Optimizing a Boundary Elements Method for Stationary Elastodynamic Problems implementation with GPUs

Giuliano A. F. Belinassi[1], Rodrigo Siqueira[1], Ronaldo Carrion[2] , Alfredo Goldman[1], Marco D. Gubitoso[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)
Avenida 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 generating two high computational power demanding matrices, and this article demonstrates how Graphical Processing Units (GPU) were used to accelerate this process. For an experiment with 4000 Mesh elements and 1600 Boundary elements, a speedup of 106.95 is obtained with a GeForce GTX980.*

1. Introduction

Differential equations governing problems of Mathematical Physics only have analytical solutions in cases in which both the domain geometry, 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 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. This tool can assist in analysis such as evaluation of vibrations in the soil rising from machine tools operation or railway lines, as well as understand the role of soil during earthquakes or even in the design of offshore oil platforms.

With the invention of GPUs, several mathematical and engineering simulation problems were redesigned to be implemented into GPUs to explore its massively parallel capabilities. However, first GPUs were designed to render graphics in real time, as a consequence, all the available libraries were graphical oriented such as OpenGL. These redesigns involved converting the original problem into a graphical domain, requiring deep knowledge of the selected graphical library.

NVIDIA noticed a new demand for their products and created an API called CUDA to enable the use of GPUs in general purpose situations. 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]. CUDA provides mechanisms to access all of them.

Before discussing any parallelization technique or results, Section 2 presents a very brief mathematical formulation of BEM for Stationary Elastodynamic Problems and the meaning of some functions presented in the current work. Section 3 shows how the most computational expensive routine was optimized using GPUs. Section 4 discusses how the results were obtained. Section 5 presents and discusses the results. Finally, Section 6 provides an overview of our future work.

2. Boundary Elements Method Background

Without addressing details on the BEM formulation, the Boundary Integral Equation for Stationary Elastodynamic Problems can be written as:

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

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

$$[H]\{u\} = [G]\{t\} \qquad (2)$$

Functions $u^*_{ij}(\xi,x,\omega)$ and $t^*_{ij}(\xi,x,\omega)$ (called fundamental solutions) present a singular behavior when $\xi = x$ ordely $O(1/r)$, called weak singularity, and $O(1/r^2)$, called strong singularity, respectively. The r value represents the distance between x and ξ points. To overcome this problem in the strong singularity, one use the artifice known as Regularization of the Singular Integral that can be expressed as follows:

$$\begin{aligned} c_{ij}(\xi)u_j(\xi,\omega) + \int_S \left[t^*_{ij}(\xi,x,\omega)_{\text{DYN}} - t^*_{ij}(\xi,x)_{\text{STA}}\right] u_j(x,\omega)\mathrm{d}S(x) + \\ + \int_S t_{ij}(\xi,x)_{\text{STA}}u_j(x)\mathrm{d}S(x) = \int_S u^*_{ij}(\xi,x,\omega)_{\text{DYN}}t_j(x,\omega)\mathrm{d}S(x) \end{aligned} \qquad (3)$$

Where DYN = Dynamic, EST = Static. The integral of the difference between the dynamic and static nuclei, first one in Equation (3), does not present singularity when executed concomitantly as expressed because they have the same order in the both problems.

Algorithmically, equation (1) is implemented into a routine named `Nonsingd`, computing the integral using the Gaussian Quadrature [Ascher and Greif 2011] without addressing problems related to singularity. To overcome singularity problems, there is a special routine called `Sing_de` that uses the artifice described in equation (3). By last, `Ghmatecd` is a routine developed to create both H and G matrices described in equation (2).

Table 1. Gprof Output for 960 Mesh Elements

Name	Nonsingd	Ghmatecd
Time	58.3%	60.9%

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, revealed the two most time-consuming routines as shown in Table 1. 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 `Singi_de`.

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 ← Sing_de(i)   ▷ two 3 × 3 complex matrices
7:              else
8:                  Gelement, Helement ← Nonsingd(i, j)
9:              G[ii : ii + 2][jj : jj + 2] ← Gelement
10:             H[ii : ii + 2][jj : jj + 2] ← 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. On the other hand, a GPU contain thousands of processors, hence even a considerable large amount of elements may not generate a workload in a way that it consumes all the device's 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 `Sing_de` to be computed in the CPU after the computation of `Ghmatecd_Nonsingd` is completed. The pseudocode in Algorithm 2 pictures a new strategy where `Nonsingd` is also computed in parallel. Let g be the number of Gauss quadrature points.

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 × 3 of size g^2
6:              if i ≠ j then
7:                  for y := 1, g do
8:                      for x := 1, g do
9:                          Hbuffer(x, y) ← GenerateMatrixH(i, j, x, y)
10:                         Gbuffer(x, y) ← GenerateMatrixG(i, j, x, y)
11:             Gelement ← SumAllMatricesInBuffer(Gbuffer)
12:             Helement ← SumAllMatricesInBuffer(Hbuffer)
13:             G[ii : ii + 2][jj : jj + 2] ← Gelement
14:             H[ii : ii + 2][jj : jj + 2] ← Helement
15: procedure GHMATECD_SING_DE
16:     for i := 1, m do
17:         ii := 3(i − 1) + 1
18:         Gelement, Helement ← Sing_de(i)
19:         G[ii : ii + 2][ii : ii + 2] ← Gelement
20:         H[ii : ii + 2][ii : ii + 2] ← Helement
21: procedure GHMATECD
22:     Ghmatecd_Nonsingd()
23:     Ghmatecd_Sing_de()
```

The `Ghmatecd_Nonsingd` can be implemented as a CUDA kernel. Inside of a CUDA block, there are created $g \times g$ 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 `Sing_de` part. The `Ghmatecd_Sing_de` can be parallelized with a simple OpenMP Parallel for clause, and it will calculate the remaining H and G.

4. Methods

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:

$$\|A\|_1 = \max_{1\leq j\leq n} \sum_{i=1}^{m} |a_{ij}| \tag{4}$$

All norms have the property that $\|A\| = 0$ if and only if $A = 0$. Let f and g be two numerical algorithms that solve the same problem, but in a different fashion. Let now

Table 2. Data experiment set

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

y_f be the result computed by f and y_g be the result computed by g. The *error* between these two values can be measured computing $\|y_f - y_g\|$.

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

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`.

For experimenting, there were four data samples as illustrated in Table 2. The application was executed in a computer with an AMD A10-7700K processor paired 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. All tests but the sequential set the number of OpenMP threads to 4.

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, where a script collected the mean and standard deviation.

GPU total time was computed by the sum of 5 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

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, and the maximum standard deviation obtained was 2.6% of the mean value.

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. As a conclusion, the presented strategy paired 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 here may not be attractive.

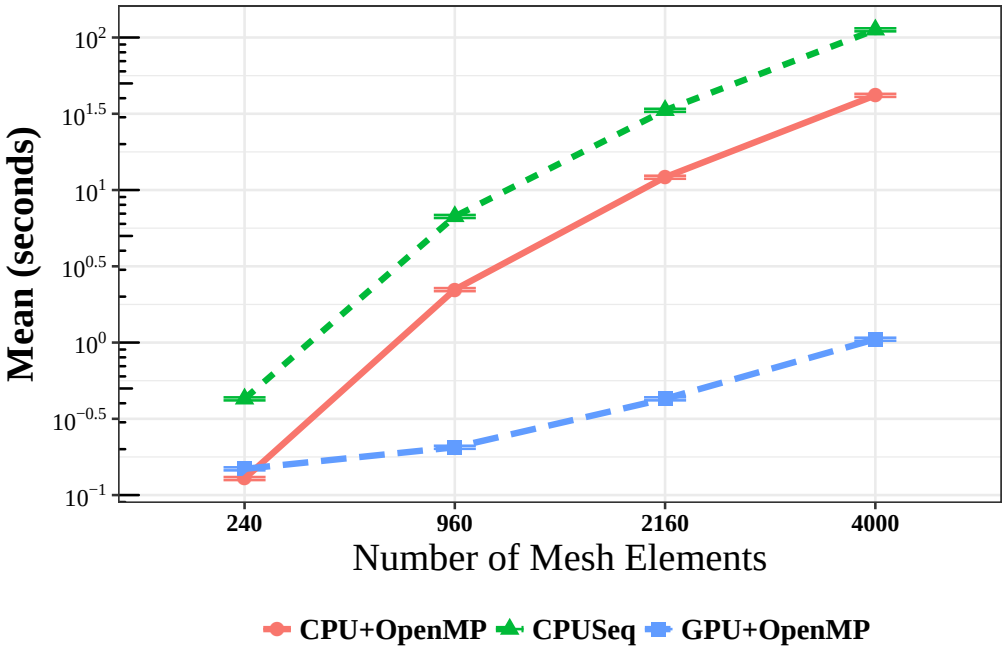

Figure 1. Time elapsed by each implementation in logarithm scale

6. Future Works

There are some issues related to the g described in Algorithm 2. Detailed studies are required to determine what is a g that provides a good relation between precision and performance. Also, better ways to compute the sum in lines 11-12 of Algorithm 2 may increase performance. The usage of GPUs in the singular case can also be analyzed.

References

Ascher, U. M. and Greif, C. (2011). *A first course on numerical methods*. SIAM.

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/`. Accessed: 2017-07-09.

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.