Universidade de São Paulo
Instituto de Matemática e Estatística
Bachalerado em Ciência da Computação

Giuliano Augusto Faulin Belinassi

# Optimizing a Boundary Elements Method Implementation with GPUs

São Paulo
Dezembro de 2017

# Optimizing a Boundary Elements Method Implementation with GPUs

Monografia final da disciplina
MAC0499 – Trabalho de Formatura Supervisionado.

Supervisor: Prof. Dr. Alfredo Goldman vel Lebman
[ Cosupervisor: Prof. Dr. Marco Dimas Gubitoso ]

São Paulo
Dezembro de 2017

# Resumo

Elemento obrigatório, constituído de uma sequência de frases concisas e objetivas, em forma de texto. Deve apresentar os objetivos, métodos empregados, resultados e conclusões. O resumo deve ser redigido em parágrafo único, conter no máximo 500 palavras e ser seguido dos termos representativos do conteúdo do trabalho (palavras-chave).

**Palavras-chave:** palavra-chave1, palavra-chave2, palavra-chave3.

# Abstract

Elemento obrigatório, elaborado com as mesmas características do resumo em língua portuguesa.

**Keywords:** keyword1, keyword2, keyword3.

# Contents

# Chapter 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.

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.

## 1.1 Boundary Elements Method Background

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

$$c_{ij}u_j(\xi,\omega) + \int_S t^*_{ij}(\xi,x,\omega)u_j(x,\omega)\mathrm{d}S(x) = \int_S u^*_{ij}(\xi,x,\omega)t_j(x,\omega)\mathrm{d}S(x) \tag{1.1}$$

After performing the geometry discretization, Equation (1.1) can be represented in matrix

form as:

$$Hu = Gt \tag{1.2}$$

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 $\xi$ points. The integral of these functions, as seen in Eq. (1.1), will generate the $G$ and $H$ matrices respectively, as is shown in Eq. (1.2).

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

$$\begin{aligned} c_{ij}(\xi)u_j(\xi,\omega) + \int_S \left[t^*_{ij}(\xi,x,\omega)_{\mathrm{DYN}} - t^*_{ij}(\xi,x)_{\mathrm{STA}}\right] u_j(x,\omega)\mathrm{d}S(x) + \\ + \int_S t_{ij}(\xi,x)_{\mathrm{STA}} u_j(x)\mathrm{d}S(x) = \int_S u^*_{ij}(\xi,x,\omega)_{\mathrm{DYN}} t_j(x,\omega)\mathrm{d}S(x) \end{aligned} \tag{1.3}$$

Where DYN = Dynamic, STA = Static. The integral of the difference between the dynamic and static nuclei, the first term in Equation (1.3), does not present singularity when executed concomitantly as expressed because they have the same order in the both problems.

## 1.2 Gaussian Quadrature Background

Some integrals can only be approximated by numerical methods such as the Gaussian quadrature, that means:

$$\int_{-1}^{1} f(x)\mathrm{d}x \approx \sum_{j=1}^{n} a_j f(x_j) \tag{1.4}$$

Where $a_j$ are called weights and $x_j$ are called abscissae, and these values can be computed using Legendre Polynomials, as introduced below.

**Definition 1.** *Legendre Polynomials are given by the following recurrence:*

$$\phi_j(x) = \begin{cases} 1, & \text{if } j = 0. \\ x, & \text{if } j = 1. \\ \frac{2j-1}{j} x\phi_{j-1}(x) - \frac{j-1}{j}\phi_{j-2}(x) & \text{if } j \in \mathbb{N} - \{0,1\} \end{cases} \tag{1.5}$$

It can be shown that those polynomials have the following important properties:

**Theorem 1.** *Legendre Polynomials satisfy the following properties (Ascher e Greif, 2011):*

1. *Orthogonality: for $i \neq j$, $\int_{-1}^{1} \phi_i(x)\phi_j(x)dx = 0$.*
2. *Calibration: $|\phi_j(x)| \leq 1$ for any $-1 \leq x \leq 1$, and $\phi_j(1) = 1$.*
3. *Oscillation: $\phi_j(x)$ has degree equal to $j$ and all its roots are inside $]-1;1[$.*

The proof of such theorem is beyond the scope of this work. See Ascher e Greif (2011).

**Theorem 2.** *Let $q(x)$ be a polynomial of degree $< n$. Then $q(x)$ is orthogonal to $\phi_n(x)$, that is:*

$$\int_{-1}^{1} q(x)\phi_n(x)dx = 0 \tag{1.6}$$

*Proof.* Since $\{\phi_0, \phi_1, \cdots, \phi_n\}$ is an orthogonal base of all polynomials of degree $\leq n$, then all polynomials of degree $\leq n$ can be written as linear combination of $\phi_0, \phi_1, \cdots, \phi_n$. Since $q(x)$ degree is $< n$, then:

$$q(x) = \sum_{k=0}^{n-1} \alpha_k \phi_k(x) \tag{1.7}$$

with such information, just calculate:

$$\int_{-1}^{1} q(x)\phi_n(x)\mathrm{d}x = \int_{-1}^{1} \left( \sum_{k=0}^{n-1} \alpha_k \phi_k(x) \right) \phi_n(x)\mathrm{d}x = \sum_{k=0}^{n-1} \alpha_k \left( \underbrace{\int_{-1}^{1} \phi_k(x)\phi_n(x)\mathrm{d}x}_{0,\text{orthogonality}} \right) = 0 \tag{1.8}$$

□

These two theorems above are important for quadrature's precision. Let's now present the Gaussian Quadrature.

Let $r(x)$ be a polynomial of degree $< n$. The Gaussian Quadrature must satisfy the equation below with equality.

$$\int_{-1}^{1} r(x)\mathrm{d}x = \sum_{j=1}^{n} a_j r(x_j) \tag{1.9}$$

Let's now show a simple trick that can be done with Legendre Polynomials to enhance the quadrature precision. Let $p(x)$ be a polynomial of degree $< 2n$. If we divide $p(x)$ by $\phi_n(x)$, both quotient $q(x)$ and the remainder $r(x)$ have degree $< n$ because $\phi_n(x)$ have degree equal to $n$. That means:

$$p(x) = q(x)\phi_n(x) + r(x) \tag{1.10}$$

Integrating both sides:

$$\int_{-1}^{1} p(x)\mathrm{d}x = \underbrace{\int_{-1}^{1} q(x)\phi_n(x)\mathrm{d}x}_{0,\text{by Theorem 2}} + \int_{-1}^{1} r(x)\mathrm{d}x = \int_{-1}^{1} r(x)\mathrm{d}x \tag{1.11}$$

Let's now select the abscissae points wisely. If all $x_j$ are zeroes of the Legendre Polynomials ($\phi_n(x_j) = 0$), then we would have:

$$p(x_j) = q(x_j)\underbrace{\phi_n(x_j)}_{0} + r(x_j) = r(x_j) \tag{1.12}$$

This means that the quadrature is exact to any polynomial of degree up to $2n-1$ if we could select the weights properly, thus the quadrature precision would be as good as we could approximate $f(x)$ by a polynomial of degree $2n-1$. For the weight points, Hildebrand (1987) shows that one could use:

$$a_j = \frac{2(1 - {x_j}^2)}{(n+1)^2(\phi_{n+1}(x_j))^2} \tag{1.13}$$

## 1.3 LU Decomposition Background

In many areas of science concerning numerical methods, it is necessary to find a solution that satisfies together many equations. In this subsection, we describe one of the most used algorithms to solve a specific kind of linear system of equations. Before showing such algorithms, a set of definitions and theorems are required to understand how it operates.

A *matrix* is denoted as an element of $\mathbb{C}^{m\times n}$, where $m$ is the number of rows and $n$ is the number of columns. A *vector* is an element of $\mathbb{C}^m$, where $m$ is the number of rows. Notice that a vector is a single column matrix.

**Definition 2.** *A system of linear equations is a equation of the form $Ax = b$, where $A \in \mathbb{C}^{m\times n}, b \in \mathbb{C}^n$ are known and $x \in \mathbb{C}^m$ is the only unknown in the equation.*

Although the definition above is general to any linear system, here we will explore properties of linear systems characterized by a square and nonsingular matrix $A$.

**Definition 3.** *A **square** matrix is such that the number of rows is equal to the number of columns. A matrix that is not square is called **rectangular**.*

**Definition 4.** *A matrix $A \in \mathbb{C}^{n\times n}$ is called nonsingular if and only if $Ax = 0 \Leftrightarrow x = 0$.*

Linear systems that have nonsingular matrices have a unique solution, as demonstrated below.

**Theorem 3.** *Let $A \in \mathbb{C}^{n\times n}$ be a nonsingular square matrix. Then the system $Ax = b$ admits a unique solution $x \in \mathbb{C}^n$.*

*Proof.* Let $A$ be a nonsingular square matrix and suppose, by absurd, that $Ax = b$ have two distinct solutions named $x$ and $y$. Since $y$ is also a solution, then $Ay = b$. But then $Ax - Ay = b - b = 0$. Isolating $A$ we find that $A(x - y) = 0$. But since $A$ is nonsingular, then by definition of nonsingularity we have that $(x - y) = 0$, implying that $x = y$. This result is an absurd because we supposed that $x$ and $y$ are distinct solutions. □

There is also an interesting special case of square matrices, called triangular matrices. There are two types of triangular matrices, lower triangular and upper triangular.

**Definition 5.** *An upper triangular matrix is such that all elements below the main diagonal are* 0*. Analogously, a lower triangular matrix is such that all elements above the main diagonal are* 0*.*

The matrices below are examples of triangular matrices. At the left, we have an upper triangular matrix. At the right, we have a lower triangular matrix.

$$\begin{pmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 0 & 0 & 6 \end{pmatrix} \qquad \begin{pmatrix} 1 & 0 & 0 \\ 2 & 3 & 0 \\ 4 & 5 & 6 \end{pmatrix}$$

Systems of equations with triangular matrices have an interesting property that it can be solved with an $O(n^2)$ algorithm, as illustrated in algorithms 1 and 2.

**Algorithm 1** Solves $Ax = b$, where $A$ is a lower triangular nonsingular matrix. Replaces b with the result

```
1: procedure FORWARD_SUBSTITUITION(A ∈ ℂ^{n×n}, b ∈ ℂ^n)
2:     for j := 1, n do
3:         if A[j][j] == 0 then
4:             Error: A is singular.
5:         b[j] ← b[j]/A[j][j]
6:         for i := j + 1, n do
7:             b[i] ← b[i] − A[i][j] ∗ b[j]
```

**Algorithm 2** Solves $Ax = b$, where $A$ is a upper triangular nonsingular matrix. Replaces b with the result

```
1: procedure BACKWARD_SUBSTITUITION(A ∈ ℂ^{n×n}, b ∈ ℂ^n)
2:     for j := n, 1 do
3:         if A[j][j] == 0 then
4:             Error: A is singular.
5:         b[j] ← b[j]/A[j][j]
6:         for i := 1, j do
7:             b[i] ← b[i] − A[i][j] ∗ b[j]
```

The question that rises now is now is how can we reduce a square nonsingular matrix $A$ to triangular matrices. This is what LU with partial pivoting does, it decomposes $A$ in three matrices $P^\intercal LU$, where $P$ is a pivoting matrix, $L$ is lower triangular and $U$ is upper triangular. Briefly, a pivoting matrix is such that when operated with a matrix, it interchanges its columns; this is used to avoid divisions by 0 (Watkins, 2004). Algorithm 3 illustrates how $A$ can be decomposed in such matrices.

**Algorithm 3** Decomposes $A$ in $P^\intercal LU$, $P$ is a permutation matrix stored in a vector. Stores $L$ and $U$ over $A$.

```
1: procedure LU_PARTIAL_PIVOTING(A ∈ ℂ^{n×n})
2:     Allocate P ∈ ℕ^{n−1}
3:     for k := 1, n − 1 do
4:         amax ← max{|A[k][k]|, |A[k + 1][k]|, ··· , |A[n][k]|}
5:         if amax == 0 then
6:             Error: A is singular.
7:         m ← smaller integer ≥ k that |A[m][k]| == amax
8:         P[k] ← m
9:         if m ≠ k then
10:            Swap column m and k
11:        for i := k + 1, n do
12:            A[i][k] ← A[i][k]/A[k][k]
13:        for j := k + 1, n do
14:            for i := k + 1, n do
15:                A[i][j] ← A[i][j] − A[i][k] ∗ A[k][j]
16:    if A[n][n] == 0 then
17:        Error: A is singular.
```

Using the fact that $A = P^\intercal LU$, one can solve $Ax = b$ by solving $P^\intercal LUx = b$. Since $P$ is a permutation matrix, then its inverse $P^{-1} = P^\intercal$ (Watkins, 2004) and now one must solve $LUx = Pb$. Let $y = Ux$. Solving $Ly = Pb$ will result in a numerical value to $y$. Solving $Ux = y$ will finally assert $x$.

Since the cost of decomposing $A$ into $P^\intercal LU$ is $O(n^3)$, the time required to compute $P^\intercal b$ is, naively, $O(n^2)$ and the time required to solve the two triangular systems is $O(n^2)$, then the cost of solving $Ax = b$ is $O(n^3)$.

## 1.4 Parallel Programming Background

Imagine the following scenario: You have to build a bridge to connect two parts of a city, named A and B, that are separated by a river. Let's say that if a single person builds this bridge from A to B, the time required to do so is $t$. How can we build this bridge faster? If we have another man building the same bridge in parallel to you but from B to A and connect it at the middle of the river, then the time required is $t/2$. This silly example captures the essence of parallel computing. How can we use multiple processors or multiple machines with some coordination to archive the same purpose? From this example comes the concept of speedup, as presented in the following definition.

**Definition 6.** *Let $T_1$ be the time consumed to complete a task using one processor. Let $T_{||}$ be the time consumed to complete the same task using $n$ processors. The speedup $S_n$ is calculated by:*

$$S_n = \frac{T_1}{T_{||}} \tag{1.14}$$

There are various computer architectures designed to handle parallel computing, as described by Flynn Taxonomy (Pacheco, 2011), but remarks are necessary to two architectures.

1. SIMD: Stands for Single Instruction, Multiple Data. This refers to vectorized processors allowing a single operation to be executed in a vector content. Examples are Intel SSE and GPUs, although GPUs are not a pure SIMD, as described later.

2. MIMD: Stands for Multiple Instruction, Multiple Data. This refers to independent multicore systems, capable of executing tasks asynchronously. Here is located both shared memory systems and distributed memory systems. Shared memory systems are an example of it, where various processors reads and writes to a shared memory. For such systems, a collection of directives called OpenMP can be used to explore its parallel capabilities with little changes in the original code. (Pacheco, 2011)

An important concept to parallel computing in shared memory systems is the concept of processes and threads. A process is an abstraction layer that allows a program to have the perception that it is running alone on a computer, although the process is controlled by the operating system such as Windows. Every process has its own resources and multiple execution units (Tanenbaum, 2009). A single execution unit is called a thread.

Although OpenMP uses simple pragmas to take advantage of multiple CPUs in a shared memory system by creating multiple threads within a process, GPUs requires severe modifications in the original code to run in such devices. NVIDIA developed CUDA for this purpose.

### 1.4.1 CUDA Programming

CUDA is an acronym to Compute Unified Device Architecture and it is an extension to C++ language that allows running general purpose code in NVIDIA GPUs. Its execution is divided into a host (CPU) and device (GPU) parts. For the device part, CUDA uses the concept of kernels, which are functions called from the host to be executed by GPU threads. The memory space is also separated between the host and the device, and data must be manually transferred from host to device in order to allow a device to access them, or from device to host to retrieve results computed by the GPU. Figure 1.2 illustrates the execution flow of a CUDA program.

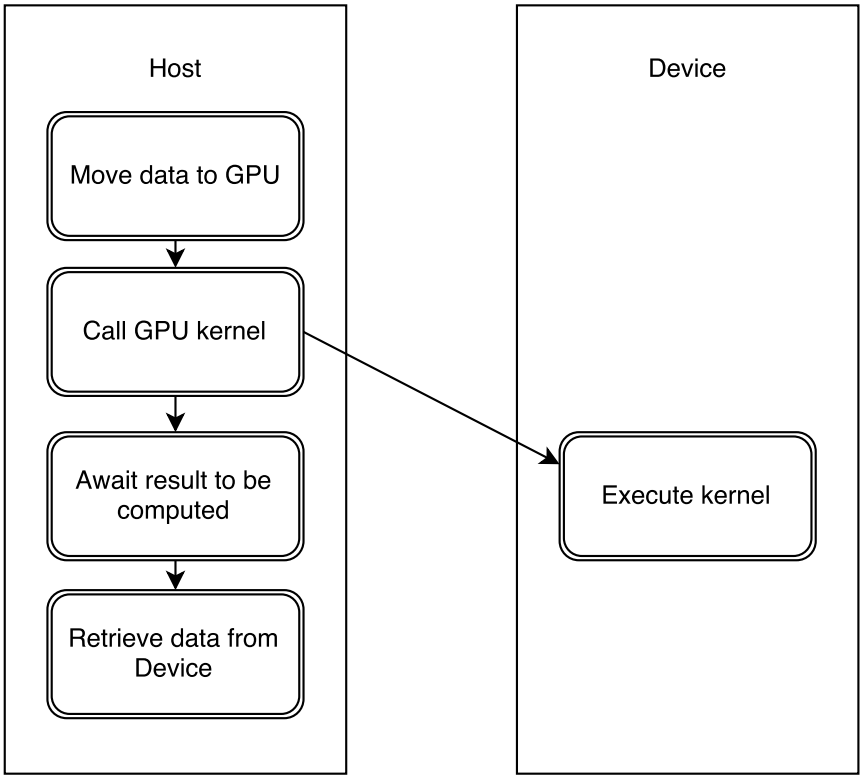


**Figure 1.1:** *A typical CUDA execution flow.*

CUDA kernels are organized into a set of blocks composed of a set of threads that cooperate with each other. The number of threads within a block and the number of blocks are specified at the launch moment, thus it can be adjusted to match the problem input size. The GPU's memory hierarchy also reflects this organization (Kirk e Wen-Mei, 2016), since it is divided into four types:

1. Registers: Private to each thread, it is used to hold frequently accessed data.
2. Shared Memory: It is a low latency memory shared between all threads within a block.
3. Constant Memory: A read-only memory that provides short latency and high bandwidth access.
4. Global Memory: A memory located outside the GPU processor chip. It can be accessed by all threads, but it is slower than all before-mentioned memories. It is the most abundant memory in the device.

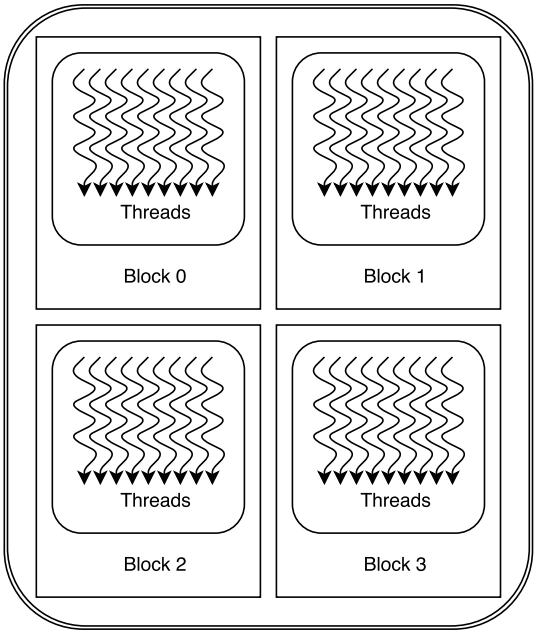


**Figure 1.2:** *Organization of a kernel execution with 9 threads per block and 4 blocks.*

It must be highlighted that all threads may not run in parallel since it is limited to the amount of Streaming Multiprocessors (SM) the device has. Without entering in details, an SM is what schedules a set of 32 threads called Warps, executing each instruction in a SIMD fashion. Hence, the number of threads running in parallel is limited by the amount of SM available in the GPU. Because a GPU contains multiples SM, it is not considered a pure SIMD device.

In order to maximize the efficiency of a CUDA application, the programmer needs to minimize the memory transfer between CPU and GPU and explore any parallelism structure of the problem, as discussed in devTalk (2012).

## 1.5 Related Works

As GPUs became famous by its massively parallel capabilities and applications were developed to explore its potential, there is no surprise that researchers would use GPUs to accelerate Boundary Elements Method implementations.

Torky e Rashed (2017) presented an implementation for generating both $H$ and $G$ matrices of equation 1.2 with GPU acceleration using two CUDA kernels: One for computing the $G$ matrix and one for computing $H$ matrix. They also manage to keep the $H$ matrix in GPU memory to avoid host to device memory transfer to solve the linear system on the GPU, managing a 108× speedup when building both $H$ and $G$ using single precision on a NVIDIA GeForce 770GTX when compared with an Intel Core i7-3770. For the linear system solving, the CPU required 2.5h to complete, whereas with the GPU it would take 4s. This implies in a 2250 times speedup. As for precision, there is no further discussion rather than the usage of 10 points for the Gaussian Quadrature and that both CPU and GPU processed data in exactly the same manner and with reliable accuracy.

# Chapter 2

# Development

The first goal of the project was to compile and execute the code with at least one sample. From this point we found a few issues such as:

- The original code provided by Carrion had issues common to legacy code. It depended on some features offered by Compaq Visual Fortran, thus the code needed modifications to be able to compile with the GFortran compiler. The code also had indentation issues and used several GOTO statements to execute simple tasks such as looping because of limitations imposed by Fortran 77.
- Fortran has a feature called implicit variable declaration that showed to be error-prone because the code had a bug due to a misspelled variable in a file named `Solfundif.for` (the `RDN` variable was misspelled `DRN`), and the compiler did not complain about the uninitialized variable.
- There was a need to call CUDA code from Fortran to avoid a complete rewrite of the application in C or C++. The Portland Group (PGI) provides a paid CUDA Fortran compiler, so another solution was required.

After all those issues were addressed, A parallel optimization of BEM began by analyzing the sequential code provided. 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.

### 2.0.1 Calling CUDA from Fortran 77

Although The Portland Group (PGI) provides a CUDA Fortran compiler, it is not free-ware software and thus might increase the application development costs. An alternative to this is to create C++ functions that are callable from Fortran and use CUDA C++ to program the kernels.

Since GFortran appends an underscore to all compiled Fortran subroutines, manually appending an underscore to a C routine makes it visible to Fortran. Since Fortran also passes all arguments as reference, then the C routine needs to expect a pointer to a variable rather than the variable itself. Now, to call C++ from Fortran it is necessary to just wrap the function with `extern "C"` clause, as exemplified below.

```c
extern "C"{
  void callable_from_fortran_(int* a, int* b){
    printf("%d\n", (*a) + (*b));
  }
}
```

```
      PROGRAM CALL_C
        INTERFACE
          SUBROUTINE callable_from_fortran(a, b)
            IMPLICIT NONE
            INTEGER a, b
          END SUBROUTINE
        END INTERFACE

        CALL callable_from_fortran(40, 2)
      END PROGRAM CALL_C
```

It must be highlighted that matrices in Fortran are stored in column-major order and indexed from 1, wherein C and C++ are stored in line-major order and indexed from 0. This means that when accessing the position `A(i,j)` from a Fortran matrix in C, one must access `A[j-1][i-1]`.

### 2.0.2 `Ghmatecd` Optimization and Parallelization

`Ghmatecd` is a routine developed to create both the $H$ and $G$ matrices for the dynamic problem described in equation (1.2). It does so by making several calls to `Nonsingd`, a routine that uses the Gaussian Quadrature as described by chapter 1.2 to compute the integral of the dynamic part in equation (1.1); and `Sing_de`, a routine that implements the artifice described by equation (1.3).

At first, we did some optimization to the original sequential code to remove unnecessary computations or memory copy. `Ghmatecd`'s code had additional matrices used in the computations that dependency analysis showed to be unnecessary. There was also a vectorial product that was computed for each column of $H$ and $G$ that were recomputed in other routines, thus it was moved to a separate function called `Normvec` and its result passed to `Ghmatecd`.

We also optimized `Nonsingd` routine by removing all calls to a procedure named `Gauleg` that were used to compute the abscissae and weight points of the Gaussian Quadrature. These points are always the zeroes of the Legendre Polynomials and there is no need to compute them more than once in the entire program, hence the result of `Gauleg` can just be passed to `Nonsingd`.

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 $3 \times 3$ matrices returned by `Nonsingd` and `Sing_de`.

**Algorithm 4** 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. With OpenMP, this can be done with a simple `pragma for` statement. 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 5 pictures a new strategy where `Nonsingd` is also computed in parallel. Let $g$ be the number of Gauss quadrature points.

**Algorithm 5** 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 $3 \times 3$, 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 $G$.

A careful examination of `Sing_de` concluded that, algorithmically, it is very similar to `Nonsingd` with a special logic to overcome singularity issues. If we merge both procedures in one, then there would be no need to compute the diagonal of $H$ and $G$ using the CPU. Algorithm 6 captures this idea.

**Algorithm 6** 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:             Allocate Hbuffer and Gbuffer, buffer of matrices 3 × 3 of size g²
6:             for y := 1, g do
7:                 for x := 1, g do
8:                     Hbuffer(x, y) ← GenerateMatrixH(i, j, x, y)
9:                     Gbuffer(x, y) ← GenerateMatrixG(i, j, x, y)
10:                    if i == j then
11:                        OvercomeSingularity(i, &Hbuffer(x, y), &Gbuffer(x, y))
12:            Gelement ← SumAllMatricesInBuffer(Gbuffer)
13:            Helement ← SumAllMatricesInBuffer(Hbuffer)
14:            G[ii : ii + 2][jj : jj + 2] ← Gelement
15:            H[ii : ii + 2][jj : jj + 2] ← Helement
```

Algorithm 6 can be implemented as a CUDA kernel analogously as `Ghmatecd_Nonsingd`. This strategy has the advantage of only using the GPU to compute the entire $H$ and $G$ and there are no idle blocks as in Algorithm 5, but tests show a significant loss of precision in this version when compared to the previous one.

### 2.0.3 `Ghmatece` Optimization and Parallelization

`Ghmatece` is a routine developed to create both the $H$ and $G$ matrices for the static problem described in equation (1.2). Unless by `RIGID` routine, a procedure designed to calculate $H$ diagonal by considerating the rigid body movement, it is very similar to `Ghmatecd` routine as described by Algorithm 7.

**Algorithm 7** Creates $H, G \in \mathbb{R}^{(3m)\times(3n)}$

```
1: procedure GHMATECE
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 ← Singge(i)          ▷ two 3 × 3 real matrices
7:             else
8:                 Gelement, Helement ← Nonsinge(i, j)
9:             G[ii : ii + 2][jj : jj + 2] ← Gelement
10:            H[ii : ii + 2][jj : jj + 2] ← Helement
11:    Rigid(H)
12: procedure RIGID(H)
13:    for i := 1, m do
14:        for j := 1, n do
15:            ii := 3(i − 1) + 1; jj := 3(j − 1) + 1
16:            if i ≠ j then
17:                H[ii : ii + 2][ii : ii + 2] ← H[ii : ii + 2][ii : ii + 2] + H[ii : ii + 2][jj : jj + 2]
```

The parallelization technique presented in Algorithm 5 can be applied to `Ghmatece` because `Nonsinge` operates in the same way as `Nonsingd`, but it computes the integral of the static nuclei in the equation (1.1). Unfortunately, since `Singge` makes 4 calls to `Nonsinge`, the strategy presented in algorithm 6 cannot be deployed efficiently because it would generate too much workload to a group of blocks.

For the `RIGID` procedure, it has no interdependency between lines and thus it can be parallelized using the `!$OMP PARALLEL DO` pragma.

### 2.0.4 Linear System Solving Optimization

In order to compute all regions surface forces and displacements, a linear system can be assembled from the matrices $H$ and $G$ from the dynamic problem, that means solving:

$$Ax = b \tag{2.1}$$

Where $A$ is a square nonsingular complex matrix. As described in chapter 1.3, the LU decomposition with partial pivoting can be deployed to solve it.

The original code used a sequential implementation provided by Compaq within a routine called `CGESV`. In order to explore multiple CPUs, a library called OpenBLAS implements this routine taking advantage of multicore architectures.

A Library named MAGMA implements the LU decomposition with GPU acceleration within a function named `magma_cgesv_gpu` that shows good results for this application.

# Chapter 3

# Methods, Results, and Conclusions

## 3.1 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:

$$\|A\|_1 = \max_{1 \leq j \leq n} \sum_{i=1}^{m} |a_{ij}| \tag{3.1}$$

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 $\|G_{cpu} - G_{gpu}\|_1$.

Gfortran 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 3.1. Each routine defines the label `gpu` in its context, but the `cpu` label means that the routine was executed only in CPU.

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.

The graphs labels consist of the implementation name concatenated with the number of OpenMP threads used, for example, `cpu8` implies that only the CPU was used with 8 OpenMP threads. All its points are the mean of the time in seconds of 30 executions, and the errorbars illustrate a 95% confidence interval.

The number of threads used were defined considerating the number of CPU cores the testing machine has. We used two machines for the tests, `BrucutuIV` and `Venus`, as defined in Appendix *A*

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 any remaining 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

**Table 3.1:** *Data experiment set*

| Number of Mesh elements | 240 | 960 | 2160 | 4000 | 14400 |
|---|---|---|---|---|---|
| Number of Boundary elements | 100 | 400 | 900 | 1600 | 6400 |

experiments set the Gauss Quadrature Points to 8.

## 3.2 Results and isolated conclusions

### 3.2.1 Ghmatecd Routine

This routine has three parallel implementations: (1) Using only CPU threads (`cpu`); (2) Using GPU for the nonsingular part and the CPU for the singular part, as described by Algorithm 5 (`gpu`); (3) Using GPU for both singular and nonsingular parts, as described by Algorithm 6 (`gpu_sing`).

#### BrucutuIV Data

The logarithmic scale graphic at Figure 3.1 illustrates the results for `BrucutuIV`. We removed the points from the graph that represents the sequential data for legibility.

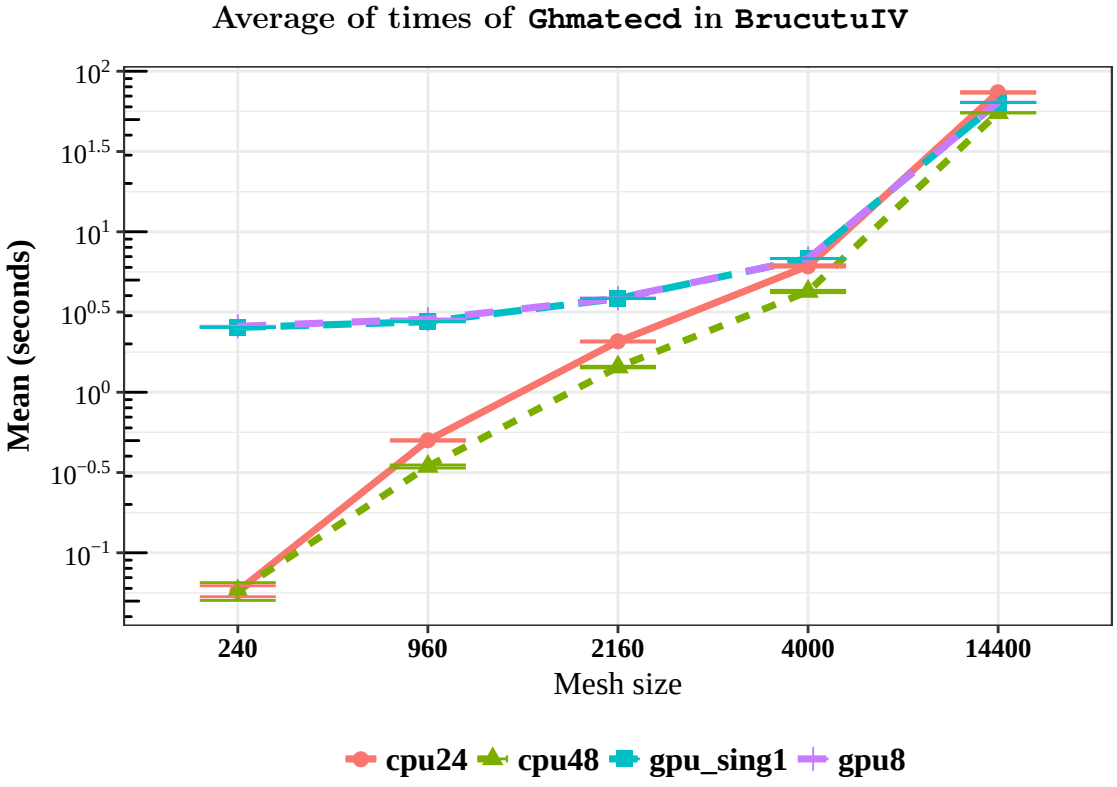


**Figure 3.1:** *`Ghmatecd`: Time elapsed by each implementation in `BrucutuIV`*

The speedup acquired in the 14400 mesh elements sample with `cpu48`, `cpu24` `gpu8` and `gpu_sing1` with respect to the sequential algorithm are 25, 19, 22 and 22 respectively in `BrucutuIV`. For this sample we can see that a single Tesla K40 was faster than a single Xeon E5-2650 v4, but slower than two of those processors.

Notice that with respect to time, `gpu8` and `cpu_sing1` behaves very similarly, but there is an interesting observation about the error, as illustrated by Figure 3.2.

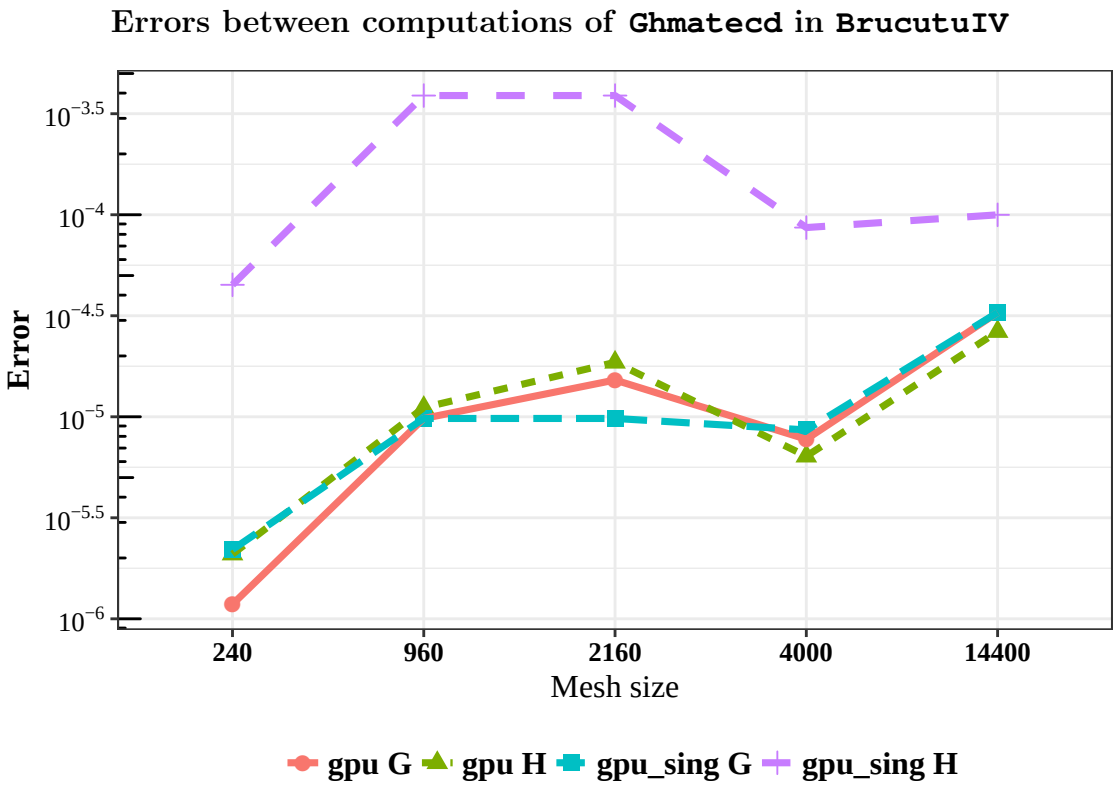


**Figure 3.2:** *`Ghmatecd`: Error of each implementation compared with the CPU implementation in `BrucutuIV`*

Since there is an error gap in the $H$ matrix between the `gpu8` and `gpu_sing1`, it suggests that most errors concentrate around computation of the singular case and it is more sensible to floating point errors, thus better precision is required here.

As a conclusion, the presented method can be used to accelerate the overall performance of the mentioned routine, and a load balancer can be developed from these results for even higher speedups because of two items: (1) similar computational time elapsed between `gpu_sing1` and `cpu24` in the 14400 sample, and (2) the fact that `gpu_sing1` only uses one CPU thread, allowing other threads to also consume workload.

### Venus Data

Here the obtained speedup was 3, 100 and 100 respectively for the `cpu4`, `gpu4` and `gpu_sing1` on the 4000 mesh sample. We could not run the 14400 sample because it requires around 16 Gigabytes of RAM, but the machine only had 8 Gigabytes. In this machine, the GeForce GTX980 was faster than the AMD A10-7700K even for the 960 sample. This was due to lower CPU-GPU memory transfer latency. The logarithm graphic at Figure 3.3 illustrates the results.

The errors followed the same behavior as discussed in `BrucutuIV Data` section, as illustrated in Figure 3.4. Notice that the errors in $H$ are worse when compared to `BrucutuIV`.

From these data, we can conclude that the presented method can also accelerate the performance in the case that there is a low number of CPU cores available, and such CPU cores can be allocated to compute the singular case to reduce errors.

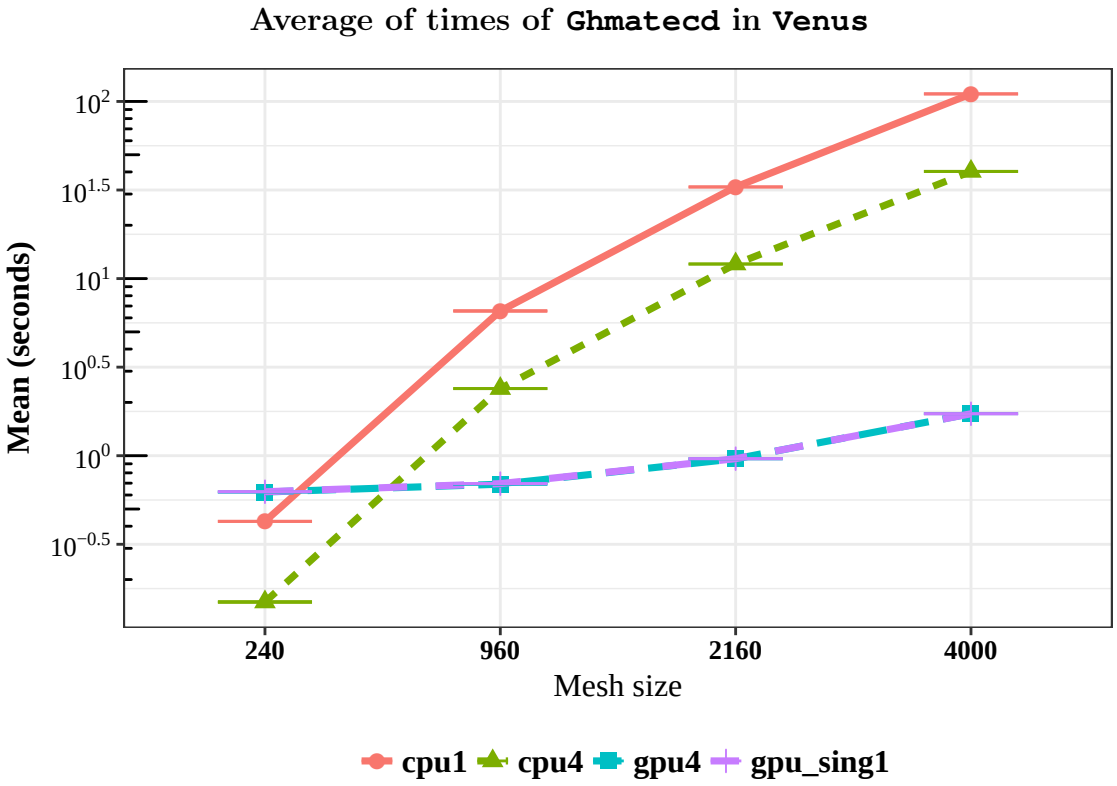


**Figure 3.3:** *Ghmatecd: Time elapsed by each implementation in Venus*

### 3.2.2 Ghmatece Routine

This routine has two parallel implementations: (1) Using only CPU threads (`cpu`); (2) Using GPU for the nonsingular part and the CPU for the singular part (`gpu`), as presented by the technique in Algorithm 5. Here we present the results for the `Ghmatece` routine wihout the `Rigid` part. We present in this way because we present a detailed analysis of the `Rigid` routine also in this chapter.

#### BrucutuIV Data

The logarithm graphic in Figure 3.5 illustrates the results for `BrucutuIV`. The speedup acquired in the 14400 mesh elements sample with `cpu24`, `cpu48`, and `gpu8` with respect to the sequential algorithm are 15, 19, and 13 respectively in `BrucutuIV`. For this sample, we can see that the Tesla K40 was slower than a single Xeon E5-2650 v4. Although `Ghmatece` routine is similar to `Ghmatecd`, it does fewer computations and thus offloading this task alone to the GPU may not be attractive.

As for the error, the 14400 was the only sample that had errors bigger than $10^{-5}$. Since the $||\bullet||_1$ grows as the number of matrix rows increases, this result was expected. Figure 3.6 illustrates the results.

As a conclusion, the GPU can be used to speedup this routine, but a load balancer is required for a better resource usage because of: (1) Smaller speedup when compared to the CPU implementation with 24 threads and (2) better than the theoretical possible speedup with only 8 CPU threads.

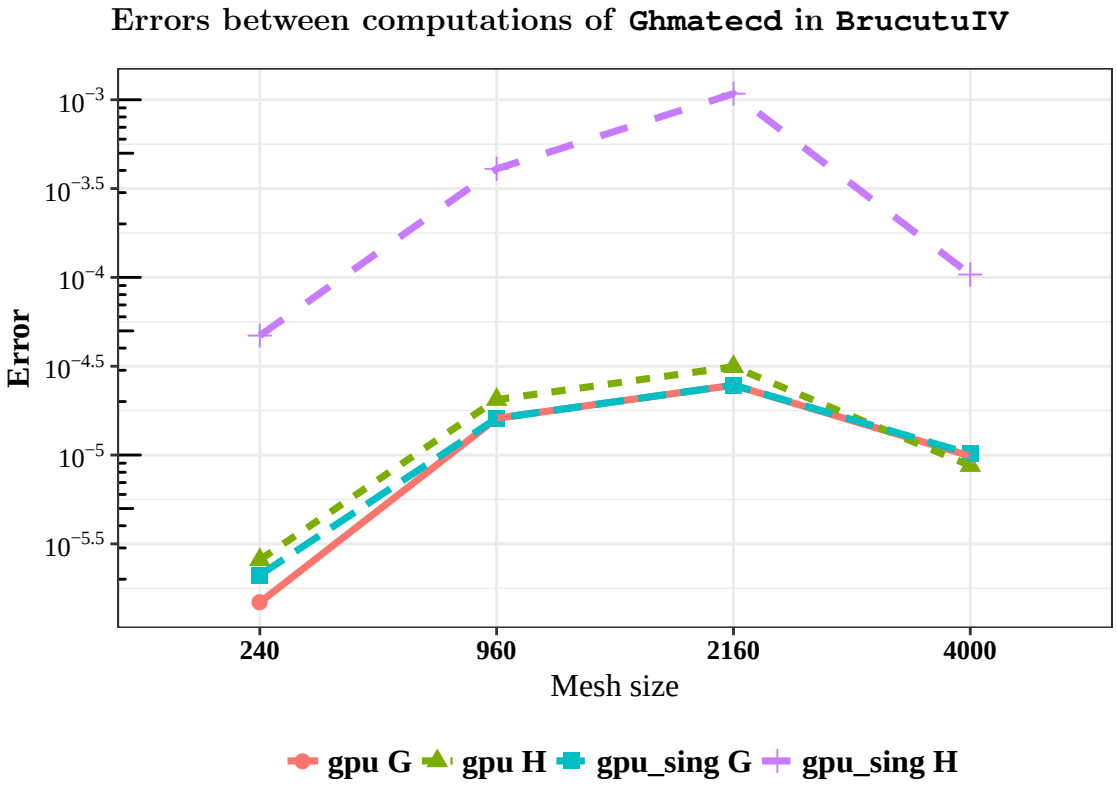


**Figure 3.4:** *Ghmatecd: Error of each implementation compared with the CPU implementation in Venus*

### Venus Data

The logarithm graphic in Figure 3.7 illustrates the results for Venus. From the figure, we can conclude that for all samples but for the 240 meshes there were speedups. The speedup acquired in the 4000 mesh elements sample with cpu4 and gpu4 with respect to the sequential algorithm are 2 and 32 respectively. As for the error, all results were below $10^{-6}$

As a conclusion, for this hardware, the presented implementation provides acceleration if the number of meshes is large enough.

## 3.2.3 Rigid routine

We only present a parallel implementation of this routine using the CPU due to the low time consumption of it when compared to the other analysed routines. We only present the BrucutuIV data because the 14400 sample was the only one that provides a meaning information about the elapsed time.

### BrucutuIV Results

Figure 3.8 illustrates the results. The speedup acquired in the 14400 mesh elements sample with cpu24 and cpu48 with respect to the sequential implementation are 12 and 13 respectively in BrucutuIV. Since all those results but the sequential are below 1$s$ and this routine is called only once in the entire program, there is no requirement for vast improvements here.

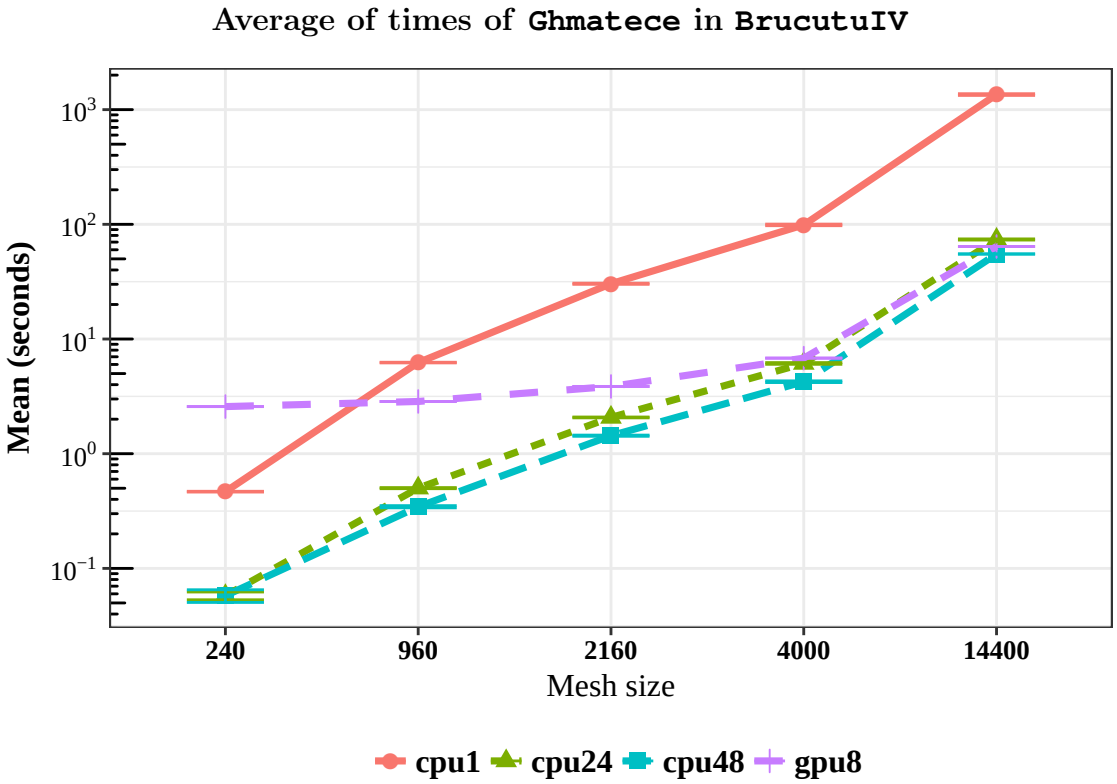


**Figure 3.5:** *Ghmatece: Time elapsed by each implementation in BrucutuIV*

### 3.2.4 Linear system solving routine

This routine has two parallel implementations: (1) Using OpenBLAS `CGESV` (`cpu`) and (2) Using MAGMA's `cgesv_gpu` routine (`gpu`).

#### BrucutuIV results

Figure 3.9 illustrates the results for `BrucutuIV`. The acquired speedup in the 14400 mesh elements sample in `cpu24`, `cpu48`, `gpu1` and `gpu48` with respect to the sequential algorithm are 10, 7, 29 and 30.

From these results, we can conclude that MAGMA's `cgesv_gpu` can be used to accelerate programs that require solving dense linear systems. Notice the slowdown when comparing `cpu24` with `cpu48`. We could not identify its causes, but we suspect of cache related issues because there are two processors in this machine, thus cache invalidation may be a problem.

#### Venus results

Figure 3.10 illustrates the results for `Venus`. The acquired speedup in the 4000 mesh elements sample in `cpu4` and `gpu4` are 2 and 38, and the `BrucutuIV` conclusions about this routine also prevails in this machine.

### 3.2.5 Final Conclusions

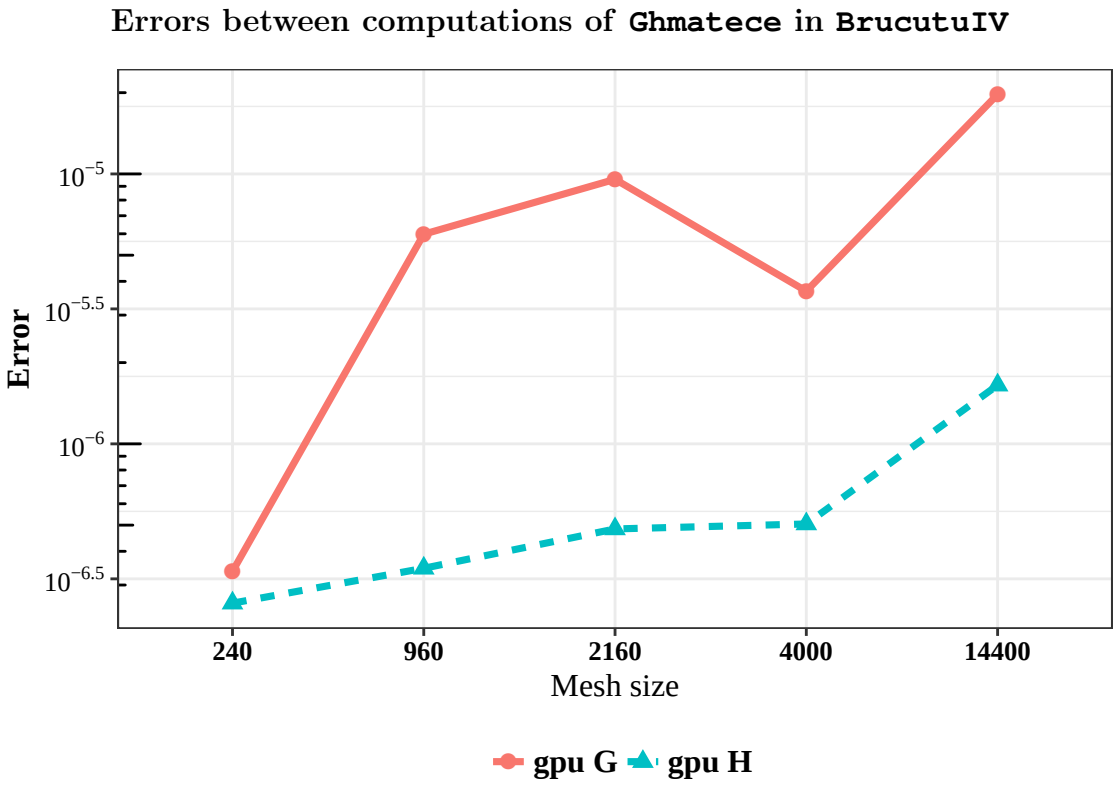


**Figure 3.6:** *Ghmatece: Error of each implementation compared with the CPU implementation in BrucutuIV*

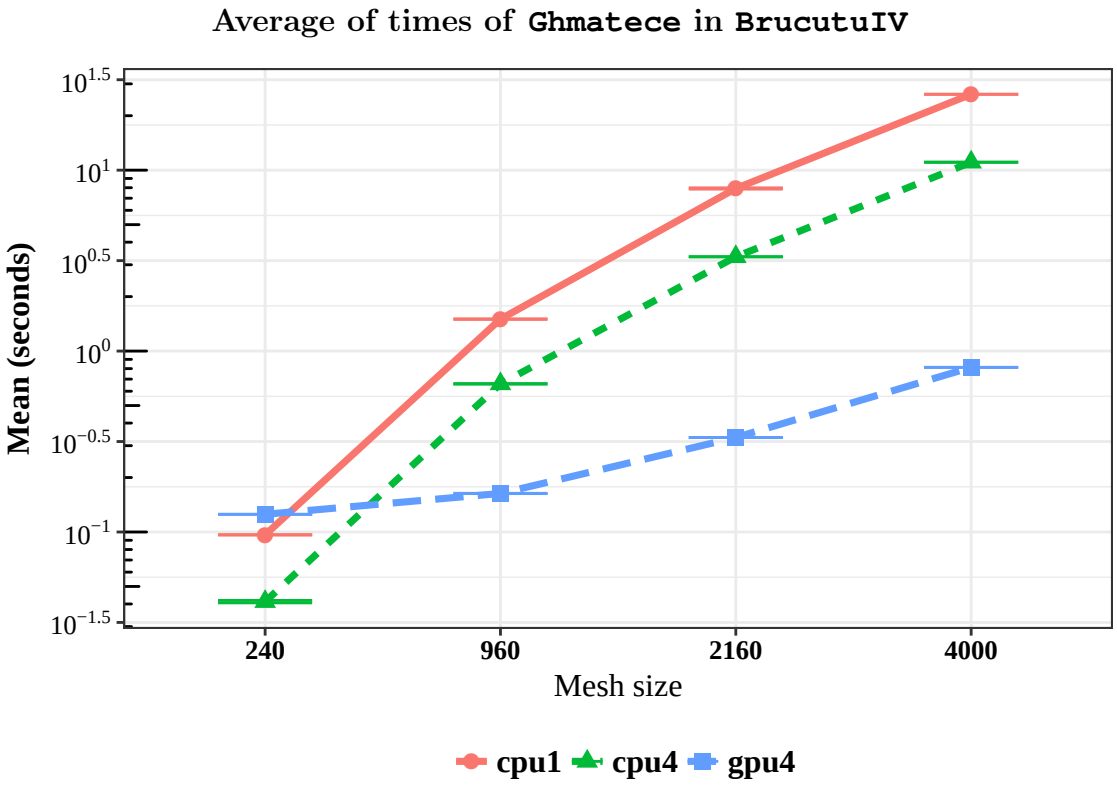


**Figure 3.7:** *Ghmatece: Time elapsed by each implementation in BrucutuIV*

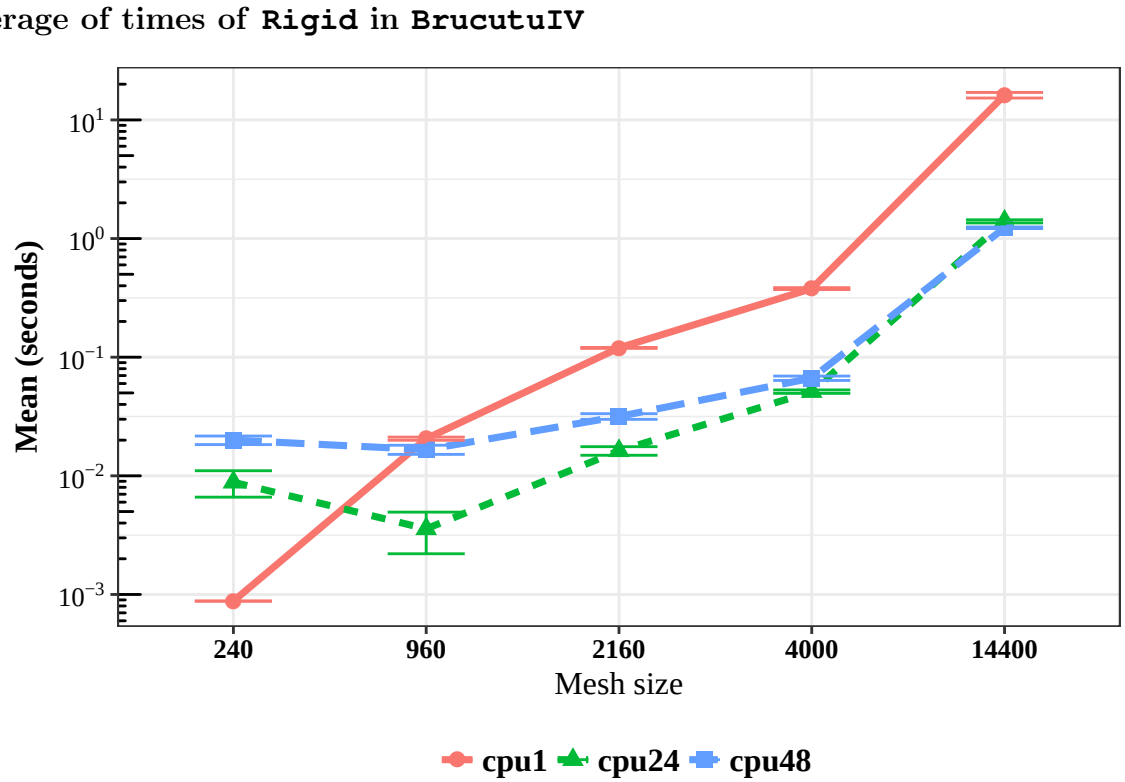


**Figure 3.8:** *Ghmatece: Time elapsed by each implementation in BrucutuIV*

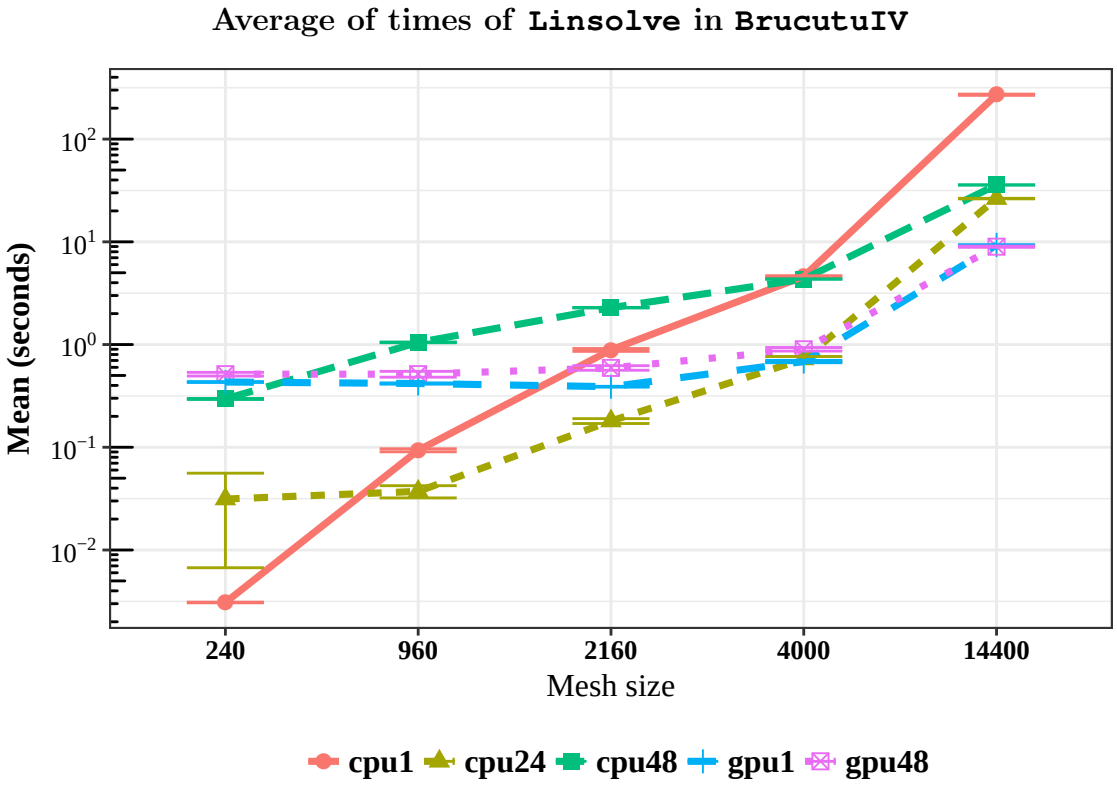


**Figure 3.9:** *Linsolve: Time elapsed by each implementation in BrucutuIV*

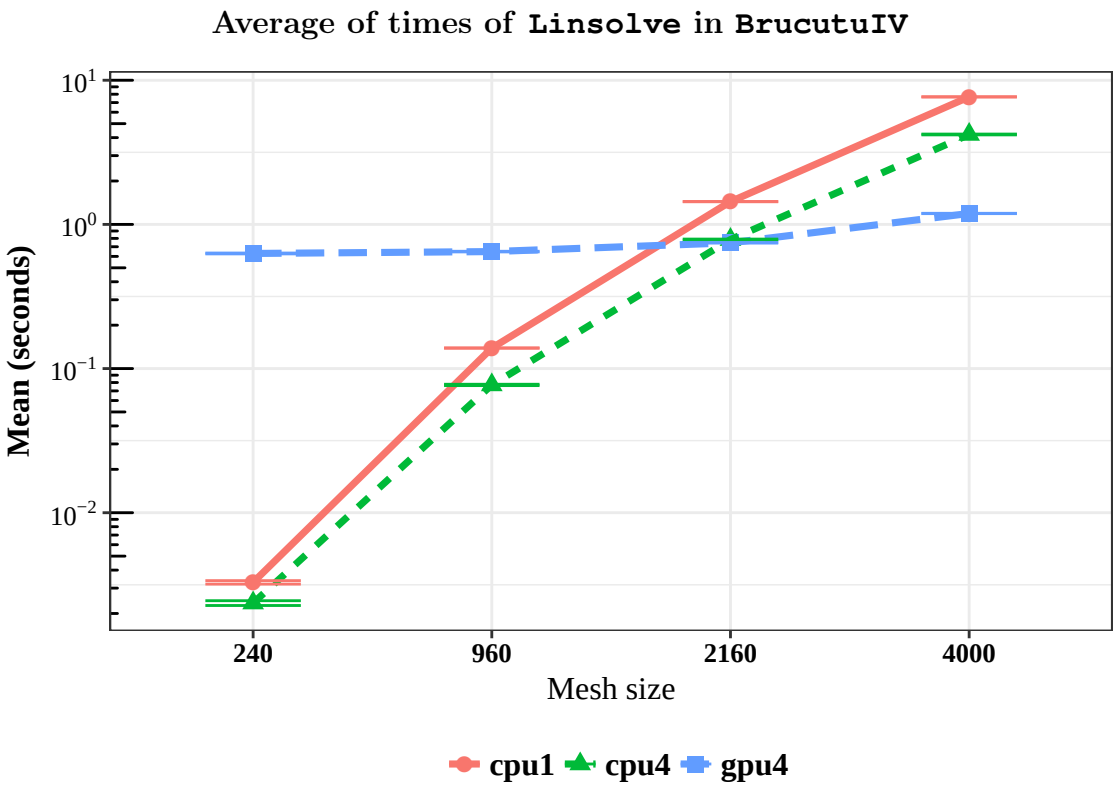


**Figure 3.10:** *`Linsolve`: Time elapsed by each implementation in `Venus`*

# Appendix A

# Computers used in tests

- Computer BrucutuIV:
  Processor: 2x Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz
  Núcleos: 24
  Threads: 48
  Number of OpenMP threads used in experiments: 1, 8, 24, 48
  GPU: NVIDIA Corporation GK110BGL [Tesla K40c]
  RAM: 378G
  Version of GFortran: 4.9.2
  Topology: Figure *A*.1
- Computer Venus:
  Processor: AMD A10-7700K Radeon R7, 10 Compute Cores 4C+6G
  Núcleos: 4
  Threads: 4
  Number of OpenMP threads used in experiments: 1, 4
  RAM: 8G
  GPU: NVIDIA Corporation GM204 [GeForce GTX 980]
  Version of GFortran: 5.4.0
  Topology: Figure *A*.2


**Figure A.1:** *Topology of BrucutuIV.*

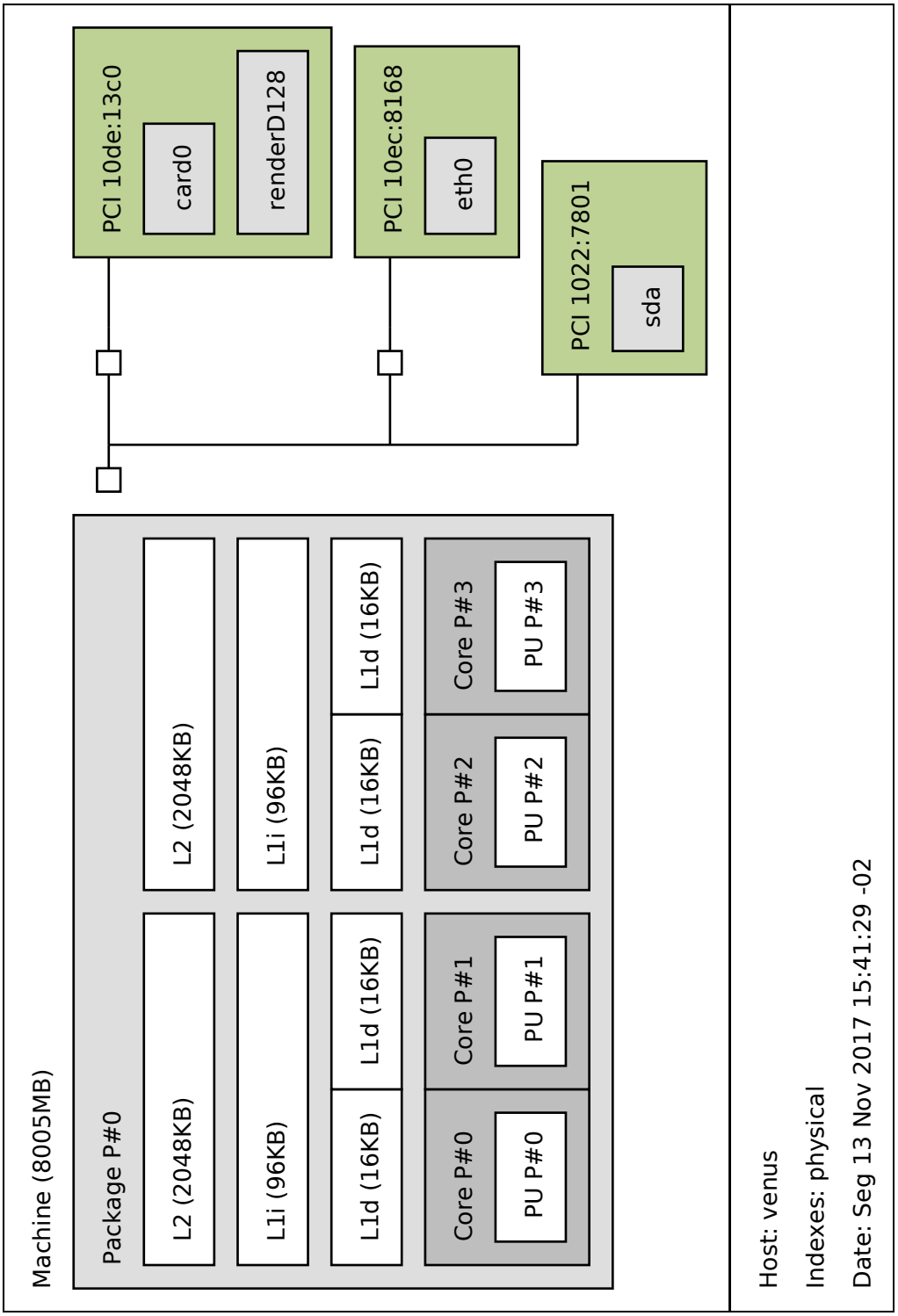


**Figure A.2:** *Topology of Venus.*

# Bibliography


**Ascher e Greif(2011)** Uri M Ascher e Chen Greif. *A first course on numerical methods*. SIAM. Citado na pág. 2

**devTalk(2012)** NVIDIA devTalk. How to optimize data transfers in cuda c/c++. https://devblogs.nvidia.com/parallelforall/how-optimize-data-transfers-cuda-cc/, 2012. Accessed: 2017-08-06. Citado na pág. 8

**Dominguez(1993)** Jose Dominguez. *Boundary elements in dynamics*. Wit Press. Citado na pág. 1

**GNU()** GNU. Gnu binutils. https://www.gnu.org/software/binutils/. Accessed: 2017-05-08. Citado na pág. 9

**Hildebrand(1987)** Francis Begnaud Hildebrand. *Introduction to numerical analysis*. Courier Corporation. Citado na pág. 3

**Katsikadelis(2016)** John T Katsikadelis. *The Boundary Element Method for Engineers and Scientists: Theory and Applications*. Academic Press. Citado na pág. 1

**Kirk e Wen-Mei(2016)** David B Kirk e W Hwu Wen-Mei. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann. Citado na pág. 7

**Pacheco(2011)** Peter Pacheco. *An introduction to parallel programming*. Elsevier. Citado na pág. 6

**Tanenbaum(2009)** Andrew Tanenbaum. *Modern operating systems*. Pearson Education, Inc.,. Citado na pág. 6

**Torky e Rashed(2017)** Ahmed A Torky e Youssef F Rashed. Gpu acceleration of the boundary element method for shear-deformable bending of plates. *Engineering Analysis with Boundary Elements*, 74:34–48. Citado na pág. 8

**Watkins(2004)** David S Watkins. *Fundamentals of matrix computations*, volume 64. John Wiley & Sons. Citado na pág. 5, 6, 15