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. In an experiment with 4000 Mesh elements and 1600 Boundary elements, a speedup of 107× was obtained with a GeForce GTX980.*

1. Introduction

Differential equations governing problems of Mathematical Physics have analytical solutions only in cases in which the domain geometry, boundary and initial conditions are reasonably simple. Problems with arbitrary domains and fairly general boundary conditions can only be solved approximately, for example, by using numerical techniques. These techniques were strongly developed due to the presence of increasingly powerful computers, enabling the solution of complex mathematical problems.

The Boundary Element Method (BEM) is a very efficient alternative for modeling unlimited domains since it satisfies the Sommerfeld radiation condition, also known as geometric damping [Katsikadelis 2016]. This method can be used for numerically modeling the stationary behavior of 3D wave propagation in the soil and it is useful as a computational tool to aid in the analysis of soil vibration [Dominguez 1993]. A BEM based tool can be used for analyzing the vibration created by heavy machines, railway lines, earthquakes, or even to aid the design of offshore oil platforms.

With the advent of GPUs, several mathematical and engineering simulation problems were redesigned to be implemented into these massively parallel devices. However, first GPUs were designed to render graphics in real time, as a consequence, all the available libraries, such as OpenGL, were graphical oriented. These redesigns involved converting the original problem to the graphics domain and required expert 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 for general purpose programming. CUDA uses the concept of kernels, which are functions called from the host to be executed by GPU threads. Kernels are organized into a set of blocks composed of a set of threads that cooperate with each other [Patterson and Hennessy 2007].

The memory of a NVIDIA GPU is divided in global memory, local memory, and shared memory. Global memory is accessible by all threads, local memory is private to a thread and shared memory is low-latency and accessible by all threads in a block[Patterson and Hennessy 2007]. CUDA provides mechanisms to access all of them.

Regarding this work, this parallelization approach is useful because an analysis of a large domain requires a proportionally large number of mesh elements, and processing a single element have a high time cost. Doing such analysis in parallel reduces the computational time requied for the entire program because multiple elements are processed at the same time. This advantage was provided by this research.

Before discussing any parallelization technique or results, Section 2 presents a very brief mathematical description of BEM for Stationary Elastodynamic Problems and the meaning of some functions presented in this paper. Section 3 shows how the most computational intensive 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 BEM formulation, the Boundary Integral Equation for Stationary Elastodynamic Problems can be written as:

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

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

$$Hu = Gt \tag{2}$$

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. The integral of these functions, as seen in Eq. (1), will generate the G and H matrices respectively, as is shown in Eq. (2). For computing these integrals numerically, the Gaussian quadrature can be deployed. Briefly, it is an algorithm that approximates integrals by sums as shown in equation (3)[Ascher and Greif 2011], where g is the number of Gauss quadrature points.

$$\int_a^b f(x)\mathrm{d}x \approx \sum_{i=1}^{g} w_i f(x_i) \tag{3}$$

To overcome the mentioned problem in the strong singularity, one can use the artifice known as Regularization of the Singular Integral, expressed as follows:

$$\begin{aligned} c_{ij}(\xi)u_j(\xi,\omega) + \int_S \left[t^*_{ij}(\xi,x,\omega)_{\text{DYN}} - t^*_{ij}(\xi,x)_{\text{STA}}\right] u_j(x,\omega)\mathrm{d}S(x) + \\ + \int_S t_{ij}(\xi,x)_{\text{STA}}u_j(x)\mathrm{d}S(x) = \int_S u^*_{ij}(\xi,x,\omega)_{\text{DYN}}t_j(x,\omega)\mathrm{d}S(x) \end{aligned} \tag{4}$$

Where DYN = Dynamic, STA = Static. The integral of the difference between the dynamic and static nuclei, the first term in Equation (4), 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 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 (4). Lastly, `Ghmatecd` is a routine developed to create both the H and G matrices described in equation (2). Both `Nonsingd` and `Sing_de` are called from `Ghmatecd` routine.

3. Parallelization Strategies

A parallel implementation of BEM began by analyzing and modifying a sequential code provided by [Carrion 2002]. Gprof, a profiling tool by [GNU], revealed the two most time-consuming routines: `Ghmatecd` and `Nonsingd`, with 60.9% and 58.3% of the program total elapsed time, respectively. Since most calls to `Nonsingd` were performed inside `Ghmatecd`, most of the parallelization effort was focused on that last routine.

3.1. `Ghmatecd` Parallelization

Algorithm 1 shows pseudocode for the `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 `Sing_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 of the loops in lines 2 and 3, so all iterations can be computed in parallel. Since typically a modern high-end CPU 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 typical GPU contain thousands of processors, hence even a considerable large amount of elements may not generate a workload that consumes all the device's resources. Since `Nonsingd` is the cause of the performance bottleneck of `Ghmatecd`, the main effort was implementing 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` routine can be implemented as a CUDA kernel. In a CUDA block, $g \times g$ threads are created to compute in parallel the two nested loops in lines 2 and 3, allocating spaces in the shared memory to keep the matrix buffers `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 $g < 3$. Note 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` routine can be parallelized with a simple OpenMP `Parallel for` clause, and it will compute the remaining H and G.

4. Methods

Matrix norms were used to assert the correctness of our results. Let $A \in \mathbb{C}^{m\times n}$. [Watkins 2004] defines matrix 1-norm as:

$$\|A\|_1 = \max_{1\le j\le n} \sum_{i=1}^{m} |a_{ij}| \tag{5}$$

Table 1. Data experiment set

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

All norms have the property that $\|A\| = 0$ if and only if $A = 0$. Let u and v be two numerical algorithms that solve the same problem, but in a different way. Now let y_u be the result computed by u and y_v be the result computed by v. The *error* between these two values can be measured computing $\|y_u - y_v\|$. The error between CPU and GPU versions of H and G matrices was 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 and CUDA 8.0 were used to compile the application. The main flags used in Gfortran were `-Ofast -march=native -funroll-loops -flto`. The flags used in CUDA nvcc compiler were: `-use_fast_math -O3 -Xptxas --opt-level=3 -maxrregcount=32 -Xptxas --allow-expensive-optimizations=true`.

For experimenting, there were four data samples as shown in Table 1. The application was executed for each sample 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. The machine used in all experiments had an AMD A10-7700K processor paired with a GeForce GTX980[1].

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 produced a file with total time elapsed, where a script computed averages and standard deviations for all experiments.

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. The average is meaningful as 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 2.7 and 107 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 was a performance loss in the 260

[1]Thanks to NVIDIA for donating this GPU.

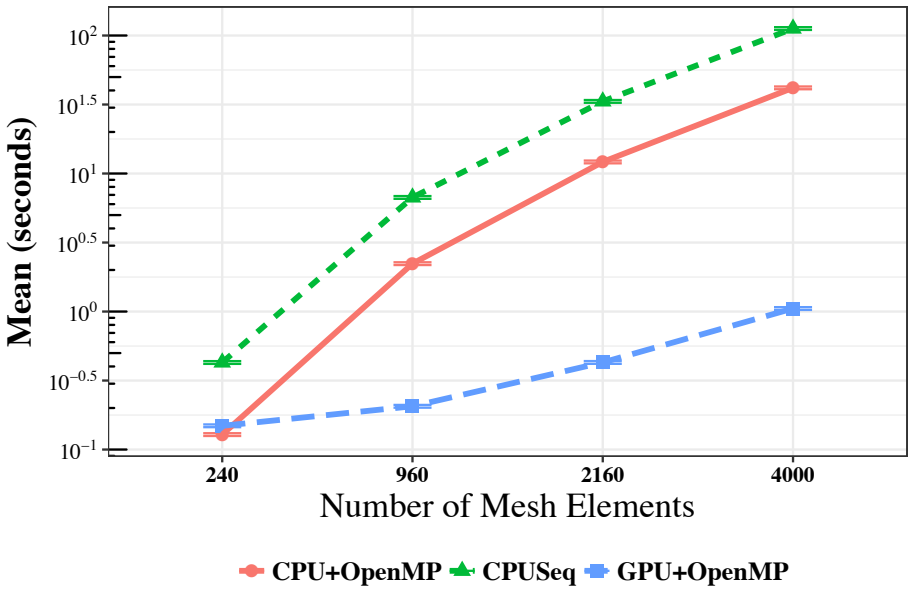

Figure 1. Time elapsed by each implementation in logarithm scale

sample between OpenMP and CUDA+OpenMP, this was caused by the high latency between CPU-GPU communication, thus the usage of GPUs may not be attractive for small meshes.

6. Future Work

There are issues related to the g described in Algorithm 2. Detailed studies are required to determine the exact value of 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 for 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.

Dominguez, J. (1993). *Boundary elements in dynamics*. Wit Press.

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

Katsikadelis, J. T. (2016). *The Boundary Element Method for Engineers and Scientists: Theory and Applications*. Academic Press.

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.