G. (2017). Gnu gfortran options. `https://gcc.gnu.org/onlinedocs/gfortran/Option-Summary.html`. Accessed: 2017-07-09.

GNU. Gnu binutils. `https://www.gnu.org/software/binutils/`. Accessed: 2017-05-08.

NVIDIA (2017). Cuda toolkit documentation. `http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#axzz4pGcmz4XI`. Accessed: 2017-07-09.

Pacheco, P. (2011). *An introduction to parallel programming*. Elsevier.

Patterson, D. A. and Hennessy, J. L. (2007). Computer organization and design. *Morgan Kaufmann*.

Watkins, D. S. (2004). *Fundamentals of matrix computations*, volume 64. John Wiley & Sons.