UNIVERSIDADE DE CAXIAS DO SUL
CENTRO DE COMPUTAÇÃO E TECNOLOGIA DA INFORMAÇÃO
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

LUCAS SIGNOR SCHWOCHOW

Utilização de computação híbrida no desenvolvimento de um modelo computacional para a simulação da elasticidade tridimensional

André Luis Martinotto
Orientador

Paulo Roberto Linzmaier
Coorientador

Caxias do Sul, Dezembro de 2013

Utilização de computação híbrida no desenvolvimento de um modelo computacional para a simulação da elasticidade tridimensional

por

Lucas Signor Schwochow

Projeto de Diplomação submetido ao curso de Bacharelado em Ciência da Computação do Centro de Computação e Tecnologia da Informação da Universidade de Caxias do Sul, como requisito obrigatório para graduação.

Projeto de Diplomação

Orientador: André Luis Martinotto
Coorientador: Paulo Roberto Linzmaier
Banca examinadora:

Ricardo Vargas Dorneles
CCTI/UCS
André Gustavo Adami
CCTI/UCS

Projeto de Diplomação apresentado em
5 de Julho de 2013

Daniel Luís Notari
Coordenador

"Imagination is more important than knowledge"
Albert Einstein

AGRADECIMENTOS

Primeiramente gostaria de agradecer a minha família pela educação que recebi e pela oportunidade de entrar em uma universidade. Pai, mãe: Sou grato por tudo que fizeram, amo muito vocês.

Quero agradecer aos meus amigos que de alguma forma me ajudaram nesta jornada, e aos professores que foram capazes de transmitir o conhecimento para mim, em especial aos que ajudaram na confecção deste trabalho.

A todos vocês, minha sincera gratidão.

Lucas Signor Schwochow

SUMÁRIO

LISTA DE ACRÔNIMOS

API	*Application Programming Interface*
GPU	*Graphic Processing Unit*
CPU	*Central Processing Unit*
EDP	*Equação Diferencial Parcial*
GPGPU	*General-Purpose Computation on Graphics Processing Units*
MEF	*Método dos Elementos Finitos*
MDF	*Método das Diferenças Finitas*
MVF	*Método dos Volumes Finitos*
CUDA	*Compute Unified Device Architecture*
OpenGL	*Open Graphics Library*
DirectX	*Microsoft DirectX*
OpenCL	*Open Computing Language*
SM	*Streaming Multiprocessor*

NOTAÇÃO

$\mathbf{x}$: Vetor coluna com componentes x_1, x_2 ... x_i (Minúsculas em negrito denotam vetores).

$\mathbf{M}$: Matriz (Maiúsculas em negrito denotam matrizes).

$\mathbf{M}_{m\times n}$: Matriz com m linhas e n colunas.

$\mathbf{K}^e$: Matriz de rigidez do elemento de índice e.

$\mathbf{K}$: Matriz de rigidez de um único elemento.

$\mathbf{K}_g$: Matriz global de rigidez.

x, F: letras (maiúsculas e minúsculas) em *itálico* representam valores escalares, com exceção do N (ver abaixo).

N_e: Funções de forma. ex: N_1, N_2, N_3.

x_1, y_2, z_3: Representam componentes de um dado vetor.

ϵ_x, γ_{xy}: Representam um escalar que está relacionado com as coordenadas (x,y,z) do plano tridimensional. Como exemplo pode-se citar ϵ_x que representa a deformação com relação ao eixo x e γ_{xy} que representa a deformação com relação ao plano (x, y) (deformações de cisalhamento).

$\mathbf{K}^T$ e $\mathbf{v}^T$: Representam a transposta da matriz $\mathbf{K}$ e a transposta do vetor $\mathbf{v}$, respectivamente.

$\mathbf{K}^{-1}$: Inversa da matriz $\mathbf{K}$.
$\boldsymbol{\epsilon}$: Vetor com as componentes indicando as deformações em cada direção de um plano tridimensional $\{\epsilon_x, \epsilon_y, \epsilon_z, \gamma_{xy}, \gamma_{yz}, \gamma_{zx}\}$.

$\boldsymbol{\sigma}$: Vetor com as componentes indicando as tensões em cada direção de um plano tridimensional $\{\sigma_x, \sigma_y, \sigma_z, \tau_{xy}, \tau_{yz}, \tau_{zx}\}$.

$\boldsymbol{J}$: Matriz *Jacobiana*

$\mathbf{K}_g$: Matriz global de rigidez.

$\{\boldsymbol{u_e}\}$: Conjunto de vetores $\boldsymbol{u}$, indexados por e.

AD ou $\mathbf{A} \times \mathbf{D}$: Representam a multiplicação entre as matrizes **A** e **D**, a segunda notação é utilizada nos algoritmos apresentados neste trabalho.

$\boldsymbol{vu}$ ou $\boldsymbol{v} \cdot \boldsymbol{u}$: Representam o produto escalar entre os vetores $\boldsymbol{v}$ e $\boldsymbol{u}$, a segunda notação é utilizada nos algoritmos apresentados neste trabalho.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE TRECHOS DE CÓDIGO

RESUMO

Com a popularização da *GPU* (*Graphic Processing Unit*) como uma plataforma para computação de propósito geral, novas possibilidades surgiram no campo da simulação numérica. Áreas como a física, engenharia, química e biologia encontram nesta arquitetura uma alternativa para a execução de aplicações que necessitam de um alto desempenho computacional.

Dentre os problemas da área da engenharia que necessitam de um alto poder computacional, destaca-se a elasticidade tridimensional. Desta forma, neste trabalho foi desenvolvido um modelo computacional para a simulação desse problema. Esse modelo foi desenvolvido através do método dos Elementos Finitos, utilizando hexaedros como elementos básicos. Além disso, esse modelo foi desenvolvido de forma a trabalhar somente sobre domínios com geometria regular simples.

Na primeira parte desse trabalho foi desenvolvido uma implementação sequencial desse modelo com auxilio da biblioteca MFEM. Já na segunda parte foi desenvolvida uma implementação em *GPU* com a tecnologia *CUDA*, seguindo o modelo de computação híbrida, ou seja, trabalhando com a *CPU* (*Central Processing Unit*) e a *GPU* com o objetivo de diminuir o tempo necessário para a execução do modelo.

Por fim, foram realizados testes com as implementações que mostram um aumento de desempenho da implementação em *GPU* quando o número de elementos da malha é elevado.

Palavras-chave: MEF, Elasticidade, GPU.

Use of hybrid computing to develop a computational model for the simulation of three-dimensional elasticity

ABSTRACT

Given the popularization of *GPU* as a platform for general purpose computing, new possibilities have emerged in the field of numerical simulation. Areas such as physics, engineering, chemistry and biology found in this architecture a new ally for executing applications that require a high computational performance.

Among the problems in the area of engineering that require high computational power, there is the three-dimensional elasticity. Thus, in this work will be developed a computational model for simulation of this problem.

This model was developed using the finite element method, using elements such as hexahedra. Furthermore, this model has been developed to work only on domains with simple regular geometry.

In the first part of this work a sequential implementation of this model with the aid of MFEM library was developed. In the second part was developed an implementation on *GPU* with *CUDA* technology, following the model of hybrid computing, ie, working with the *CPU* and *GPU* with the aim of reducing the time required for model execution.

Finally, tests showed that the *GPU* implementation has increased performance when the number of mesh elements is high.

Keywords: FEM, Elasticity, GPU.

1 INTRODUÇÃO

A simulação computacional é uma abordagem utilizada para analisar o comportamento de um sistema quando um experimento real é economicamente inviável ou impossível de ser realizado. Esse tipo de experimentação ganhou notória atenção devido principalmente ao avanço da tecnologia computacional. De fato, a simulação computacional vem sendo utilizada com sucesso em diversas áreas, sendo que dentre essas áreas destacam-se a física, engenharia, química e biologia, que utilizam esse tipo de ferramenta para a análise de um determinado sistema, propriedades de materiais, entre outros (NIEMANN; HUDERT; EYMANN, 2010).

Frequentemente, os modelos simulados são representados matematicamente por Equações Diferenciais Parciais (*EDPs*) que, em geral, não apresentam uma solução analítica disponível. Desta forma, a solução destas equações é realizada através da discretização do domínio físico, gerando um conjunto finito de pontos (conhecido como malha), sendo que nesses pontos os termos das *EDPs* são aproximados a cada unidade de tempo (iteração). Desta forma, a cada iteração da simulação tem-se uma aproximação do resultado (DORNELES et al., 2008).

A obtenção de resultados de alta qualidade numérica depende, além do método de discretização adotado, do número de pontos da malha. Quanto maior o número de pontos da malha, maior será a precisão do resultado. Em contrapartida, maior será o poder computacional necessário. Desta forma, para a realização de simulações realísticas de grande escala espaço-temporal é necessário um alto poder computacional (DORNELES et al., 2008).

Uma alternativa para solucionar este problema é a utilização da computação paralela, ou seja, a divisão de uma tarefa em partes menores e a distribuição dessas entre os processadores disponíveis. Assim, os processadores executarão essas tarefas em paralelo e, consequentemente, tem-se uma redução no tempo de processamento (DORNELES et al., 2008) e (MEDINA; CHWIF, 2010).

De acordo com HAGER; WELLEIN (2012), as arquiteturas paralelas podem ser divididas em dois grupos: arquiteturas com memória compartilhada e arquiteturas com memória distribuída. As arquiteturas com memória compartilhada são com-

postas por duas ou mais *Central processing units* (*CPUs*) que acessam uma mesma área de memória (TANENBAUM, 2007). Exemplos desta arquitetura são os processadores Ivy Bridge (INTEL, 2013) da Intel, que podem apresentar dois, quatro ou seis núcleos. Já as arquiteturas de memória distribuída são compostas por dois ou mais processadores que não compartilham memória, ou seja, cada processador possui sua própria memória, não podendo acessar diretamente a memória do outro processador. Desta forma, a comunicação entre os processadores é feita pela troca de mensagens através de uma interface de rede (HAGER; WELLEIN, 2012). Como exemplo desta arquitetura tem-se o Condor Cluster (GAMASUTRA, 2010) que é formado por 1.760 unidades do Playstation 3.

Recentemente, com o advento da *GPGPU* (*General-Purpose Computation on Graphics Processing Units*), teve-se uma aumento considerável na utilização das *GPUs* (*Graphic Processing Units*) em aplicações científicas (SANDERS; KANDROT, 2010). Os resultados obtidos com a utilização da computação híbrida (isto é, a combinação da *CPU* (*Central Processing Unit*) e *GPU*) demonstraram que essa é uma alternativa viável para a execução de aplicações que necessitam de alto poder computacional (NVIDIA, 2013a). De fato, atualmente, muitos dos computadores mais rápidos do mundo são compostos por *GPUs*, sendo um exemplo o TSUBAME 2.0 da Tokyo Tech no Japão (MCINTOSH-SMITH, 2011).

Dentre as aplicações que necessitam um alto poder computacional destaca-se a simulação da elasticidade tridimensional, que consiste no objeto de estudo deste trabalho. Mais especificamente este trabalho apresenta como principal objetivo o desenvolvimento de um modelo computacional para a simulação da elasticidade tridimensional, sendo que esse modelo será desenvolvido utilizando o paradigma de computação híbrida. A simulação da elasticidade tridimensional se caracteriza por simular a deformação que um sólido sofre ao ser submetido a solicitações (força, temperatura, peso, etc) externas. Essa classe de problema tem aplicação em áreas como a engenharia mecânica e civil (RIBEIRO, 2004).

Devido à natureza do problema de simulação da elasticidade tridimensional estar relacionada a um modelo matemático representado por uma *EDP*, torna-se necessário a discretização dessa *EDP* e do domínio a ser simulado. Para tanto, existem diferentes métodos para a discretização de uma *EDP* sendo que os mais conhecidos são: método dos Elementos Finitos (*MEF*), métodos das Diferenças Finitas (*MDF*) e método dos Volumes Finitos (*MVF*). Para o desenvolvimento desse trabalho será utilizado o método dos elementos finitos em uma malha estruturada. Destaca-se ainda que o modelo desenvolvido será um modelo simplificado que permita a simulação da deformação de objetos com o formato de cubo e paralelepípedo. Além disso, em uma primeira versão não será considerada a questão temporal.

1.1 Motivação

A *GPU* é um processador baseado em uma arquitetura paralela, composta por vários núcleos simples de processamento. Inicialmente seu objetivo era o processamento de dados gráficos, processamento esse que envolve basicamente operações vetoriais em ponto flutuante. De fato, por muito tempo a função das GPUs ficou restrita ao processamento de imagens que seriam renderizadas e enviadas ao monitor (SANDERS; KANDROT, 2010).

Em meados do ano de 2001, a capacidade de realização de operações de ponto flutuante das *GPUs* motivou muitos pesquisadores a adotarem soluções híbridas em seus softwares, ou seja, dividir as tarefas entre a *CPU* e *GPU*, deixando que as operações em ponto flutuante fossem realizadas pela *GPU*. Este método ficou conhecido por *GPGPU* (SANDERS; KANDROT, 2010).

O problema dessa abordagem era a necessidade do conhecimento prévio de ferramentas de uso específico da área de computação gráfica. Isso porque a manipulação da *GPU* era feita através do uso de *Application Programming Interfaces* (*APIs*) próprias para a renderização de imagens como, por exemplo, o *OpenGL* (*Open Graphics Library*) (LIBRARY, 1992) e *DirectX* (*Microsoft DirectX*) (MICROSOFT, 1995). Desta forma, para o uso da *GPU* tornava-se necessária a conversão do problema de interesse em um problema de renderização (SANDERS; KANDROT, 2010).

Uma abordagem mais robusta apareceu em 2006 com o lançamento da placa de vídeo GeForce 3, que era baseada na arquitetura *Compute Unified Device Architecture* (*CUDA*) da NVIDIA. Esta tecnologia apresentava uma interface para acesso às funções da *GPU*, removendo a dependência das *APIs* (*Application Programming Interfaces*) gráficas em aplicações de propósito geral (KIRK; W. HWU, 2012). Atualmente, existem três APIs que permitem a manipulação da *GPU*, que são: *OpenCL* (*Open Computing Language*) (KHRONOS, 2008), DirectCompute (MICROSOFT, 2008) e NVIDIA *CUDA* (*Compute Unified Device Architecture*) (NVIDIA, 2007).

De acordo com BRODTKORB et al. (2010), para que o software obtenha um aumento considerável de desempenho em uma *GPU* é necessário que o desenvolvedor considere as peculiaridades dessa arquitetura e utilize de forma adequada os recursos disponíveis na *GPU*. Alguns desafios existentes na programação com *GPU* são: o controle adequado dos três níveis de memória (registradores, memória compartilhada e memória global), tentando manter os dados utilizados na memória de acesso mais rápido; o balanceamento do número de *threads* de forma a minimizar a latência da memória; evitar, quando possível, o efeito de ramificação (*branch*), onde a ramificação criada por uma operação condicional pode adicionar um atraso

na execução de várias *threads*.

Existem muitos casos de sucesso na adoção da *GPU* por aplicações de propósito geral, sendo que alguns desses casos podem ser encontrados em (MCINTOSH-SMITH, 2011). Esses exemplos abrangem questões como a fluidodinâmica computacional, acoplamento molecular e aceleração de operações de um banco de dados. O sucesso no uso da arquitetura *GPU* nessas aplicações demonstra as vantagens desta arquitetura, e servem de motivação para o desenvolvimento deste trabalho.

1.2 Objetivos

O principal objetivo deste trabalho é o desenvolvimento de um modelo para a simulação do fenômeno de elasticidade tridimensional utilizando computação híbrida. Para que esse objetivo seja atingido os seguintes objetivos específicos devem ser realizados:

1. Definição do modelo matemático (*EDPs*) para representar a elasticidade tridimensional.
2. Discretização das *EDPs* definidas no passo anterior.
3. Implementação de um modelo sequencial para a simulação do fenômeno de elasticidade tridimensional.
4. Implementação de um modelo híbrido para a simulação do fenômeno de elasticidade tridimensional
5. Análise comparativa de desempenho entre a implementação sequencial e a implementação híbrida, sendo que para essa análise comparativa serão utilizadas medidas de tempo e *speedup*.

2 DEFORMAÇÕES ELÁSTICAS

Nesta seção será realizada uma breve descrição sobre as deformações elásticas, tratando as relações entre a tensão e a deformação sofrida pelo material. Além disso, serão tratados conceitos relacionados com a influência da variação do tipo de material e a relação existente entre uma força aplicada e a deformação recorrente (mínimo da energia potencial). Esta breve descrição será feita pois esses conceitos serão utilizadas no Capítulo 3 para a discretização das *EDPs* (*Equações Diferenciais Parciais*).

2.1 Elasticidade

Elasticidade é uma propriedade mecânica que permite aos materiais sofrerem deformações reversíveis, ou seja, ao aplicar uma força ao material ele sofrerá uma deformação proporcional a magnitude desta força e quando esta força cessar ele voltará a sua forma original (FISH; BELYTSCHKO, 2007). Na Figura 2.1 tem-se um exemplo de uma mola, onde uma força F é aplicada gerando uma deformação. A magnitude da deformação sofrida, representada por Δl, corresponde a diferença entre a forma natural da mola e sua deformação após a aplicação da força, sendo que a relação entre a força aplicada e a deformação sofrida por um material pode ser representada pela Lei de Hooke através da equação

$$F = k\Delta l \tag{2.1}$$

onde F representa a força exercida sobre a mola, k é a constante da mola (varia de acordo com tipo de material) e Δl corresponde a deformação sofrida pela mola (BONJORNO et al., 1999).

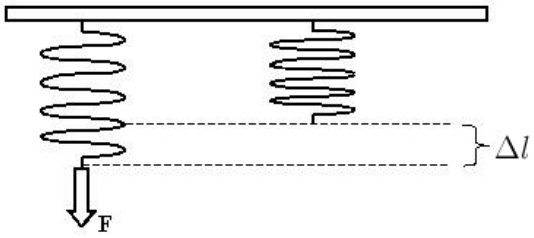

Figura 2.1: Efeito elástico: a mola sofre uma deformação Δl devido à aplicação da força F. Adaptado da referência (FISICAFACIL, 2013)

2.2 Lei de Hooke Aplicada a Materiais

A lei de Hooke é, frequentemente, demonstrada em molas. Porém, em sólidos essa característica elástica é igualmente válida, uma vez que o comportamento do material ao receber uma força externa é essencialmente igual ao de uma mola (FISH; BELYTSCHKO, 2007). Assim, de forma a contemplar as notações usuais da área de engenharia para a mecânica dos sólidos, a lei de Hooke pode ser reescrita como

$$\sigma = E\epsilon \tag{2.2}$$

onde σ representa a tensão sobre o material (medida em Pa), E é conhecida como módulo de Young (varia de acordo com o material), ϵ é a deformação sofrida pelo material (ANANTHASURESH, 2011). Por exemplo, considere a Figura 2.2, onde uma força F é aplicada a barra w. Esta força gera uma tensão σ no material, que corresponde a uma força de direção inversa à aplicada na barra. Este aumento de tensão é seguido pela deformação elástica da barra w.

A tensão também pode ser definida como a distribuição da força aplicada sobre um ponto, área, ou toda superfície do material. Por exemplo, para a Figura 2.3 tem-se uma barra unidimensional w com uma seção transversal constante A, e uma força F que é aplicada na direção axial. Neste caso a Lei de Hooke pode ser reescrita

Figura 2.2: Barra com uma força F sendo aplicada. Adaptado da referência (EBAH, 2013)

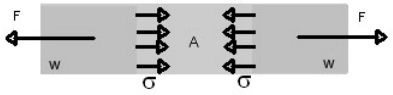

Figura 2.3: Distribuição da força sobre uma barra. Adaptado da referência (EBAH, 2013)

como

$$\sigma = F/A \tag{2.3}$$

onde A representa a área da seção transversal , σ a tensão distribuída sobre a área A e F é a força aplicada em A.

As Equações (2.2) e (2.3) são equivalentes, sendo que a Equação (2.2) é utilizada quando a magnitude da deformação e a rigidez do material (E) são conhecidas. Já a Equação (2.3) é utilizada quando os valores da força aplicada e a seção transversal do material são conhecidos.

A deformação ϵ pode ser definida como a proporção entre a deformação e seu estado natural. No caso da barra da Figura 2.3, a deformação corresponde a variação do comprimento em relação ao comprimento original da barra. Desta forma, tem-se que a deformação é dada por

$$\epsilon = \Delta l/L \tag{2.4}$$

onde L representa o comprimento original da barra.

De modo a relacionar a tensão e a deformação sofrida por um objeto tem-se o módulo de Young[3]. Assim, o módulo de Young é um parâmetro mecânico que proporciona uma medida da rigidez do material em estudo, sendo que esse pode ser representado através da equação

$$E = \sigma/\epsilon \tag{2.5}$$

As Equações (2.2), (2.3) e (2.5) permitem normalizar os valores de tensão, deformação e rigidez dos materiais, proporcionando um padrão para comparações. Um exemplo seria a comparação do limite de elasticidade de duas barras metálicas com diâmetros diferentes e comprimentos iguais. Conhecendo-se os valores das forças aplicadas e dos diâmetros dessas barras é possível calcular a tensão limite para o

[3]Também conhecido como módulo de elasticidade

rompimento das barras (TEACHENGINEERING, 2013).

2.3 Mínimo da Energia Potencial

Para a análise da tensão e deformação de estruturas é utilizado o princípio do Mínimo da Energia Potencial, ou seja, quando uma estrutura suporta alguma força externa e alcança um estado de equilíbrio, sua energia potencial deve ser mínima (ANANTHASURESH, 2011). Por exemplo, considere a Figura 2.4, onde uma corda está presa em equilíbrio entre dois pontos (Figura 2.4(a)). Já na Figura 2.4(b) tem-se as possíveis deformações que a corda pode sofrer quando forças externas são aplicadas sobre ela. Porém, para uma determinada força a corda terá uma única deformação, que corresponde a deformação de menor energia potencial (Figura 2.4(c)).

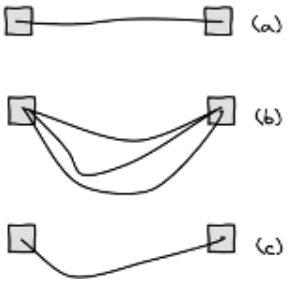

Figura 2.4: Mínimo da energia potencial: (a) Corda em estado de equilíbrio, (b) Deformações geradas por forças externas, (c) Deformação resultante de uma única força.

A definição da energia potencial, que relaciona a força aplicada com a deformação e a energia envolvida no sistema pode ser expressa através da equação

$$\Pi = \Lambda - W \tag{2.6}$$

onde Π representa a energia potencial, Λ a energia de deformação e W o trabalho realizado pela força externa. Uma vez que a energia de deformação é armazenada no elemento deformado, essa pode ser definida por

$$\Lambda = \int_v \lambda \, dv \tag{2.7}$$

onde λ é a densidade da energia de deformação (energia de deformação por unidade do volume) e v o volume do elemento.

Desta forma, energia de deformação está relacionada com a variação da tensão aplicada ao sólido e sua deformação decorrente. Na Figura 2.5 tem-se um gráfico que mostra o aumento da tensão σ em função da deformação ϵ. Como pode ser observado

nesse gráfico a tensão cresce linearmente em função da deformação, sendo que a área abaixo da função (área em cinza no gráfico) representa a densidade da energia de deformação no material (ANANTHASURESH, 2011). Desta forma, tem-se que a densidade da energia de deformação é dada por

$$\lambda = \frac{1}{2}\sigma\epsilon \tag{2.8}$$

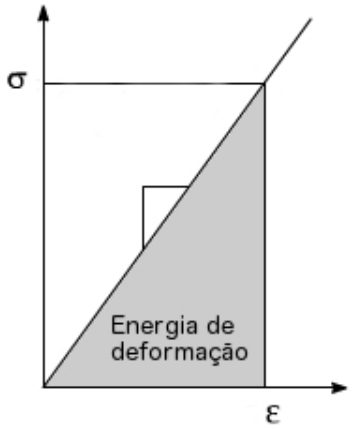

Figura 2.5: Relação entre a deformação e a tensão gerada no sólido. Adaptado da referência (NEPTEL, 2013)

Já o trabalho W é definido como o produto entre o deslocamento u da área afetada e o valor da força externa F (FAGAN, 1992), ou seja, o trabalho pode ser definido como

$$W = Fu \tag{2.9}$$

onde F corresponde à força aplicada e u representa o deslocamento da área afetada pela força F.

Expandindo a Equação (2.6) utilizando as definições apresentadas nas Equações (2.8) e (2.9) tem-se que a energia potencial é dada por

$$\Pi = \int_v (\frac{1}{2}\sigma\epsilon)\ dv - (Fu) \tag{2.10}$$

A energia potencial pode ainda ser reescrita em função da tensão (Equação 2.2). Desta forma tem-se que a energia potencial corresponde a

$$\Pi = \int_v (\frac{1}{2}E\epsilon^2)\ dv - (Fu) \tag{2.11}$$

O mínimo da energia potencial pode ser calculado a partir da derivada de Π quando a taxa de variação é zero (Equação 2.12), ou seja, neste ponto o material está em equilíbrio uma vez que não ocorre uma variação na energia potencial em

relação a deformação (FAGAN, 1992). Desta forma o mínimo da energia potencial corresponde a

$$\frac{d\Pi}{du} = 0 \tag{2.12}$$

onde u representa o deslocamento sofrido na área afetada pela força.

2.4 Coeficiente de Poisson

De acordo com (SMITH; HASHEMI, 2010), a deformação elástica longitudinal de um material é seguida pela variação das dimensões transversais. Nos elementos utilizados neste trabalho, os isotrópicos[1], esse coeficiente, que é conhecido como coeficiente de Poisson, pode ser definido como a variação da deformação lateral e longitudinal através da expressão

$$v = -\frac{\epsilon_{lateral}}{\epsilon_{longitudinal}} \tag{2.13}$$

Normalmente os valores do coeficiente de Poisson são tabelados, sendo que na Tabela 2.1 tem-se valores que serão utilizados neste trabalho. Em (ATCP, 2013) tem-se uma lista com o coeficiente de Poisson para vários materiais.

Tabela 2.1: Coeficiente de Poisson para alguns materiais

Material	Coeficiente de Poisson
Borracha	0,50
Alumínio	0,33
Cobre	0,34

A influência P do coeficiente de Poisson em sólidos isotrópicos é definida como

$$P = E\frac{1-v}{(1+v)(1-2v)} \tag{2.14}$$

onde E corresponde ao módulo de Young e v é o coeficiente de Poisson (LAKES, 2013).

[1]Propriedades com valores iguais independente da direção

3 MODELO MATEMÁTICO

Neste capítulo será apresentado o método dos Elementos Finitos (*MEF*), utilizado para o desenvolvimento deste trabalho (FISH; BELYTSCHKO, 2007). Optou-se pela utilização do MEF devido a sua vasta aplicação na simulação de problemas físicos governados por Equações Diferenciais Parciais (*EDPs*). Algumas vantagens relacionadas à aplicação deste método é a possibilidade da sua utilização em elementos formados por materiais diferentes, adaptação de elementos para representar fronteiras irregulares e elementos com tamanho variado (AZEVEDO, 2003).

Na Seção 3.1 será realizada uma breve introdução ao *MEF* descrevendo sua aplicação no campo da simulação. Na Seção 3.2 tem-se o início do processo de discretização, para tal serão apresentadas as funções de forma que irão definir o formato dos elementos finitos a serem utilizados. Já na Seção 3.3 será apresentado o processo para montagem da matriz de rigidez de cada elemento. Esta matriz será responsável por unir as propriedades do material com as suas características de forma. Na Seção 3.4 será descrito o processo de montagem do sistema de equações lineares, que será criado a partir das matrizes de rigidez dos elementos, os deslocamentos e as forças aplicadas. A partir da solução desse sistema de equações será obtida uma aproximação da solução do problema. E, por fim, na Seção 3.5 será apresentado um exemplo da aplicação do *MEF* em um elemento quadrilátero.

3.1 Método dos Elementos Finitos

O MEF é uma ferramenta numérica para resolver *EDPs*. Para tanto, inicialmente o domínio deve ser discretizado em um conjunto de elementos menores (compostos por nós e arestas). Por exemplo, considere a Figura 3.1, onde a estrutura de um elemento complexo é subdividida em elementos menores, neste caso hexaedros, que são chamadas de *elementos finitos*.

Posteriormente, funções interpoladoras são utilizadas para mapear os pontos do domínio de cada elemento finito, permitindo uma aproximação da solução da *EDP* naquele ponto. Para uma solução acurada pode-se aumentar o número de elementos

do domínio discreto gerando, desta forma, um número maior de pontos na malha. Porém, será necessário um maior poder computacional devido ao aumento no número de equações a serem resolvidas.

Além do *MEF*, existem outros métodos que podem ser utilizados para a solução de problemas que envolvem *EDPs*. Como exemplo desses métodos pode-se citar o método dos Volumes Finitos, método das Diferenças Finitas, método de Galerkin, método de Rayleigh-Ritz, entre outros (FAGAN, 1992). É importante destacar que nenhum desses métodos pode ser considerado superior ao outro, isto depende muito do tipo de aplicação, qualidade da solução desejada e capacidade computacional disponível (SILVA, 2009).

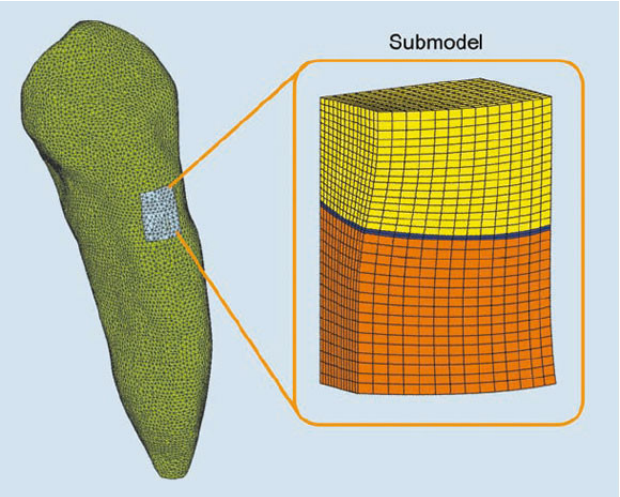

Figura 3.1: Exemplo de região discretizada com hexaedros. Adaptado da referência (BESSONE; JR., 2010)

3.2 Funções de Forma

A definição das funções de forma é uma das primeiras etapas associadas à discretização do domínio, uma vez que essas refletirão na estrutura do elemento finito. As funções de forma, normalmente, são definidas em termos das coordenadas naturais, ou seja, centralizadas dentro do plano de projeção do próprio elemento finito. Desta forma, a integração do domínio do elemento pode ser feita sobre o seu plano. Porém, posteriormente, é necessário a criação de funções responsáveis pela conversão da posição de cada nodo do plano do elemento para o plano do domínio geral da simulação (KATAN, 2008).

A definição das funções de forma para este trabalho foi feita utilizando os *polinômios de interpolação de Lagrange* (FISH; BELYTSCHKO, 2007). Para elementos hexaédricos (Figura 3.2), que foram utilizados neste trabalho, as funções de forma podem ser definidas de acordo com a Equação 3.1.

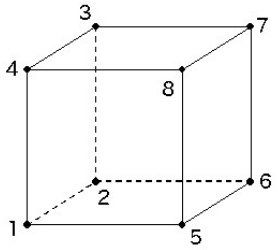

Figura 3.2: Elemento finito utilizado na discretização do domínio da simulação.

$$
\begin{aligned}
N_1 &= \frac{1}{8}(1-\xi)(1-\eta)(1+\zeta) \\
N_2 &= \frac{1}{8}(1-\xi)(1-\eta)(1-\zeta) \\
N_3 &= \frac{1}{8}(1-\xi)(1+\eta)(1-\zeta) \\
N_4 &= \frac{1}{8}(1-\xi)(1+\eta)(1+\zeta) \\
N_5 &= \frac{1}{8}(1+\xi)(1-\eta)(1+\zeta) \\
N_6 &= \frac{1}{8}(1+\xi)(1-\eta)(1-\zeta) \\
N_7 &= \frac{1}{8}(1+\xi)(1+\eta)(1-\zeta) \\
N_8 &= \frac{1}{8}(1+\xi)(1+\eta)(1+\zeta)
\end{aligned}
\tag{3.1}
$$

Na Equação 3.1 cada função N_i está relacionada a um nodo i do elemento finito, enquanto os valores ξ, η, ζ representam as coordenadas naturais do elemento. Essas coordenadas representam o domínio individual do elemento, ou seja, cada elemento possui uma representação sobre as coordenadas naturais permitindo um padrão para a aplicação das deformações. É importante destacar que as coordenadas naturais estão definidas no intervalo [-1,1] o que possibilitará a utilização do método da *Quadratura de Gauss* durante o processo de integração sobre o domínio do elemento (FISH; BELYTSCHKO, 2007).

A partir da definição das funções de forma N_i os valores dos pontos internos do elemento podem ser obtidos a partir de interpolação das funções de forma. Porém, para tanto será necessário a definição das funções que permitem mapear estes pontos para o plano do domínio geral da simulação, ou seja, o domínio compartilhado por todos os elementos finitos. Na Equação 3.2 tem-se as funções de interpolação utilizadas nesse trabalho, definidas para um plano tridimensional.

$$
\begin{aligned}
x &= N_1x_1 + N_2x_2 + N_3x_3 + N_4x_4 + N_5x_5 + N_6x_6 + N_7x_7 + N_8x_8 \\
y &= N_1y_1 + N_2y_2 + N_3y_3 + N_4y_4 + N_5y_5 + N_6y_6 + N_7y_7 + N_8y_8 \\
z &= N_1z_1 + N_2z_2 + N_3z_3 + N_4z_4 + N_5z_5 + N_6z_6 + N_7z_7 + N_8z_8
\end{aligned} \tag{3.2}
$$

Onde as funções x, y, z correspondem às funções que mapeiam os valores dos pontos de cada elemento para plano do domínio geral da simulação, e que representam respectivamente, os eixos x, y, z do plano. Os valores x_i, y_i, z_i definem a posição inicial de cada nodo i do elemento dentro do plano geral da simulação (KATAN, 2008).

3.3 Montagem da matriz de rigidez do elemento

Feita a definição das funções de forma, a etapa seguinte consiste na formulação das características do elemento, ou seja, a montagem da matriz de rigidez do elemento. Essa etapa define como as propriedades do material (Seções 2.2 e 2.4) que compõe o elemento irão afetar o domínio do elemento. Ou seja, são características que irão influenciar no comportamento de cada nodo i quando exposto a uma força e, consequentemente, irão definir a deformação sofrida pelo elemento. Para a definição das características do elemento será utilizado o conceito do Mínimo da Energia Potencial (Seção 2.3).

A definição da energia potencial de um elemento pode ser descrita como o total da energia acumulada pela deformação do elemento subtraindo-se o total de trabalho executado pela força externa no elemento (Equação 2.6). A definição da energia de deformação do elemento Λ pode ser reescrita de forma matricial como

$$
\Lambda = \int_v \frac{1}{2}(\boldsymbol{\epsilon}^T \boldsymbol{\sigma})\, dv \tag{3.3}
$$

$$
\boldsymbol{\epsilon}^T = \{\epsilon_x, \epsilon_y, \epsilon_z, \gamma_{xy}, \gamma_{yz}, \gamma_{zx}\} \tag{3.4}
$$

$$
\boldsymbol{\sigma}^T = \{\sigma_x, \sigma_y, \sigma_z, \tau_{xy}, \tau_{yz}, \tau_{zx}\} \tag{3.5}
$$

onde $\boldsymbol{\epsilon}$ é o vetor com o total das deformações do elemento, sendo $\epsilon_x, \epsilon_y, \epsilon_z$ as deformações axiais do elemento e $\gamma_{xy}, \gamma_{yz}, \gamma_{zx}$ as deformações de cisalhamento [1] do elemento. Enquanto que $\boldsymbol{\sigma}$ é o vetor com os componentes de tensão do elemento, sendo $\sigma_x, \sigma_y, \sigma_z$ as tensões axiais do elemento e $\tau_{xy}, \tau_{yz}, \tau_{zx}$ as tensões de cisalhamento do elemento. Os vetores $\boldsymbol{\epsilon}$ e $\boldsymbol{\sigma}$ podem ser representados pelas Equações 3.4 e

[1] Deformações causadas por forças aplicadas nas diagonais do elemento.

3.5, respectivamente.

É importante destacar que tensões e deformações iniciais do elemento poderiam ser adicionadas à Equação 3.3. Neste caso, por exemplo, poderia ser considerado que o elemento estaria inicialmente sofrendo solicitações de algum fator externo como, por exemplo, a variação da temperatura. Porém, neste trabalho optou-se pelo desenvolvimento de um modelo simplificado, desta forma, este valor foi removido da equação (FAGAN, 1992).

Uma vez que a tensão e a deformação do elemento podem ser relacionadas pelo módulo de Young (Equação 2.2) a Equação 3.3 pode ser reescrita como

$$\boldsymbol{\sigma} = \boldsymbol{D}\boldsymbol{\epsilon} \tag{3.6}$$

onde $\boldsymbol{D}$ é a matriz que define as propriedades do material, ou seja, relaciona o módulo de Young com o coeficiente de Poisson (descrito na Seção 2.4), e pode ser representada por

$$\boldsymbol{D} = \frac{E}{(1+v)(1-2v)} \begin{bmatrix} 1-v & v & v & 0 & 0 & 0 \\ v & 1-v & v & 0 & 0 & 0 \\ v & v & 1-v & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1-2v}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1-2v}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1-2v}{2} \end{bmatrix}$$

A Equação 3.3 pode ser reescrita em função da definição de tensão do elemento apresentada na Equação 3.6, como

$$\Lambda = \frac{1}{2} \int_v \boldsymbol{\epsilon}^T \boldsymbol{D}\boldsymbol{\epsilon} \, dv \tag{3.7}$$

Já a definição da deformação ϵ, que é utilizada na Equação 3.7, pode ser escrita através de

$$\boldsymbol{\epsilon} = \boldsymbol{B}\boldsymbol{u} \tag{3.8}$$

onde $\boldsymbol{B}$ é uma matriz de dimensões [6 x 24], composta pelas derivadas parciais das funções de interpolação N_i.

A matriz $\boldsymbol{B}$ define a deformação no elemento em relação aos deslocamentos nodais dentro do plano de projeção do domínio, ou seja, define a forma do elemento de acordo com a variação dos valores nodais no domínio da simulação, podendo ser representada através de

$$\boldsymbol{B} = \begin{bmatrix} \frac{\partial N_1}{\partial x} & 0 & 0 & \frac{\partial N_2}{\partial x} & 0 & 0 & \frac{\partial N_3}{\partial x} & 0 & 0 & \cdots & \frac{\partial N_8}{\partial x} & 0 & 0 \\ 0 & \frac{\partial N_1}{\partial y} & 0 & 0 & \frac{\partial N_2}{\partial y} & 0 & 0 & \frac{\partial N_3}{\partial y} & 0 & \cdots & 0 & \frac{\partial N_8}{\partial y} & 0 \\ 0 & 0 & \frac{\partial N_1}{\partial z} & 0 & 0 & \frac{\partial N_2}{\partial z} & 0 & 0 & \frac{\partial N_3}{\partial z} & \cdots & 0 & 0 & \frac{\partial N_8}{\partial z} \\ \frac{\partial N_1}{\partial y} & \frac{\partial N_1}{\partial x} & 0 & \frac{\partial N_2}{\partial y} & \frac{\partial N_2}{\partial x} & 0 & \frac{\partial N_3}{\partial y} & \frac{\partial N_3}{\partial x} & 0 & \cdots & \frac{\partial N_8}{\partial y} & \frac{\partial N_8}{\partial x} & 0 \\ 0 & \frac{\partial N_1}{\partial z} & \frac{\partial N_1}{\partial y} & 0 & \frac{\partial N_2}{\partial z} & \frac{\partial N_2}{\partial y} & 0 & \frac{\partial N_3}{\partial z} & \frac{\partial N_3}{\partial y} & \cdots & 0 & \frac{\partial N_8}{\partial z} & \frac{\partial N_8}{\partial y} \\ \frac{\partial N_1}{\partial z} & 0 & \frac{\partial N_1}{\partial x} & \frac{\partial N_2}{\partial z} & 0 & \frac{\partial N_2}{\partial x} & \frac{\partial N_3}{\partial z} & 0 & \frac{\partial N_3}{\partial x} & \cdots & \frac{\partial N_8}{\partial z} & 0 & \frac{\partial N_8}{\partial x} \end{bmatrix}$$

onde x, y, z são as coordenadas físicas do elemento e N_i são as funções de forma do nodo i (Equação 3.1). Já o vetor $\boldsymbol{u}$ é composto pelos deslocamentos nodais do elemento, podendo ser definido como:

$$\boldsymbol{u}^T = \{x_1, y_1, z_1, ..., x_i, y_i, z_i\} \tag{3.9}$$

onde x_i, y_i, z_i representam os deslocamentos em cada eixo (x, y, z) do nodo i.

Como exemplo, considere a Equação 3.10 onde são calculados os valores da deformação ϵ_x de um elemento hexaédrico.

$$\epsilon_x = \frac{\partial N_1}{\partial x}x_1 + \frac{\partial N_2}{\partial x}x_2 + \frac{\partial N_3}{\partial x}x_3 + \frac{\partial N_4}{\partial x}x_4 + \frac{\partial N_5}{\partial x}x_5 + \frac{\partial N_6}{\partial x}x_6 + \frac{\partial N_7}{\partial x}x_7 + \frac{\partial N_8}{\partial x}x_8 \tag{3.10}$$

A Equação 3.10 é o produto da primeira linha da matriz $\boldsymbol{B}$ pelo vetor $\boldsymbol{u}$, conforme foi definido na Equação 3.8.

Para a obtenção das derivadas parciais da Equação 3.10 torna-se necessário utilizar a matriz *Jacobiana* $\boldsymbol{J}$ (FISH; BELYTSCHKO, 2007), que é definida para o plano tridimensional como

$$\boldsymbol{J} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} & \frac{\partial z}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} & \frac{\partial z}{\partial \eta} \\ \frac{\partial x}{\partial \zeta} & \frac{\partial y}{\partial \zeta} & \frac{\partial z}{\partial \zeta} \end{bmatrix} \tag{3.11}$$

sendo x, y, z as funções de mapeamento do plano do elemento para o plano da simulação (definidas na Equação 3.2), enquanto que ξ, η, ζ são as coordenadas naturais do elemento. Desta forma, as derivadas parciais das funções de forma N_i, sobre as coordenadas físicas x, y, z, são obtidas através de

$$\begin{Bmatrix} \frac{\partial N_i}{\partial x} \\ \frac{\partial N_i}{\partial y} \\ \frac{\partial N_i}{\partial z} \end{Bmatrix} = \boldsymbol{J}^{-1} \begin{Bmatrix} \frac{\partial N_i}{\partial \xi} \\ \frac{\partial N_i}{\partial \eta} \\ \frac{\partial N_i}{\partial \zeta} \end{Bmatrix} \tag{3.12}$$

O processo de cálculo das deformações ϵ_y, ϵ_z , γ_{xy}, γ_{yz}, γ_{zx} é realizado de forma si-

milar, para tanto é necessário considerar somente a variação das coordenadas nodais $(x_i,\, y_i,\, z_i)$.

A Equação 3.7 pode ser expandida utilizando-se a definição de deformação apresentada na Equação 3.8. A partir dessa expansão tem-se que a energia de um único elemento do domínio é dada por

$$\Lambda = \frac{1}{2} \int_v \boldsymbol{u}^T \boldsymbol{B}^T \boldsymbol{D} \boldsymbol{B} \boldsymbol{u}\, dv \tag{3.13}$$

desta forma, para o cálculo da energia total do domínio será necessário efetuar a soma da energia de deformação de cada elemento (FAGAN, 1992).

Obtida a definição da energia de deformação do elemento (Equação 3.13), torna-se necessário efetuar a subtração do trabalho W (Equação 2.9). Considerando-se um nodo que sofre um deslocamento devido à aplicação de uma força, pode-se representar o trabalho executado sobre esse como o produto entre a magnitude desta força e o deslocamento resultante do nodo (FAGAN, 1992). As forças que podem ser aplicadas aos elementos são:

a) Uma força aplicada a todo o volume de um corpo como, por exemplo, a força da gravidade. Considerando X, Y, Z como as forças aplicadas em cada unidade do volume de um corpo (ou seja, em todos os nodos de um elemento), tem-se que:

$$W_b = \int_v \boldsymbol{u}^T \boldsymbol{N}^T \begin{Bmatrix} X \\ Y \\ Z \end{Bmatrix} dv \tag{3.14}$$

b) Uma força distribuída em uma determinada área de um corpo. Considerando P_x, P_y, P_z como forças paralelas aos eixos e s como a área da superfície onde essas forças serão aplicadas, tem-se:

$$W_s = \int_s \boldsymbol{u}^T \boldsymbol{N}^T \begin{Bmatrix} P_x \\ P_y \\ P_z \end{Bmatrix} ds \tag{3.15}$$

c) Uma força aplicada em um único nodo. Considerando $\boldsymbol{p}$ um vetor que representa a força a ser aplicada, tem-se:

$$W_c = \int_v \boldsymbol{u}^T \boldsymbol{p}\, dv \tag{3.16}$$

Considerando os trabalhos W_b, W_s, W_c, a Equação 2.6 pode ser escrita, para cada elemento do domínio, como

$$\Pi^e = \Lambda^e - W_b^e - W_s^e - W_c^e \tag{3.17}$$

onde e identifica o elemento.

A energia potencial mínima é dada pela Equação 3.18, onde o conjunto de vetores $\{\boldsymbol{u_e}\}$ representa a união dos deslocamentos nodais $\boldsymbol{u}$ de cada elemento e do domínio.

$$\frac{\partial \Pi}{\partial \{\boldsymbol{u_e}\}} = 0 \tag{3.18}$$

Conforme a Equação 3.18, para cada deslocamento nodal $\boldsymbol{u_e}$ considera-se que a energia potencial esteja em seu mínimo, ou seja, considerando o deslocamento $\boldsymbol{u_1}$ de um dado nodo ($e = 1$), tem-se que:

$$\frac{\partial \Pi}{\partial \boldsymbol{u_1}} = 0 \tag{3.19}$$

Expandindo a Equação 3.18 em função da energia de deformação (Equação 3.13) e trabalho executado (Equações 3.14, 3.15 e 3.16), tem-se

$$\frac{\partial \Pi}{\partial \{\boldsymbol{u_e}\}} = (\frac{1}{2}\int_v \boldsymbol{B}^T \boldsymbol{D}\boldsymbol{B}\ dv)\boldsymbol{u_e} - \int_v \boldsymbol{N}^T \begin{Bmatrix} X \\ Y \\ Z \end{Bmatrix}\ dv - \int_s \boldsymbol{N}^T \begin{Bmatrix} P_x \\ P_y \\ P_z \end{Bmatrix}\ ds - \int_v \boldsymbol{p} = 0 \tag{3.20}$$

A partir do rearranjo dos valores de força e da solução das integrais envolvidas na Equação 3.20, tem-se o sistema de equações lineares que define o comportamento elástico de um elemento finito. Podendo ser representada por

$$\boldsymbol{K}\boldsymbol{u} = \boldsymbol{f} \tag{3.21}$$

onde $\boldsymbol{K}$ é a matriz de rigidez de um elemento, representada como

$$\boldsymbol{K} = \int_v \boldsymbol{B}^T \boldsymbol{D}\boldsymbol{B}\ dv \tag{3.22}$$

e $\boldsymbol{f}$ é o vetor com as forças aplicadas ao elemento. É importante destacar que os valores desconhecidos são os valores dos deslocamentos, ou seja, $\boldsymbol{u}$ é o vetor com as incógnitas do elemento (ANANTHASURESH, 2011; FAGAN, 1992; FISH; BELYTSCHKO, 2007).

Devido às derivadas parciais das funções de forma N_i, sobre as coordenadas físicas do elemento (Equação 3.10), serem resolvidas em função da matriz *Jacobiana* (Equação 3.12) e essa estar definida sobre as coordenadas naturais do elemento, é preciso reescrever a matriz de rigidez $\boldsymbol{K}$ do elemento para ser integrada sobre as coordenadas naturais do elemento. Dessa forma, tem-se

$$\boldsymbol{K} = \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} \boldsymbol{B}(\xi,\eta,\zeta)^T \boldsymbol{D}\boldsymbol{B}(\xi,\eta,\zeta)|\boldsymbol{J}(\xi,\eta,\zeta)|\, d\xi\, d\eta\, d\zeta \tag{3.23}$$

onde $|\boldsymbol{J}|(\xi,\eta,\zeta)$ é o determinante da matriz *Jacobiana* (Equação 3.11) e ξ,η,ζ são as coordenadas naturais do elemento definidas entre [-1,1].

Na próxima seção será apresentada a montagem da matriz global de rigidez, onde as matrizes de rigidez dos elementos serão conectadas.

3.4 Montagem da matriz global de rigidez

Uma vez que as propriedades de cada elemento estejam definidas (matriz de rigidez de cada elemento), é possível agrupá-las de forma a gerar as propriedades da malha. Esse agrupamento é representado pela matriz global de rigidez. Quando uma força for aplicada sobre a malha, a matriz global de rigidez irá definir o comportamento de cada nodo da malha, gerando uma deformação correspondente à força aplicada.

Para a montagem da matriz global de rigidez é necessário agrupar a matriz de rigidez de cada elemento. Para tal é necessário converter a representação nodal de cada elemento para a representação nodal global. Neste processo os elementos compartilharão alguns nodos de acordo com a estrutura da malha. A matriz global de rigidez é representada como uma matriz [mn x mn] onde m é a quantidade de nodos na malha e n a quantidade de graus de liberdade. Por exemplo, considere o domínio apresentado na Figura 3.3.

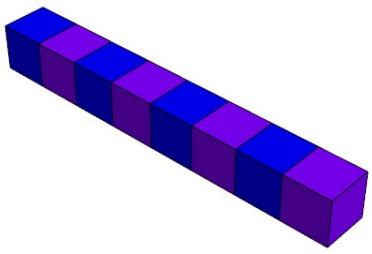

Figura 3.3: Malha com 8 elementos hexaédricos. Adaptado da referência (GLVIS, 2010)

Esse domínio é composto por 8 elementos hexaédricos, sendo que cada um deles apresenta 8 nodos. Como os elementos estão compartilhando alguns nodos tem-se uma malha com o total de 36 nodos. Uma vez que foram utilizados 3 graus de liberdade em cada nodo tem-se uma matriz global de rigidez com dimensões de [108

x 108] (FISH; BELYTSCHKO, 2007).

Assim, devido ao tamanho da matriz de rigidez, neste trabalho será demonstrado o processo de montagem de uma matriz para 2 elementos quadriláteros com apenas 2 graus de liberdade (Figura 3.4). Desta forma, será gerado uma matriz global de rigidez com dimensões [12 x 12]. Esse mesmo processo pode ser aplicado para o agrupamento das matrizes de rigidez dos elementos hexaédricos.

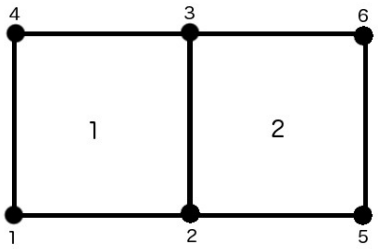

Figura 3.4: Dois elementos quadriláteros numerados no sentido anti-horário

Na Figura 3.4 é apresentada uma malha composta por 2 elementos quadriláteros, que foram numerados como 1 e 2, sendo que os nodos em cada elemento foram dispostos no sentido anti-horário. Nas Equações 3.24 e 3.25 tem-se a matriz de rigidez $\boldsymbol{K_e}$ de cada elemento e. Uma vez que cada elemento possui 4 nodos com 2 graus de liberdade, a matriz de rigidez de cada elemento terá dimensões [8 x 8].

$$\boldsymbol{K_1} = \begin{array}{c} \begin{matrix} x_1 & y_1 & x_2 & y_2 & x_3 & y_3 & x_4 & y_4 \end{matrix} \\ \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} & a_{17} & a_{18} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} & a_{27} & a_{28} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} & a_{37} & a_{38} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} & a_{47} & a_{48} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} & a_{57} & a_{58} \\ a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} & a_{67} & a_{68} \\ a_{71} & a_{72} & a_{73} & a_{74} & a_{75} & a_{76} & a_{77} & a_{78} \\ a_{81} & a_{82} & a_{83} & a_{84} & a_{85} & a_{86} & a_{87} & a_{88} \end{bmatrix} \begin{matrix} x_1 \\ y_1 \\ x_2 \\ y_2 \\ x_3 \\ y_3 \\ x_4 \\ y_4 \end{matrix} \end{array} \tag{3.24}$$

$$\boldsymbol{K_2} = \begin{array}{c} \begin{matrix} x_2 & y_2 & x_5 & y_5 & x_6 & y_6 & x_3 & y_3 \end{matrix} \\ \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} & b_{15} & b_{16} & b_{17} & b_{18} \\ b_{21} & b_{22} & b_{23} & b_{24} & b_{25} & b_{26} & b_{27} & b_{28} \\ b_{31} & b_{32} & b_{33} & b_{34} & b_{35} & b_{36} & b_{37} & b_{38} \\ b_{41} & b_{42} & b_{43} & b_{44} & b_{45} & b_{46} & b_{47} & b_{48} \\ b_{51} & b_{52} & b_{53} & b_{54} & b_{55} & b_{56} & b_{57} & b_{58} \\ b_{61} & b_{62} & b_{63} & b_{64} & b_{65} & b_{66} & b_{67} & b_{68} \\ b_{71} & b_{72} & b_{73} & b_{74} & b_{75} & b_{76} & b_{77} & b_{78} \\ b_{81} & b_{82} & b_{83} & b_{84} & b_{85} & b_{86} & b_{87} & b_{88} \end{bmatrix} \end{array} \begin{matrix} x_2 \\ y_2 \\ x_5 \\ y_5 \\ x_6 \\ y_6 \\ x_3 \\ y_3 \end{matrix} \tag{3.25}$$

A matriz global de rigidez contempla todos os nodos da malha. No exemplo da Figura 3.4 a matriz global de rigidez terá dimensões de [12 x 12]. Para mapear as matrizes de rigidez de cada elemento, será utilizada a numeração nodal presente na Figura 3.4. Essa numeração define um identificador para cada nodo, dentro do plano de projeção da simulação. A seguir é apresentada a matriz global de rigidez $\boldsymbol{K_g}$ para o domínio da Figura 3.4

$$\boldsymbol{K_g} = \begin{array}{c} \begin{matrix} x_1 & y_1 & x_2 & y_2 & x_3 & y_3 & x_4 & y_4 & x_5 & y_5 & x_6 & y_6 \end{matrix} \\ \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & 0 & 0 & a_{17} & a_{18} & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & a_{24} & 0 & 0 & a_{27} & a_{28} & 0 & 0 & 0 & 0 \\ a_{31} & a_{32} & a_{33}+b_{11} & a_{34}+b_{12} & a_{35}+b_{17} & a_{36}+b_{17} & 0 & 0 & b_{13} & b_{14} & 0 & 0 \\ a_{41} & a_{42} & a_{43}+b_{21} & a_{44}+b_{22} & a_{45}+b_{27} & a_{46}+b_{27} & 0 & 0 & b_{23} & b_{24} & 0 & 0 \\ 0 & 0 & a_{53}+b_{71} & a_{54}+b_{72} & a_{55}+b_{77} & a_{56}+b_{78} & a_{57} & a_{58} & 0 & 0 & b_{75} & b_{76} \\ 0 & 0 & a_{63}+b_{81} & a_{64}+b_{82} & a_{65}+b_{87} & a_{66}+b_{88} & a_{57} & a_{58} & 0 & 0 & b_{85} & b_{86} \\ a_{71} & a_{72} & 0 & 0 & a_{75} & a_{76} & a_{77} & a_{76} & 0 & 0 & 0 & 0 \\ a_{81} & a_{82} & 0 & 0 & a_{85} & a_{86} & a_{87} & a_{88} & 0 & 0 & 0 & 0 \\ 0 & 0 & b_{31} & b_{32} & 0 & 0 & 0 & 0 & b_{33} & b_{34} & b_{35} & b_{36} \\ 0 & 0 & b_{41} & b_{42} & 0 & 0 & 0 & 0 & b_{43} & b_{44} & b_{45} & b_{46} \\ 0 & 0 & 0 & 0 & b_{57} & b_{58} & 0 & 0 & b_{53} & b_{54} & b_{55} & b_{56} \\ 0 & 0 & 0 & 0 & b_{67} & b_{67} & 0 & 0 & b_{63} & b_{64} & b_{65} & b_{66} \end{bmatrix} \end{array} \begin{matrix} x_1 \\ y_1 \\ x_2 \\ y_2 \\ x_3 \\ y_3 \\ x_4 \\ y_4 \\ x_5 \\ y_5 \\ x_6 \\ y_6 \end{matrix}$$

(3.26)

onde x_i e y_i identificam as coordenadas do nodo i.

A matriz global de rigidez relaciona os valores em cada nodo da malha. Por exemplo para o nodo 3 pode-se identificar que esse está relacionado respectivamente, com os nodos 2, 3, 4 e 6. Logo, na matriz global de rigidez, observa-se que as duas linhas referentes a x_3 e y_3 correspondem ao mapeamento entre esses nodos.

Com a matriz global de rigidez pronta, o próximo passo consiste na montagem do vetor de força $\boldsymbol{f}$, onde serão agrupadas as solicitações aplicadas a todos os nodos da malha. A seguir é apresentada a forma do vetor $\boldsymbol{f}$

$$\boldsymbol{f}^T = \{f_{x1}, f_{y1}, f_{x2}, f_{y2} ... f_{xi}, f_{yi}\} \tag{3.27}$$

onde f_{xi} representa a força aplicada na coordenada x do nodo i, e f_{yi} representa a força aplicada na coordenada y do nodo i.

A partir da Matriz 3.26, do vetor de solicitações externas (Equação 3.27) e do vetor $\boldsymbol{u}$ com os deslocamentos desconhecidos, tem-se um sistema de equações lineares que representa uma aproximação da solução para o domínio da simulação

$$\boldsymbol{K_g u} = \boldsymbol{f} \tag{3.28}$$

Neste estado o sistema não pode ser resolvido pois a matriz global de rigidez $\boldsymbol{K_g}$ não possui uma inversa (FAGAN, 1992). De forma a resolver este problema, torna-se necessário a aplicação das condições de contorno. As condições de contorno definem restrições para a solução do sistema de equações lineares, por exemplo, em problemas relacionados a análise de tensões sobre um corpo a falta destas restrições pode gerar um movimento ao invés de deformações. Dentre os métodos disponíveis destaca-se as condições de contorno de *Dirichlet*, que definem valores estáticos para determinadas forças nodais, assim, impedindo a variação dos valores de deslocamentos nodais (AZEVEDO, 2003).

Ao aplicar a condição de contorno, o sistema de equações lineares (Equação 3.28) pode ser resolvido por algum método de resolução de sistemas lineares. Como exemplo pode-se citar o*Método de Eliminação de Gauss* e a *Factorização LU*(FISH; BELYTSCHKO, 2007).

3.5 Exemplo: *MEF* aplicado a um quadrilátero

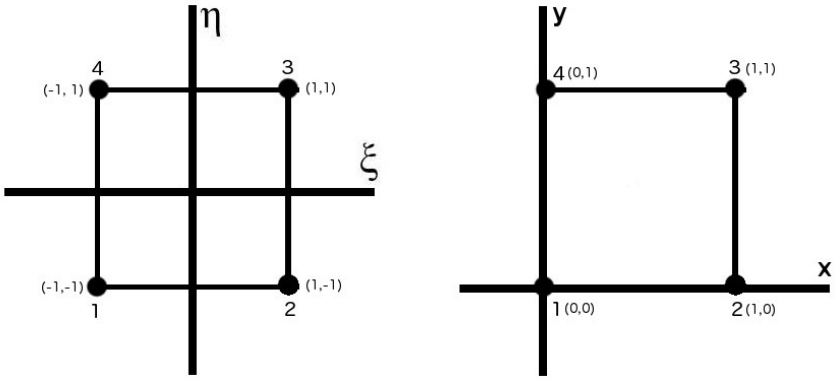

Figura 3.5: Planos de projeção do elemento

Nesta seção será apresentado um exemplo de aplicação do modelo descrito na seção anterior, onde uma força é aplicada a um elemento quadrilátero. É importante salientar que esse processo é idêntico ao aplicado em uma malha composta por hexaedros. Sendo que nesse caso as diferenças referem-se ao número de funções

de forma, as dimensões da matriz de rigidez e consequentemente as dimensões da matriz global de rigidez. Por exemplo, considere a malha da Figura 3.5, onde tem-se a representação de um elemento quadrilátero com as coordenadas naturais e com as coordenadas físicas.

No exemplo da Figura 3.5, cada nodo de um elemento possui 2 graus de liberdade e uma vez que cada elemento possui 4 nodos tem-se no total 8 graus de liberdade no elemento, resultando em uma matriz de rigidez de dimensões [8x8]. Na Figura 3.6 é representada uma deformação, onde a lateral 1-4 está presa e a aresta superior 3-4 recebe uma força $\boldsymbol{f}$.

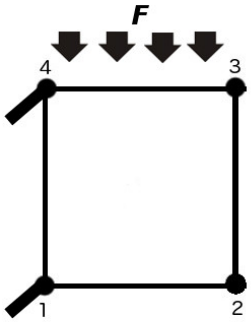

Figura 3.6: Elemento com a aresta 1-4 presa e uma força F sendo aplicada à aresta 3-4.

As funções de forma utilizadas para o elemento são descritas como:

$$\begin{aligned} N_1 &= \frac{1}{4}(1-\xi)(1-\eta) \\ N_2 &= \frac{1}{4}(1+\xi)(1-\eta) \\ N_3 &= \frac{1}{4}(1+\xi)(1+\eta) \\ N_4 &= \frac{1}{4}(1-\xi)(1+\eta) \end{aligned} \tag{3.29}$$

Utilizando um valor para o módulo de Young E de $3 * 10^7$ e um valor para o coeficiente de Poisson v de 0.3, tem-se que a matriz de propriedades $\boldsymbol{D}$ terá a forma da Equação 3.30. Destaca-se que os valores do módulo de Young e do coeficiente de Poisson utilizados no exemplo correspondem a materiais hipotéticos.

$$\boldsymbol{D} = \frac{3 * 10^7}{1-(0.3^2)} \begin{bmatrix} 1 & 0.3 & 0 \\ 0.3 & 1 & 0 \\ 0 & 0 & \frac{1-0.3}{2} \end{bmatrix} = 3.3 * 10^7 \begin{bmatrix} 1 & 0.3 & 0 \\ 0.3 & 1 & 0 \\ 0 & 0 & 0.35 \end{bmatrix} \tag{3.30}$$

De acordo com a Equação 3.11 será necessário resolver a matriz Jacobiana $\boldsymbol{J}$ definida para o elemento. Para tanto serão utilizadas as funções de interpolação definidas na Equação 3.31.

$$\begin{aligned} x &= N_1 x_1 + N_2 x_2 + N_3 x_3 + N_4 x_4 \\ y &= N_1 y_1 + N_2 y_2 + N_3 y_3 + N_4 y_4 \end{aligned} \tag{3.31}$$

Desta forma, tem-se que

$$\boldsymbol{J} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} \end{bmatrix} = \frac{1}{4} \begin{bmatrix} \eta - 1 & 1 - \eta & 1 + \eta & -\eta - 1 \\ \xi - 1 & -\xi - 1 & 1 + \xi & 1 - \xi \end{bmatrix} \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix} \tag{3.32}$$

O determinante e a inversa da matriz Jacobiana $\boldsymbol{J}$ correspondem a

$$|\boldsymbol{J}| = \frac{1}{4} \tag{3.33}$$

$$\boldsymbol{J}^{-1} = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \tag{3.34}$$

Obtida a matriz Jacobiana, pode-se calcular as derivadas parciais das funções N_i em relação às coordenadas físicas (Equação 3.8) para a matriz de rigidez do elemento. Para a integração da matriz de rigidez será utilizado a método da Quadratura de Gauss (FISH; BELYTSCHKO, 2007) sendo que os valores ξ_i, η_i, W_i são dados por

$$\begin{aligned} \xi_1 = -\frac{1}{\sqrt{3}}, \xi_2 = \frac{1}{\sqrt{3}} \\ \eta_1 = -\frac{1}{\sqrt{3}}, \eta_2 = \frac{1}{\sqrt{3}} \\ W_1 = W_2 = 1 \end{aligned} \tag{3.35}$$

De acordo com FAGAN (1992) os valores definidos na Equação 3.35 são geralmente utilizados para a solução das integrais da matriz de rigidez de um elemento quadrilátero.

Utilizando-se os valores da Equação 3.35 na Equação 3.23 tem-se:

$$\boldsymbol{K} = \sum_{i=1}^{2} \sum_{j=1}^{2} W_i W_j |\boldsymbol{J}(\xi_i, \eta_j)| \boldsymbol{B}^T(\xi_i, \eta_j) \boldsymbol{D} \boldsymbol{B}(\xi_i, \eta_j) \tag{3.36}$$

A partir da solução da Equação 3.36 tem-se a matriz de rigidez $\boldsymbol{K}$ do quadrilátero

$$\boldsymbol{K} = 10^7 \begin{bmatrix} 1.48 & 0.54 & -0.91 & -0.04 & -0.74 & -0.54 & 0.16 & 0.04 \\ 0.54 & 1.48 & 0.04 & 0.16 & -0.54 & -0.74 & -0.04 & -0.91 \\ -0.91 & 0.04 & 1.48 & -0.54 & 0.16 & -0.04 & -0.74 & 0.54 \\ -0.04 & 0.16 & -0.54 & 1.48 & 0.04 & -0.91 & 0.54 & -0.74 \\ -0.74 & -0.54 & 0.16 & 0.04 & 1.48 & 0.54 & -0.91 & -0.04 \\ -0.54 & -0.74 & -0.04 & -0.91 & 0.54 & 1.48 & 0.04 & 0.16 \\ 0.16 & -0.04 & -0.74 & 0.54 & -0.91 & 0.04 & 1.48 & -0.54 \\ 0.04 & -0.91 & 0.54 & -0.74 & -0.04 & 0.16 & -0.54 & 1.48 \end{bmatrix} \tag{3.37}$$

Considerando o valor da força F aplicada sobre a aresta 3-4 como $-10N$ (Negativa devido ao sentido da força) pode-se gerar o vetor $\boldsymbol{f}$ com as forças aplicadas ao elemento.

$$\boldsymbol{f}^T = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & -10 & 0 & -10 \end{bmatrix} \tag{3.38}$$

Uma vez que a aresta 4-1 é fixa para este exemplo, serão aplicadas as condições de contorno de *Dirichlet* nestes nodos. Montando o sistema de equações tem-se

$$10^7 \begin{bmatrix} 1.00 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1.00 & 0 & 0 & 0 & 0 & 0 & 0 \\ -0.91 & 0.04 & 1.48 & -0.54 & 0.16 & -0.04 & -0.74 & 0.54 \\ -0.04 & 0.16 & -0.54 & 1.48 & 0.04 & -0.91 & 0.54 & -0.74 \\ -0.74 & -0.54 & 0.16 & 0.04 & 1.48 & 0.54 & -0.91 & -0.04 \\ -0.54 & -0.74 & -0.04 & -0.91 & 0.54 & 1.48 & 0.04 & 0.16 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1.00 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.00 \end{bmatrix} \begin{bmatrix} u_{x1} \\ u_{y1} \\ u_{x2} \\ u_{y2} \\ u_{x3} \\ u_{y3} \\ u_{x4} \\ u_{y4} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ -10 \\ 0 \\ 0 \end{bmatrix} \tag{3.39}$$

onde u_i são as incógnitas que representam os deslocamentos nodais.

Resolvendo o sistema de equações lineares (Equação 3.39) encontra-se os deslocamentos em cada nodo do elemento. Desta forma, a partir da solução da Equação 3.39 tem-se

$$\begin{bmatrix} u_{x1} \\ u_{y1} \\ u_{x2} \\ u_{y2} \\ u_{x3} \\ u_{y3} \\ u_{x4} \\ u_{y4} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -0.0607 \\ -0.1318 \\ 0.0741 \\ -0.1764 \\ 0 \\ 0 \end{bmatrix} \tag{3.40}$$

Esses deslocamentos u_{xi}, u_{yi} serão aplicados nas coordenadas x, y de cada nodo i do elemento para refletir a deformação causada por uma força F aplicada.

No Apêndice A.1 é apresentado uma implementação deste exemplo em MATLAB (MATHWORKS, 2013).

4 IMPLEMENTAÇÃO E TESTES REALIZADOS

Neste capítulo será realizada uma descrição da implementação desenvolvida, bem como serão apresentados alguns resultados obtidos. Para os testes da implementação desenvolvida foram realizadas simulações utilizando objetos formados por diferentes materiais. Mais especificamente, foram realizadas simulações utilizando um cubo de borracha e outro de alumínio. E, posteriormente, foram realizadas simulações utilizando uma barra de alumínio e outra de cobre. Na Seção 4.1 será apresentada a implementação desenvolvida e na Seção 4.2 serão apresentados os resultados obtidos nas simulações que foram realizadas.

4.1 Implementação do Modelo

A implementação desenvolvida é composta basicamente por dois programas independentes. O primeiro deles é responsável pela execução da simulação propriamente dita. Já o segundo é responsável pela visualização dos resultados. A comunicação entre esses programas é realizada utilizando *sockets*, uma vez que a biblioteca utilizada para o desenvolvimento da aplicação de visualização, GLVis, utiliza este tipo de mecanismo para a comunicação dos dados a serem renderizados.

O programa da simulação é responsável basicamente pela leitura da malha, pelo cálculo das matrizes de rigidez de cada elemento, montagem da matriz de rigidez global e pela resolução do sistema de equações lineares. Para o desenvolvimento dessa aplicação utilizou-se a biblioteca MFEM, que provê mecanismos para filtrar os tipos de materiais utilizados, gerar os vetores de força e integrar as matrizes de rigidez dos elementos (MFEM, 2011).

Essa biblioteca implementa, entre outras coisas, funções para leitura da malha que podem ser formadas por hexaedros, quadriláteros, triângulos e tetraedros. Na Seção 4.1.1 é apresentado um exemplo de arquivo de entrada que contém uma malha formada por um único hexaedro.

Além disso, a MFEM apresenta facilidades para a montagem dos sistemas de equações lineares. Uma vez que os sistemas gerados, em geral, são esparsos essa

biblioteca apresenta funções que trabalham com o formato de armazenamento CSR (*Compressed Sparse Row*), que possibilita o armazenamento dos elementos não nulos da matriz em locais contíguos da memória (MARTINOTTO, 2004).

Já para a solução das integrais que definem o elemento finito e que são utilizadas para a montagem da matriz de rigidez do elemento foi utilizado o método da Quadratura de Gauss (FISH; BELYTSCHKO, 2007). Esse método foi utilizado devido ao fato de ser o método padrão para resolver integrais na MFEM (MFEM, 2011) e por já ter sido utilizado com sucesso em outras aplicações (FAGAN, 1992).

A MFEM implementa diversos métodos para a solução dos sistemas de equações gerados. Entre esses métodos estão os métodos de subespaço de Krylov: PCG (*Preconditioned Conjugate Gradient method*), GMRES (*Generalizaed Minimal Residual method*) e BiCGStab (*Biconjudate Gradient Stabilized method*). Para a solução dos sistemas gerados nesse trabalho foi utilizado o método PCG. Optou-se por esse método pois as matrizes de coeficientes são simétricas definida-positivas, e o método do PCG é considerado um dos métodos iterativos mais eficientes para este caso (MARTINOTTO, 2004).

Já para o desenvolvimento da aplicação de visualização foi utilizada a ferramenta GLVis, que permite a visualização de malhas baseadas em Elementos Finitos. Optou-se pela utilização dessa pois seu desenvolvimento foi baseado no projeto do MFEM, sendo assim, essa apresenta funcionalidades implementadas para a comunicação com aplicações que foram desenvolvidas utilizando a biblioteca MFEM (GLVIS, 2010; MFEM, 2011).

4.1.1 Geração da malha

As malhas utilizadas na simulação apresentam formatos simples, sendo que o elemento básico para a composição destas malhas é o hexaedro (Figura 3.2). No Trecho de Código 4.1 tem-se um exemplo do formato do arquivo que descreve uma malha formada por um único hexaedro. A malha é descrita pelas 4 seções que compõem o arquivo. A primeira seção informa a dimensão onde a malha será definida: 1D, 2D, 3D. A segunda seção apresenta os índices de cada vértice do elemento, um atributo relacionado ao tipo do elemento (permitindo trabalhar com materiais diferentes) e o formato do elemento. A terceira seção define as faces que limitam o domínio da malha. E, finalmente, a quarta seção define as coordenadas x, y, z (dependendo da dimensão) do elemento.

Trecho de Código 4.1: Formato do arquivo da malha

```
MFEM mesh v1.0

# POINT       = 0
# SEGMENT     = 1
# TRIANGLE    = 2
```

```
# SQUARE      = 3
# TETRAHEDRON = 4
# CUBE        = 5

dimension
3

# <número de elementos>
# <atributo do elemento> <tipo da geometria> <indice do vértice 1> ... <indice
    do vértice m>
elements
1
2 5 0 1 2 3 4 5 6 7

#<número de elementos da fronteira>
#<atributo do elemento da fronteira> <tipo da geometria> <indice do vértice 1>
    ... <indice do vértice m>
boundary
6
2 3 0 1 2 3
2 3 5 1 2 6
2 3 4 5 6 7
2 3 4 0 3 7
2 3 4 5 1 0
1 3 3 2 6 7

#<número de vértices>
#<dimensão>
#<coordenada 1> ... <coordenada <dimensão>>
vertices
8
3
0 0 0
1 0 0
1 1 0
0 1 0
0 0 1
1 0 1
1 1 1
0 1 1
```

Na Figura 4.1 tem-se a visualização da malha do Trecho de Código 4.1. É importante destacar que após o carregamento da malha da Figura 4.1, foi executado um processo de refinamento uniforme. Esse processo é implementado pela MFEM e divide a região da malha em elementos menores, permitindo a geração de novos elementos dinamicamente.

4.1.2 Característica dos materiais

Na biblioteca MFEM as características do material são introduzidas através do coeficiente de Lamé (BORESI; CHONG; LEE, 2011). Desta forma, para a aplicação do módulo de Young (Seção 2.2) e coeficiente de Poisson (Seção 2.4) nas funções da biblioteca MFEM, foi necessário convertê-los para os coeficientes de Lamé, sendo que a expressão utilizada para a conversão corresponde a

$$\lambda = \frac{Ev}{(1+v)(1-2v)} \tag{4.1}$$

$$G = \frac{E}{2(1+v)} \tag{4.2}$$

onde λ é conhecido como o *Primeiro parâmetro de Lamé* e G como *Segundo parâmetro de Lamé* (também chamado de módulo de cisalhamento), E é o módulo de Young e v o coeficiente de Poisson.

4.2 Resultados das simulações

Nessa seção serão apresentados alguns resultados das simulações realizadas, destacando visualmente as deformações geradas por uma força F aplicada à malha. Nas simulações foram utilizados dois objetos, que são uma barra e um bloco. Para cada um desses objetos foram realizadas simulações utilizando dois diferentes tipos de materiais. Mais especificamente, no caso do bloco foram realizadas simulações com um bloco de borracha e um bloco de alumínio. Já no caso da barra, foram realizadas simulações utilizando uma barra de alumínio e uma de cobre. Os valores do módulo de Young e do coeficiente de Poisson desses materiais encontram-se na Tabela 4.1.

Tabela 4.1: Coeficiente de Young e Poisson para os materiais

Material	Módulo de Young	Coeficiente de Poisson
Borracha	0.1 GPa	0.5
Alumínio	69 GPa	0.33
Cobre	117 GPa	0.36

4.2.1 Resultados: Bloco de Borracha e Bloco de Alumínio

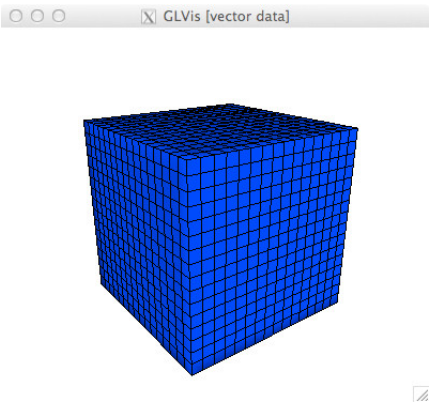

Figura 4.1: Malha do bloco renderizada pelo GLVis

Cada bloco foi dividido em 4096 elementos finitos (Figura 4.1), resultando em um sistema de equações lineares formadas por 14739 equações. Nesses testes foi aplicada uma força de tração F na face frontal do bloco enquanto sua face traseira é mantida presa. Ou seja, uma força F puxa os nodos da face frontal do bloco. Na Figura 4.2 é apresentado o resultado da aplicação dessa força F no bloco de borracha e no bloco de alumínio.

Como pode ser observado na Figura 4.2, no bloco de borracha, a força F é suficiente para deformar todo o seu corpo, enquanto que no bloco de alumínio a força aplicada foi insuficiente para gerar uma deformação aparente.

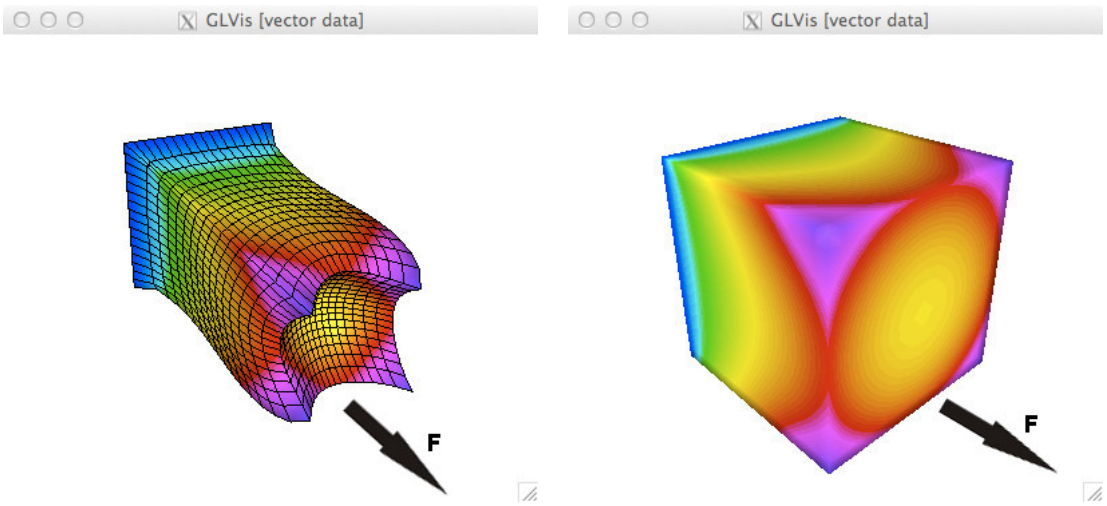

Figura 4.2: Deformação sofrida pelo bloco de borracha e pelo bloco de alumínio, respectivamente

4.2.2 Resultados: Barra de Cobre e Barra de Alumínio

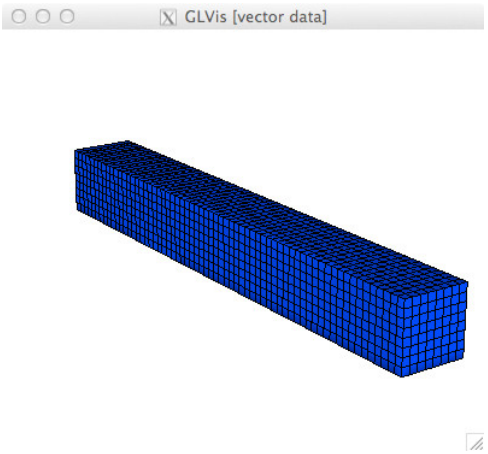

Figura 4.3: Malha da barra renderizada pelo GLVis

Nesses testes aplicou-se uma força F na face superior da barra (Figura 4.3)

pressionando-a para baixo, enquanto sua face traseira é mantida presa. Para esses testes utilizou-se, novamente, 4096 elementos finitos, resultando em um sistema de equações composto por 14739 equações. Na Figura 4.4 tem-se o resultado da aplicação da força F na barra de alumínio e na barra de cobre.

Como pode ser observado na Figura 4.4, na barra de alumínio a força foi o suficiente para torcer o metal. Enquanto que no cobre, esta mesma força F provocou uma deformação menor. Essa diferença deve-se ao fato do cobre apresentar um módulo de Young maior do que o do alumínio.

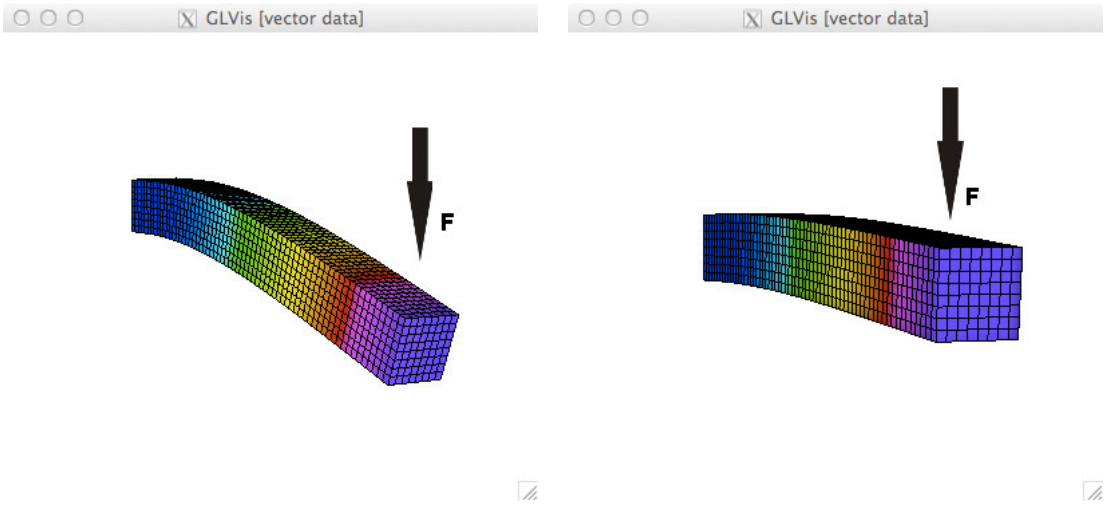

Figura 4.4: Deformação sofrida pela barra de alumínio e pela barra de cobre, respectivamente

4.2.3 Considerações finais

Nesta seção foram apresentados os testes e alguns resultados obtidos com o modelo desenvolvido. Como pode ser observado nos testes, o modelo apresenta resultados coerentes para os diferentes formatos e materiais testados.

Todavia, uma análise mais aprofundada é necessária de forma a verificar se os resultados apresentados pelo modelo encontram-se numericamente corretos. Para isso, torna-se necessário a obtenção da solução analítica das EDPs do problema da elasticidade tridimensional em um caso simples, ou seja, que apresenta uma solução analítica conhecida. E, posteriormente, a comparação da solução analítica com a solução numérica obtida pelo modelo.

5 IMPLEMENTAÇÃO EM GPU

5.1 Introdução

Neste capítulo será apresentada a implementação em *GPU* que foi desenvolvida neste trabalho. Na Seção 5.2 será apresentada uma breve introdução à tecnologia *CUDA*, que foi a tecnologia adotada para desenvolvimento da versão paralela da simulação. Na Seção 5.3 é apresentada uma breve descrição da análise realizada sobre a implementação sequencial. E, por fim, na Seção 5.4 é apresentada uma breve descrição do método do Gradiente Conjugado, juntamente com sua implementação paralela.

5.2 NVIDIA *CUDA*

A tecnologia *CUDA* da NVIDIA foi selecionada para este trabalho devido a sua maturidade e a vasta disponibilidade de documentação. Nesta seção serão tratadas algumas características básicas para a utilização dessa tecnologia. Uma descrição mais completa sobre o *CUDA* pode ser encontrado em SANDERS; KANDROT (2010) e KIRK; W. HWU (2012).

A *GPU* é composta por um conjunto de *Streaming Multiprocessor* (*SM*) responsáveis por controlar a execução das *threads*. Cada *SM* é composto por *CUDA cores* que executam as operações numéricas, sendo que esses *cores* possuem as unidades de processamento inteiro e ponto flutuante. Os *SMs* também possuem unidades de funções especiais (*SFU*) onde são executadas operações mais complexas como funções trigonométricas, raiz quadrada, entre outras (KIRK; W. HWU, 2012).

Para o desenvolvimento de uma aplicação com a tecnologia *CUDA* é necessário a compreensão de alguns conceitos que são utilizados para a distribuição das tarefas entre um vasto conjunto de *threads*. Na *GPU* as *threads* são organizadas em blocos, ou seja, é definida uma quantidade de blocos a serem alocados, e para cada bloco é definida uma quantidade de *threads* a serem usadas. A organização da *GPU* é semelhante a uma matriz, sendo que os blocos podem ser organizados em 1 ou 2 dimensões, e as *threads* de cada bloco podem ser organizadas em 1, 2 ou 3 dimensões

(KIRK; W. HWU, 2012). Na Figura 5.1 é apresentada uma possível configuração de blocos e *threads* na GPU, sendo que *Host* refere-se ao dispositivo (computador) no qual está sendo executada a aplicação, *Device* refere-se à *GPU*, *Kernel* é a denominação das funções que executam na GPU e *Grid* é a disposição lógica dos blocos na *GPU*.

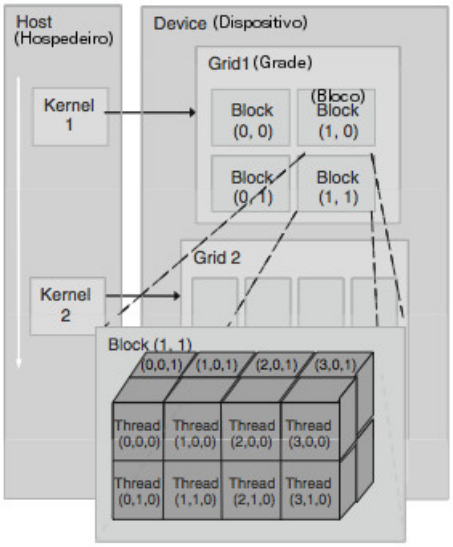

Figura 5.1: Distribuição de blocos e *threads* na GPU. Adaptado da referência (KIRK; W. HWU, 2012)

No exemplo da Figura 5.1 tem-se uma configuração que permitirá a execução da aplicação em 4 blocos dispostos como uma matriz bidimensional com dimensões de 2 x 2 elementos, enquanto que cada bloco terá suas *threads* dispostas em uma matriz tridimensional com dimensões de 2 x 4 x 2 elementos. Além disso, cada bloco e *thread* receberá um identificador que permitirá a seleção dos dados a serem processados por cada *thread*. Por exemplo, a *thread* com identificador $(1, 0, 0)$ poderá ser utilizada para executar operações sobre o valor contido no índice $(1, 0, 0)$ de uma matriz tridimensional.

Outro fator importante a se destacar é a organização de execução das *threads* dentro de um bloco. A *GPU* utiliza como unidade básica de execução os *warps*, onde cada *warp* é composto por 32 *threads* que são executadas em sincronia, ou seja, uma instrução será executada ao mesmo tempo em todas as *threads* que compõem o *warp*. É importante destacar que o número de *threads* que compõe a *warp* pode variar de acordo com a tecnologia da placa gráfica (KIRK; W. HWU, 2012; SANDERS; KANDROT, 2010).

Outra caraterística importante são os níveis de memória existentes na tecnologia *CUDA*. É importante conhecer a hierarquia de memória da *GPU* pois a utilização adequada de cada um dos níveis de memória poderá levar a um aumento de per-

formance significativo (SANDERS; KANDROT, 2010). A Figura 5.2 apresenta os níveis hierárquicos da memória na *GPU*. A sigla *SM* e *SP* da Figura 5.2 correspondem respectivamente aos *Streaming Multiprocessors* e *CUDA cores* da *GPU*. Como exemplo, podemos citar a *GPU* utilizada neste trabalho (Seção 6.2), ela possui 5 *SMs* e cada um possui 192 *CUDA cores*.

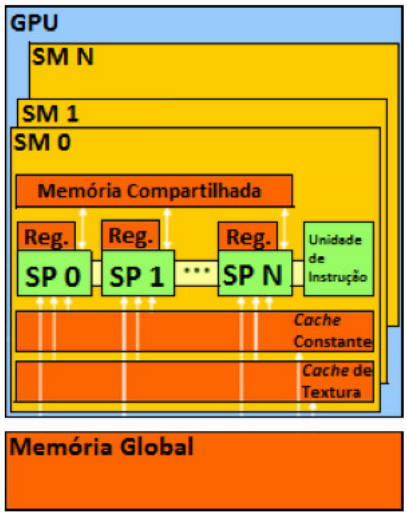

Figura 5.2: Hierarquia de memória da GPU. Adaptado da referência (GRISA, 2010)

A seguir é apresentado um breve resumo de cada nível da memória da *GPU*:

- Registradores: geralmente utilizados para armazenar variáveis locais e dados temporários das *threads*. É o tipo de memória mais rápida da *GPU* (KIRK; W. HWU, 2012).
- Memória Compartilhada: é uma memória localizada em cada *SM* e encontra-se disponível para cada bloco criado no *SM*. A memória compartilhada alocada para um bloco só pode ser acessada pelas *threads* que compõem esse bloco. Seu tamanho depende da arquitetura da *GPU*. Devido a sua baixa latência é, normalmente, utilizada para agilizar o acesso aos dados comuns às *threads* de um mesmo bloco, evitando assim buscas desnecessárias à memória global (KIRK; W. HWU, 2012).
- Memória Global: memória principal da *GPU*, utilizada para a transferência de dados entre *CPU* e *GPU*. É de grande capacidade, porém apresenta alta latência. Essa memória pode ser acessada por qualquer *thread* da *GPU*, permitindo o compartilhamento de dados entre os *kernels* (SANDERS; KANDROT, 2010; GRISA, 2010).

- Memória Constante: embora localizada na memória principal, as *threads* podem acessar essa memória com baixa latência (cada *SM* possui um *cache* para acesso a esta memória). Porém, é uma memória que, normalmente, possui baixa capacidade de armazenamento (SANDERS; KANDROT, 2010).

- Memória de textura: também localizada na memória principal, e cada *SM* possui um *cache* para acesso a essa memória. Normalmente é utilizada na renderização dos gráficos, mas também pode ser utilizada para o processamento em *CUDA* (GRISA, 2010; KIRK; W. HWU, 2012).

5.3 Perfilamento da implementação sequencial

Para a identificação dos trechos de código que apresentam maior custo computacional na implementação sequencial foi utilizada a ferramenta *gprof*, que permite o perfilamento de uma aplicação (FENLASON, 1993). A partir do uso do *gprof* foi possível identificar algumas características do programa de simulação como, por exemplo, o tempo que cada uma das funções permanece em execução e o percentual que esse tempo representa no tempo total de execução da simulação.

Para a realização do perfilamento foi considerada a variação do número de elementos na malha da simulação. No primeiro caso foi utilizada uma malha com 4.096 elementos, no segundo caso 32.768 elementos e no terceiro com 262.144 elementos. Nos Trechos de Código 5.1, 5.2 e 5.3 encontram-se os resultados gerados pelo *gprof*, sendo que, as linhas marcadas apresentam os dados relevantes para este trabalho. O tempo médio de execução na implementação sequencial (em cada caso) foi calculado utilizando-se 10 execuções.

No primeiro caso (Trecho de Código 5.1) o tempo médio de execução na *CPU* foi de 0,74 segundos onde o método do Gradiente Conjugado (GC) ocupou 63% desse tempo, sendo que 56,2% do tempo total de execução da simulação foram gastos na operação matriz-vetor.

Trecho de Código 5.1: Saída do *gprof* para perfilamento com 4.096 elementos

```
index % time    self  children    called     name
                                                 <spontaneous>
[1] 63.0 0.00 0.46 CG(Operator const&, Vector const&, Vector&, int, int, double, double) [1]
                0.41    0.00     462/462          SparseMatrix::Mult(Vector const&, Vector&) const [2]
                0.03    0.00    1382/1382         add(Vector const&, double, Vector const&, Vector&) [8]
                0.02    0.00     923/923          Vector::operator*(Vector const&) const [11]
                0.00    0.00       1/1            subtract(Vector const&, Vector const&, Vector&) [162]
                0.00    0.00       1/32774        Vector::operator=(Vector const&) [38]
-----------------------------------------------
                0.41    0.00     462/462          CG(Operator const&, Vector const&, Vector&, int, int,
                     double, double) [1]
[2] 56.2 0.41 0.00 462 SparseMatrix::Mult(Vector const&, Vector&) const [2]
                0.00    0.00     462/2639         Vector::operator=(double) [85]
-----------------------------------------------
                                                 <spontaneous>
[3]     32.8    0.00    0.24                 BilinearForm::Assemble(int) [3]
                0.10    0.09    4096/4096         ElasticityIntegrator::AssembleElementMatrix(FiniteElement
                     const&, ElementTransformation&, DenseMatrix&) [4]
```

```
                0.05    0.00    4096/4096        SparseMatrix::AddSubMatrix(Array<int> const&, Array<int>
     const&, DenseMatrix const&, int) [6]
                0.00    0.00    4096/18688       FiniteElementSpace::GetElementDofs(int, Array<int>&) const
      [52]
                0.00    0.00    4096/8192        FiniteElementSpace::GetElementVDofs(int, Array<int>&)
     const [71]
                0.00    0.00    4096/8192        FiniteElementSpace::GetFE(int) const [72]
                0.00    0.00    4096/8192        Mesh::GetElementTransformation(int) [68]
                0.00    0.00       1/1           SparseMatrix::SparseMatrix(int, int) [167]
```

No segundo caso (Trecho de Código 5.2) o tempo médio de execução na *CPU* foi de 11 segundos, sendo que o método do GC ocupou 80,2% do tempo de *CPU*. Já a operação matriz-vetor consumiu 76,3% do tempo total de *CPU*.

Trecho de Código 5.2: Saída do *gprof* para perfilamento com 32.768 elementos

```
index % time    self  children    called     name
                                                 <spontaneous>
[1] 80.2 0.00 8.37 CG(Operator const&, Vector const&, Vector&, int, int, double, double) [1]
                7.96    0.00     898/898         SparseMatrix::Mult(Vector const&, Vector&) const [2]
                0.27    0.00    2690/2690        add(Vector const&, double, Vector const&, Vector&) [8]
                0.14    0.00    1795/1795        Vector::operator*(Vector const&) const [11]
                0.00    0.00       1/1           subtract(Vector const&, Vector const&, Vector&) [171]
                0.00    0.00       1/262150      Vector::operator=(Vector const&) [60]
-----------------------------------------------
                7.96    0.00     898/898         CG(Operator const&, Vector const&, Vector&, int, int,
     double, double) [1]
[2] 76.3 7.96 0.00 898 SparseMatrix::Mult(Vector const&, Vector&) const [2]
                0.00    0.00     898/9603        Vector::operator=(double) [28]
-----------------------------------------------
                                                 <spontaneous>
[3]     16.2    0.00    1.69                 BilinearForm::Assemble(int) [3]
                0.73    0.68   32768/32768       ElasticityIntegrator::AssembleElementMatrix(FiniteElement
     const&, ElementTransformation&, DenseMatrix&) [4]
                0.28    0.00   32768/32768       SparseMatrix::AddSubMatrix(Array<int> const&, Array<int>
     const&, DenseMatrix const&, int) [5]
                0.00    0.00   32768/140288      FiniteElementSpace::GetElementDofs(int, Array<int>&) const
      [70]
                0.00    0.00   32768/65536       FiniteElementSpace::GetElementVDofs(int, Array<int>&)
     const [83]
                0.00    0.00   32768/65536       FiniteElementSpace::GetFE(int) const [84]
                0.00    0.00   32768/65536       Mesh::GetElementTransformation(int) [80]
                0.00    0.00       1/1           SparseMatrix::SparseMatrix(int, int) [175]
```

Por fim, no terceiro caso (Trecho de Código 5.3) o tempo de execução na *CPU* foi de 247 segundos, onde o método do GC ocupou 92,9% do tempo total de *CPU*. Já a operação matriz-vetor consumiu 89,9% do tempo total de *CPU*.

Trecho de Código 5.3: Saída do *gprof* para perfilamento com 262.144 elementos

```
[1] 92.9 0.00 228.29 CG(Operator const&, Vector const&, Vector&, int, int, double, double) [1]
              220.81    0.05    1754/1754        SparseMatrix::Mult(Vector const&, Vector&) const [2]
                4.94    0.00    5258/5258        add(Vector const&, double, Vector const&, Vector&) [5]
                2.49    0.00    3507/3507        Vector::operator*(Vector const&) const [8]
                0.00    0.00       1/2097158     Vector::operator=(Vector const&) [57]
                0.00    0.00       1/1           subtract(Vector const&, Vector const&, Vector&) [187]
-----------------------------------------------
              220.81    0.05    1754/1754        CG(Operator const&, Vector const&, Vector&, int, int,
         double, double) [1]
[2] 89.9 220.81 0.05 1754 SparseMatrix::Mult(Vector const&, Vector&) const [2]
                0.05    0.00    1754/36571       Vector::operator=(double) [12]
-----------------------------------------------
                                                 <spontaneous>
[3]      5.6    0.01   13.70                 BilinearForm::Assemble(int) [3]
                5.39    5.54  262144/262144      ElasticityIntegrator::AssembleElementMatrix(FiniteElement
     const&, ElementTransformation&, DenseMatrix&) [4]
                2.73    0.00  262144/262144      SparseMatrix::AddSubMatrix(Array<int> const&, Array<int>
     const&, DenseMatrix const&, int) [6]
```

```
0.00    0.01  262144/524288      Mesh::GetElementTransformation(int) [44]
0.00    0.00  262144/1085440     FiniteElementSpace::GetElementDofs(int, Array<int>&) const
      [37]
0.00    0.01  262144/524288      FiniteElementSpace::GetElementVDofs(int, Array<int>&)
     const [53]
0.01    0.00  262144/524288      FiniteElementSpace::GetFE(int) const [61]
```

Os resultados do *gprof* demonstraram que o método do GC representa o maior percentual de tempo de execução da simulação, e grande parte desse tempo deve-se à operação de matriz-vetor. Assim, este trabalho será focado na paralelização do GC, mais especificamente, será realizada a paralelização das operações básicas de álgebra linear que são executadas pelo método do GC que são: multiplicação matriz-vetor, produto escalar e soma de vetores.

5.4 Gradiente Conjugado

O método do GC é um método iterativo para a solução de sistemas de equações lineares cuja matriz é simétrica definida positiva (SDP). Esse método baseia-se na minimização da função quadrática. Ou seja, quando as equações de um sistema $\mathbf{Ax} = \mathbf{b}$ são abertas, verifica-se que $F(x_i)$ é uma função quadrática em relação aos componentes x_i do vetor $\mathbf{x}$. A representação espacial da função $F(x_i)$ é uma parabolóide, sendo assim, essa possui apenas um único mínimo (GRISA, 2010). A Figurai 5.3 ilustra o gráfico de uma função quadrática.

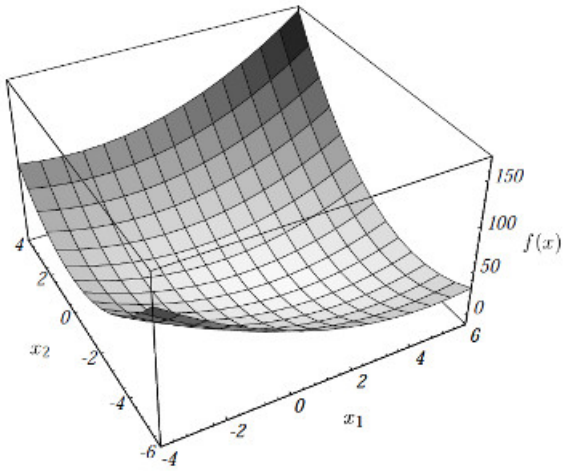

Figura 5.3: Exemplo de função quadrática. Adaptado da referência (GRISA, 2010)

O mínimo da função $F(x_i)$ é atingido quando $\mathbf{Ax} - \mathbf{b} = 0$, ou seja, quando o vetor $\mathbf{x}$ convergir para a solução do sistema. Como o vetor gradiente de $F(x_i)$ aponta para a direção de crescimento máximo da função, é natural que na busca

do mínimo seja utilizado o negativo do gradiente de $F(x_i)$ (CRISTINA; CUNHA, 2000; GRISA, 2010).

Desta forma, o método calcula cada aproximação da solução por base do resíduo ($\mathbf{b}-\mathbf{Ax}$) da iteração anterior, sempre caminhando na direção contrária do gradiente de $F(x_i)$. Geralmente, quando não se tem uma boa estimativa da solução $\mathbf{x}$, define-se como a aproximação inicial $\mathbf{x} = 0$, e normalmente são utilizados dois critérios de parada que são: um número máximo de iterações e o valor de tolerância, que é comparada com o resíduo gerado na iteração anterior.

No Algoritmo 1 tem-se a versão do método do GC utilizado pela biblioteca MFEM.

Algoritmo 1: Gradiente Conjugado utilizado na MFEM

Dados A, **x**, **b**, maxiter, atol, rtol
r = b - **A** × **x**
d = **r**
nom = **r** · **r**
r0 = nom * rtol
Se r0 < atol **Então**
    r0 = atol
Se nom < r0 **Então**
    Retorna o resultado em **x**
Ad = **A** × **d**
den = **d** · **Ad**
Se den <= 0 **E** nom > 0 **Então**
    A não é Positivo Definido
k = 0
Para k < maxiter **faça**
    alpha = nom / den
    x = **x** + alpha * **d**
    r = **r** - alpha * **Ad**
    betanom = **r** · **r**
    Se betanom < r0 **Então**
        Retorna o resultado em **x**
    beta = betanom / nom
    d = **r** + beta * **d**
    Ad = **A** × **d**
    den = **d** · **Ad**
    Se den <= 0 **E** **d** · **d** > 0 **Então**
        A não é Positivo Definido
    nom = betanom

Devido ao método do GC ser composto por operações entre vetores e matrizes, torna-se adequado a paralelização dessas operações na arquitetura da *GPU* (GRISA, 2010). Conforme mencionado na Seção 5.3 as operações internas do GC que foram paralelizadas são a soma de vetores, produto escalar e multiplicação matriz-vetor.

A operação de matriz-vetor foi paralelizada pois apresentou o maior custo computacional da simulação. A soma de vetores e o produto escalar, embora não tenham apresentado um custo computacional muito elevado, foram paralelizadas de forma a permitir que a maior parte dos cálculos fosse executado na *GPU*. Desta forma, evitou-se o acúmulo de transferências de dados entre a memória da *CPU* e da *GPU*, o que poderia ocasionar uma perda de performance considerável, pois a transferência de dados entre a memória da *GPU* e da *CPU* é uma operação que gera grande impacto no tempo de execução da simulação (SANDERS; KANDROT, 2010).

No Apêndice A.2 pode ser encontrado o código completo da versão do método do GC para *GPU* que foi desenvolvido neste trabalho. E o código fonte do GC sequencial, implementado pelos criadores do MFEM (MFEM, 2011), está disponível no Apêndice A.3. No Apêndice A.5 é apresentado, graficamente, a lógica de execução das operações de multiplicação matriz-vetor, produto escalar e soma de vetores implementadas em *GPU*.

5.4.1 Paralelização da operação de multiplicação matriz-vetor

A operação de multiplicação matriz-vetor é a que apresenta o maior custo computacional no método do GC. No Trecho de Código 5.4 é apresentada a implementação da operação de multiplicação matriz-vetor para a arquitetura de uma *GPU*.

Trecho de Código 5.4: Operação de multiplicação matriz-vetor em CUDA

```
1 #define THREAD_PER_BLOCK 128
2 #define WARP_SIZE 32
3
4 __global__ void _matrixByVector(const double *A, const int *I, const int *J,
5                                  const double *x, double *y, int size) {
6
7
8     __shared__ double sums[THREAD_PER_BLOCK];
9
10    double tempY = 0;
11    int i = 0;
12    int rowIndex = (THREAD_PER_BLOCK/WARP_SIZE * blockIdx.x) + (threadIdx.x /
          WARP_SIZE);
13
14    if (rowIndex >= size) return;
15
16    int rowOffset = I[rowIndex];
17    int rowWidth = I[rowIndex + 1];
18    int nIdx = threadIdx.x % WARP_SIZE;
19
20    for (i = rowOffset + nIdx; i < rowWidth; i += WARP_SIZE ){
21        tempY += A[i] * x[J[i]];
```

```
22      }
23
24      sums[threadIdx.x] = tempY;
25
26      if ( (threadIdx.x  /  WARP_SIZE) % 2 == 0) {
27          for (i = 16; i > 0; i >>= 1) {
28              sums[threadIdx.x] += sums[threadIdx.x + i];
29          }
30      }
31
32      if (threadIdx.x % WARP_SIZE == 0) {
33          y[rowIndex] = sums[threadIdx.x];
34      }
35
36   }
37 }
```

Na implementação desenvolvida cada bloco é responsável por tratar uma ou mais linhas da matriz **A**, sendo que a divisão de linhas é feita por *warps*, ou seja, cada *warp* dentro de um bloco será responsável por realizar o produto escalar entre uma linha da matriz **A** e o vetor **x**. Na Linha 12 é calculado o índice da linha da matriz a ser tratada por cada *thread*. A primeira parte da equação da Linha 12, `(THREAD_PER_BLOCK/WARP_SIZE * blockIdx.x)` é responsável por selecionar o intervalo de linhas que cada bloco irá operar, já a segunda parte `(threadIdx.x / WARP_SIZE)` por selecionar as *threads* de cada *warp* que irão trabalhar em uma mesma linha. Por exemplo, considerando 128 *threads* no bloco 1 tem-se que a *thread* de *id* 20 irá operar na linha 4 da matriz.

Na linha 20 até 22 é realizado o produto escalar entre cada linha da matriz **A** e o vetor **x**, sendo que cada *thread* do bloco acessa os valores da matriz em intervalos de 32 valores, ou seja, do tamanho de um *warp*. Dessa forma, todas as *threads* que compõem o *warp* estarão acessando dados contínuos na memória global. Esse padrão de acesso a memória é detectado pela *GPU* sendo realizado como uma única requisição (KIRK; W. HWU, 2012). Caso o acesso a memória fosse aleatório, seria necessário realizar uma requisição por *thread*, afetando a performance devido ao sincronismo existente na execução das *threads* de um *warp*.

Na linha 24, o resultado de cada *thread* é armazenado em um vetor na memória compartilhada. A alocação da memória compartilhada é restrita por bloco, ou seja, somente as *threads* do bloco tem acesso à área alocada (SANDERS; KANDROT, 2010).

Uma vez que cada *thread* tenha terminado de calcular o valor do produto escalar, é realizada a operação de *reduction* [1] (Linhas 23 a 27), onde as *threads* do bloco são divididas em grupos de 32 *threads* e a operação de *reduction* é realizada sobre cada

[1] Basicamente a *reduction* é uma operação que mapeia uma entrada de n elementos para um único escalar. Neste trabalho a operação de *reduction* efetua a soma dos valores de entrada, ou seja, dada uma entrada de n valores o resultado do *reduction* será a soma de todos os n valores.

grupo. Por fim, o valor do produto escalar é atribuído ao vetor y (o vetor resultante da operação matriz-vetor), na posição referenciada pela *rowIndex* de cada *warp*. O vetor y está armazenado na memória global da GPU.

5.4.2 Paralelização da operação de produto escalar

A operação de produto escalar não apresentou um custo computacional elevado no método do GC, porém sua implementação em GPU evita a troca excessiva de dados entre a memória da CPU e da GPU, se essa operação não fosse implementada na GPU seria necessário enviar os dados para a CPU efetuar a operação e depois retornar o resultado a GPU. Desta forma, ter-se-ia uma perda considerável de desempenho. A implementação para a operação de produto escalar na GPU é apresentada no Trecho de Código 5.5.

Trecho de Código 5.5: Operação de produto escalar em CUDA

```
1 #define THREAD_PROD 512
2 __global__ void _dotProduct(const double *v1, const double *v2, double *r, int
      size) {
3        __shared__ double sums[THREAD_PROD];
4
5         double tempY = 0;
6         int i = 0;
7         int rowOffset = 0;
8         int rowWidth = size;
9
10        for (i = rowOffset + threadIdx.x; i < rowWidth; i += THREAD_PROD ){
11            tempY += v1[i] * v2[i];
12        }
13
14        sums[threadIdx.x] = tempY;
15        __syncthreads();
16
17
18        if (threadIdx.x < THREAD_PROD/2) {
19            for (i = THREAD_PROD/2; i > 0; i >>= 1) {
20
21                sums[threadIdx.x] += sums[threadIdx.x + i];
22            }
23        }
24
25        if (threadIdx.x == 0) {
26            (*r) = sums[0];
27        }
28 }
```

Esta é uma implementação simplificada pois sua execução se baseia na utilização de um único bloco com 512 ou 1024 *threads* (De acordo com a capacidade do dispositivo). Cada *thread* é responsável por executar uma etapa do produto escalar (Linhas 10 a 12), sendo que após a conclusão dos cálculos os resultados parciais são armazenados na memória compartilhada e as *threads* são sincronizadas (Linhas 14 e 15).

Uma vez que os valores parciais estejam armazenados é necessário uma operação de *reduction* (Linhas 18 a 23) para efetuar a soma de todos os resultados. Por fim, a *thread* 0 é responsável por armazenar o resultado final no endereço referenciado pela variável r (Linhas 25 a 27), ou seja, esse endereço apresenta o resultado final do produto escalar.

5.4.3 Paralelização da operação de soma de vetores

A implementação da operação de soma entre vetores em *CUDA* é realizada através da divisão da soma dos elementos dos vetores entre as *threads* disponíveis em cada bloco, ou seja, cada *thread* será responsável por calcular a soma entre dois elementos de mesmo índice e armazenar o resultado no vetor resultante. A implementação desenvolvida neste trabalho está no Trecho de Código 5.6

Trecho de Código 5.6: Operação de soma de vetores em CUDA

```
1 __global__ void _addVector(const double* v1, const double alpha, const double*
      v2, double *r, int size) {
2     int idx = blockDim.x * blockIdx.x + threadIdx.x;
3     double a,b;
4     if (idx < size) {
5         a = v1[idx];
6         b = v2[idx];
7         r[idx] = a + alpha * b;
8     }
9 }
```

Na Linha 2 é realizada a identificação do índice dos elementos de $v1$ e $v2$ a serem somados, e a posição onde o resultado será armazenado no vetor r. Nas Linhas 4 a 8, é verificado se o índice não ultrapassou os limites dos vetores, ou seja, impede que alguma *thread* que possua um índice *idx* além do tamanho do vetor execute a soma. E caso não tenha ultrapassado esse limite, é realizada a soma entre os elementos dos vetores. Essa implementação já considera a possibilidade de multiplicação por um escalar *alpha*, pois normalmente as somas de vetores no método do GC incluem a multiplicação de um dos vetores por um escalar. Além disso, essa implementação fornece um meio conveniente de realizar a subtração de vetores, para tanto basta utilizar um valor de *alpha* igual a -1.

5.5 Verificação da Implementação em GPU

Para verificar a acurácia dos resultados obtidos na *GPU* em relação aos resultados gerados pela *CPU*, foi calculado o erro quadrático médio entre os resultados gerados pela implementação sequencial e os resultados gerados pela implementação em *GPU*. Os testes foram realizados sobre as três malhas utilizadas no perfilamento (Seção 5.3), os resultados são apresentados na Tabela 5.1.

Tabela 5.1: Resultado do erro quadrático médio

ID	Erro quadrático médio
Malha 1	6,19 x 10^{-16}
Malha 2	2,86 x 10^{-16}
Malha 3	1,03 x 10^{-16}

Essa diferença de acurácia pode ocorrer devido a ordem em que as operações de ponto flutuante foram executadas na *GPU* (WHITEHEAD; FIT-FLOREA, 2011). Por exemplo, considerando a propriedade associativa da soma, dependendo da ordem em que ela for realizada na *GPU* o resultado final pode divergir do resultado gerado pela *CPU*.

6 ANÁLISE DE DESEMPENHO

Neste capítulo serão apresentados os resultados da análise de desempenho entre a implementação sequencial e a implementação em *GPU*. Na Seção 6.1 serão apresentadas as características das malhas utilizadas nos testes. Na Seção 6.2 serão descritas as características do *hardware* que foi utilizado para os testes. E, por fim, na Seção 6.3 serão apresentados os resultados dos testes para cada uma das malhas utilizadas.

6.1 Malha para teste

Para a realização do teste de desempenho será utilizada uma barra cuja a descrição da malha é apresentada no Apêndice A.4. Esse objeto será utilizado em três testes, variando-se somente o número de elementos da malha. A quantidade de elementos foi definida com base na função de refinamento do MFEM. Essa função permite aumentar o número de elementos que compõe a malha sem alterar seu tamanho e forma. O número de elementos das malhas que serão utilizadas nos testes correspondem a 4.096 (Malha 1), 32.768 (Malha 2) e 262.144 (Malha 3) elementos. A Figura 6.1 mostra a renderização dessa malha com 4.096 elementos.

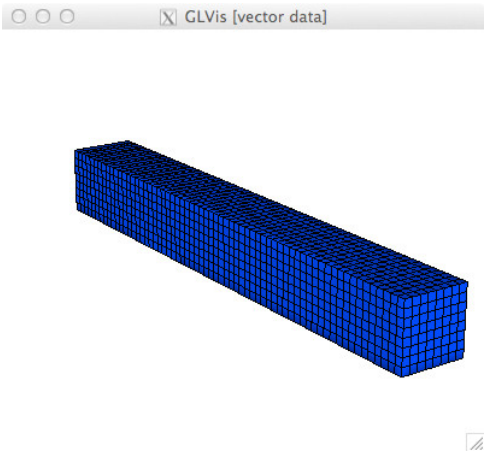

Figura 6.1: Malha utilizada para os testes

A partir dessa malha, será gerada a matriz global de rigidez que corresponde ao sistema de equações lineares a ser resolvido pelo Gradiente Conjugado em cada simulação. Na Tabela 6.1 é apresentado um resumo das informações das matrizes geradas.

Tabela 6.1: Matrizes utilizadas na análise de desempenho

ID	Elementos da malha	Dimensão da matriz	Valores não-nulos
Malha 1	4.096	15.795	1.040.124
Malha 2	32.768	111.843	8.141.590
Malha 3	262.144	839.619	64.415.367

6.2 Recursos de hardware

Para a realização dos testes foi utilizado um computador com o processador Intel Core i5 3570k tendo um *clock speed* de 3.4 GHz, 4 *cores*, *cache* L1 de 32 KB por *core* e *cache* L2 de 256 KB por *core*. O computador possui ainda 8 GB de RAM do tipo DDR3. O sistema operacional utilizado foi o Ubuntu 12.04 64-bit com o *Kernel* 3.5.0-23. Já a placa gráfica utilizada foi uma GeForce GTX 660 com 2 GB de memória, *clock speed* de 1.06 GHz e 960 núcleos *CUDA*. Na Tabela 6.2 são apresentadas informações adicionais sobre essa placa gráfica.

Tabela 6.2: Informações da *GPU*

GPU - GeForce GTX 660	
Capacidade *CUDA*	3.0
Clock Speed	1.06 GHz
Quantidade de memória global	2 GB
Quantidade de memória compartilhada por bloco	48 KB
Quantidade de registradores disponíveis por bloco	65.536
Número de *Streaming Multiprocessors* (*SM*)	5
Número de *CUDA cores* por *SM*	192
Tamanho do *warp*	32 *threads*
Número máximo de *threads* por *SM*	2048
Número máximo de blocos por *SM*	16
Número máximo de *threads* por bloco	1024
Dimensão máxima de um bloco	(1024, 1024, 64)

6.3 Avaliação de desempenho

Para a avaliação do desempenho entre as duas implementações será considerado o tempo total de execução da simulação, ou seja, o tempo total de execução da

simulação inclui as etapas de geração da malha, montagem da matriz global de rigidez e solução do sistema de equações através do método do GC. O cálculo da média e do desvio padrão foi realizado a partir de resultados obtidos através de 50 execuções, ou seja, para cada uma das malhas foram executadas 50 simulações. Os dados foram computados utilizando variáveis do tipo *double*, devido a implementação da biblioteca MFEM utilizar esse tipo de dado para armazenamento das matrizes e vetores. Para a convergência do método do GC considera-se uma tolerância de 10^{-8}. A Tabela 6.3 apresenta os resultados da execução da implementação sequencial e da versão em *GPU*.

Tabela 6.3: Avaliação da implementação sequencial e paralela em *GPU*

ID	Iterações	Tempo médio (seg)		Desvio Padrão		Redução de tempo*
		CPU	GPU	CPU	GPU	
Malha 1	461	0,76	0,66	0,01	0,01	13%
Malha 2	897	10,81	6,87	0,01	0,01	36%
Malha 3	1753	244,04	88,00	1,27	0,06	64%

* Redução de tempo na execução da implementação em *GPU* sobre a implementação sequencial

Nos testes realizados a implementação em *GPU* apresentou um ganho de desempenho quando comparada à implementação sequencial. Porém, é observado que nos dois primeiros casos (Malha 1 e Malha 2) a diferença apresentada foi pequena, sendo que o real ganho de desempenho com a *GPU* só foi alcançado quando o sistema de equações tornou-se maior (Malha 3).

De acordo com CHEN et al. (2012) e MOORKAMP et al. (2010) o menor desempenho da implementação em *GPU* para sistemas de equações menores deve-se principalmente ao custo computacional envolvido na alocação e transferência de dados entre a *GPU* e a *CPU*. Ou seja, com uma quantidade inferior de dados, o ganho obtido com o uso da *GPU*, acaba não compensando o tempo consumido para a alocação e transferência de dados entre a *CPU* e a *GPU*. De fato, para sistemas menores, a transferência de dados entre a *CPU* e a *GPU* provocam uma redução significativa no percentual de ocupação dos processadores da *GPU*. Resultados similares podem ser encontrados no trabalho de RAHMADDIANSYAH; RASHID (2011) e no *benchmark* da *API ArrayFire* em ACCELEREYES (2013).

Para verificar a influência da alocação e transferência de dados entre a *CPU* e *GPU* foram realizados dois testes de desempenho entre a implementação sequencial e em *GPU* do método do GC. No primeiro teste, foram comparados os tempos de execução da implementação sequencial e da implementação em *GPU*, considerando-se o tempo de alocação e transferência de dados na implementação em *GPU*. Já no segundo teste o tempo de alocação e transferencia de dados não foi considerado, ou

seja, será considerado somente o tempo de processamento das operações na *GPU*. Na Tabela 6.4 são apresentados os resultados do primeiro teste, e na Tabela 6.5 os resultados do segundo teste.

Tabela 6.4: Avaliação de desempenho das implementações do método do GC, considerando-se o tempo de alocação e transferência de dados na implementação em *GPU*

ID	Iterações	Tempo médio (seg)		Desvio Padrão		Redução de tempo*
		CPU	GPU	CPU	GPU	
Malha 1	461	0,45	0,38	0,01	0,01	15%
Malha 2	897	8,42	4.85	0,02	0,02	42%
Malha 3	1753	228,86	71.70	1,20	0,05	68%

* Redução de tempo da implementação em *GPU* sobre a implementação sequencial

Tabela 6.5: Avaliação de desempenho das implementações do método do GC, desconsiderando-se o tempo de alocação e transferência de dados na implementação em *GPU*

ID	Iterações	Tempo médio (seg)		Desvio Padrão		Redução de tempo*
		CPU	GPU	CPU	GPU	
Malha 1	461	0,45	0,21	0,01	0,01	53%
Malha 2	897	8,42	3,86	0,01	0,01	54%
Malha 3	1753	228,86	64,57	1,20	0,05	71%

* Redução de tempo na execução da implementação em *GPU* sobre a implementação sequencial

Conforme pode ser pode ser observado nas Tabelas 6.4 e 6.5, para malhas com um número de elementos menor e, consequentemente, sistemas de equações menores (como a Malha 1), a influência do tempo de alocação e transferência de dados é alta. De fato, para a Malha 1, a simples remoção das operações de alocação e de transferência dos dados, provoca um aumento de desempenho de 15% para 53%. Já com um aumento no número de elementos da malha, tem-se uma influência menor do tempo de alocação e transferência de dados, uma vez que esse tempo será diluído entre o tempo necessário para a execução das operações de álgebra linear que compõem o método do GC.

Desta forma, pode-se concluir que para malhas com poucos elementos, o uso somente da *CPU* seria mais adequado, uma vez que o percentual de tempo necessário para a alocação e para a transferência de dados entre a *CPU* e *GPU* provoca uma queda significativa na eficiência de uso dos recursos da *GPU*. Já com o aumento no número de elementos da malha, a utilização da *GPU* torna-se mais adequada, uma vez que neste caso tem-se uma aumento significativo no percentual de ocupação dos processadores da *GPU*.

7 CONCLUSÃO

O objetivo deste trabalho foi desenvolver um modelo matemático da elasticidade tridimensional e implementa-lo utilizando computação híbrida. Para o desenvolvimento desse modelo foi utilizado o método dos Elementos Finitos e para a implementação do mesmo foram utilizadas as bibliotecas *MFEM* (MFEM, 2011), *GLVis* (GLVIS, 2010) e a tecnologia *CUDA* da NVIDIA (NVIDIA, 2013a). Optou-se pela utilização do *MFEM* pois essa biblioteca apresenta um conjunto de funções que implementam operações comuns em aplicações do *MEF* (*Método dos Elementos Finitos*), desta forma, agilizando e simplificando o desenvolvimento desse tipo de aplicação.

Com base no perfilamento da implementação sequencial observou-se que grande parte do tempo de execução da simulação deve-se à execução do método do GC. Desta forma, a implementação em *GPU* concentrou-se na paralelização das operações que compõem o método do GC, ou seja, as operações de soma de vetores, produto escalar e multiplicação matriz-vetor. Um estudo sobre a paralelização da operação de montagem da matriz global de rigidez não foi realizado devido às restrições de tempo. Desta forma, considera-se a análise e implementação da operação de montagem da matriz global de rigidez como uma possível extensão deste trabalho.

Durante o processo de desenvolvimento observou-se que a simulação da elasticidade tridimensional, principalmente o método do GC, baseia-se na execução de uma elevada quantidade de operações matemáticas por unidade de tempo, normalmente sendo as mesmas operações executadas iterativamente sobre conjuntos diferentes de dados. Esta característica favoreceu a utilização da arquitetura da GPU, pois a mesma trabalha com blocos de processadores (*CUDA cores*) que são divididos em *warps* que executam uma mesma operação sobre um conjunto diferente de dados (KIRK; W. HWU, 2012).

Além disso, observou-se que o uso da *GPU* não é adequado para problemas onde a quantidade de dados a serem processados é baixa. De fato, testes realizados mostraram que o tempo necessário para a alocação e transferência de dados entre a memória da *CPU* e da *GPU* acaba provocando uma queda significativa na eficiência

de uso dos processadores da GPU.

Por fim, considerando-se os objetivos deste trabalho e os resultados na avaliação de desempenho, pode-se concluir que a *GPU* apresentou um ganho de desempenho satisfatório na simulação implementada. E a tecnologia *CUDA* mostrou-se madura e de fácil adoção para aplicativos que possam tirar proveito da computação híbrida.

7.1 Trabalhos Futuros

Este trabalho cria algumas opções para trabalhos futuros. Entre eles, podemos citar:

- Análise e implementação da operação de montagem da matriz global de rigidez em *GPU*.

- Uma nova implementação da simulação utilizando a *GPU* e bibliotecas como *Message Passing Interface* (MPI) ou *Open Multi-Processing* (OpenMP). Ou uma nova comparação de desempenho com a versão paralela do MFEM, que utiliza as bibliotecas OpenMP e MPI.

- Utilização de outras plataformas de desenvolvimento com *GPU*, como a OpenCL desenvolvida pela *Khronos Group*.

- Expandir o modelo utilizado para considerar a questão temporal da simulação.

GLOSSÁRIO

Coordenadas Naturais: Nome dado ao intervalo [-1,1] utilizado no Plano do elemento.

CUDA core (SP): Núcleos internos dos *SMs* que executam as operações numéricas.

Deformações de cisalhamento: Deformações tangenciais que normalmente ocorrem com a aplicação de forças paralelas e opostas em um objeto.

Reduction : Basicamente a *reduction* é uma operação que mapeia uma entrada de n x n elementos para um único escalar. Neste trabalho a operação de *reduction* efetua a soma dos valores de entrada, ou seja, dada uma entrada de n valores o resultado do *reduction* será a soma de todos os n valores.

Plano do domínio: Plano dimensional onde todos os elementos são mapeados, possivelmente compartilhando nodos. Representa a malha.

Plano do elemento: Plano tridimensional onde todos os eixos estão limitados ao intervalo [-1,1] . Cada elemento é mapeado para seu plano (Plano do elemento) e posteriormente suas coordenadas são convertidas para o Plano do domínio.

Quadratura de Gauss: Regra utilizada para calcular a aproximação da integral de uma dada função. Normalmente baseada em um somatório de alguns pontos conhecidos da função e seus respectivos pesos, dentro do domínio de integração.

Streaming Multiprocessor (SM): Processadores da *GPU* compostos pelos núcleos *CUDA*

Warp: Unidade básica de execução da *GPU*. As *threads* de cada bloco são divididas em *warps*, sendo que quando escalonadas pela *GPU* todas as *threads* do *warp* executam em sincronia. O tamanho atual dos *warps* é de 32 *threads*.

REFERÊNCIAS

ACCELEREYES. **ArrayFire Benchmarks**. [S.l.: s.n.], 2013. <Disponível em: http://www.accelereyes.com/products/benchmarks_arrayfire#>. Acesso em: Outubro de 2013.

ANANTHASURESH, G. K. **Basic introduction to finite element analysis**. <Disponível em: http://www.mecheng.iisc.ernet.in/~suresh/me237/content.html>. Acesso em: Abril de 2013, Notas de aula.

ATCP. **Tabelas de propriedades**. [S.l.: s.n.], 2013. <Disponível em: http://www.atcp.com.br/pt/produtos/caracterizacao-materiais/propriedades-materiais/tabelas-propriedades.html>. Acesso em: Maio de 2013.

AZEVEDO, A. F. M. **Método dos Elementos Finitos**. <Disponível em: http://civil.fe.up.pt/pub/apoio/ano5/mnae/Livro_MEF_AA.htm>. Acesso em: Março de 2013.

BESSONE, L.; JR., E. F. B. **Evaluation of Different Post Systems**: finite element method. [S.l.: s.n.], 2010. <Disponível em: http://www.scielo.cl/scielo.php?pid=S0718-381X2010000300004&script=sci_arttext>. Acesso em: Abril de 2013.

BONJORNO, J. R. et al. **Física Fundamental - Novo**. [S.l.]: FTD S.A., 1999. ISBN 85-322-4371-1.

BORESI, A. P.; CHONG, K.; LEE, J. D. **Elasticity in Engineering Mechanics**. [S.l.]: John Wiley & Sons, 2011. ISBN 978-0-470-88036-4.

BRODTKORB, A. R. et al. State-of-the-art in heterogeneous computing. **Scientific Programming**, [S.l.], 2010. Vol 18. 1-13 p. ISSN 1058-9244.

CHEN, Z. et al. **GICUDA**: a parallel program for 3d correlation imaging of large scale gravity and gravity gradiometry data on graphics processing units with cuda. [S.l.: s.n.], 2012.

CRISTINA, M.; CUNHA, C. **Métodos númericos**. [S.l.]: Editora da Unicamp, 2000. ISBN: 85-268-0521-5.

DORNELES, R. V. et al. **Simulação de Modelos Físicos Utilizando Processamento de Alto Desempenho**. [S.l.]: Universidade de Caxias do Sul e Universidade do Oeste do Paraná, 2008.

EBAH. **Ensaio sobre Tração**. [S.l.: s.n.], 2013. <Disponível em: `http://www.ebah.com.br/content/ABAAAA5gEAA/ensaio-tracao`>. Acesso em: Abril de 2013.

FAGAN, M. J. **Finite Element Analysis Theory and Practice**. [S.l.]: Addison Wesley Longman, 1992. ISBN 0-582-02247-9.

FENLASON, J. **GNU gprof**. [S.l.: s.n.], 1993. <Disponível em: `http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html`>. Acesso em: Outubro de 2013.

FISH, J.; BELYTSCHKO, T. **A First Course in Finite Elements**. [S.l.]: John Wiley & Sons Ltd, 2007. ISBN 978-0-470-03580-1.

FISICAFACIL. **Elasticidade**. [S.l.: s.n.], 2013. <Disponível em: `http://www.fisicafacil.pro.br/elastica.htm`>. Acesso em: Abril de 2013.

GAMASUTRA. **U.S. Air Force Creates Powerful Supercomputer Out Of PS3s**. [S.l.: s.n.], 2010. <Disponível em: `http://www.gamasutra.com/view/news/31784/US_Air_Force_Creates_Powerful_Supercomputer_Out_Of_PS3s.php#.URZr9VpFug4`>. Acesso em: Janeiro de 2013.

GLVIS. **OpenGL Visualization Tool and Library**. [S.l.: s.n.], 2010. <Disponível em: `https://code.google.com/p/glvis/wiki/OptionsAndUse`>. Acesso em: Junho de 2013.

GRISA, M. **GPU Computing**: implementação do gradiente conjugado em cuda. [S.l.: s.n.], 2010.

HAGER, G.; WELLEIN, G. **Introduction to High Performance Computing for Scientists and Engineers**. [S.l.]: CRC Press, 2012. ISBN-13 978-1439811924.

INTEL. **Intel Processor Comparison**. [S.l.: s.n.], 2013. <Disponível em: `http://www.intel.com/content/www/us/en/processor-comparison/compare-intel-processors.html`>. Acesso em: Janeiro de 2013.

KATAN, P. **MATLAB Guide to Finite Elements**. 2.ed. [S.l.]: Springer, 2008. ISBN 978-3-540-70697-7.

KHRONOS. **Open Computing Language**. [S.l.: s.n.], 2008. <Disponível em: http://www.khronos.org/opencl/>. Acesso em: Abril de 2013.

KIRK, D. B.; W. HWU, W. mei. **Programming Massively Parallel Processors A Hands-on Approach**. 2.ed. [S.l.]: Morgan Kaufmann, 2012. ISBN-13 978-0124159921.

LAKES, R. **Meaning of Poisson's ratio**. [S.l.: s.n.], 2013. <Disponível em: http://silver.neep.wisc.edu/~lakes/PoissonIntro.html>. Acesso em: Maio de 2013.

LIBRARY, O. G. **Open Graphics Library**. [S.l.: s.n.], 1992. <Disponível em: http://www.opengl.org/>. Acesso em: Abril de 2013.

MARTINOTTO, A. L. **Resolução de Sistemas de Equações Lineares através de Métodos de Decomposição de Domínio**. [S.l.: s.n.], 2004.

MATHWORKS. **MATLAB**. [S.l.: s.n.], 2013. <Disponível em: http://www.mathworks.com>. Acesso em: Junho de 2013.

MCINTOSH-SMITH, S. **The GPU Computing Revolution, From Multi-Core CPUs to Many-Core Graphics Processors**. [S.l.: s.n.], 2011. <Disponível em: https://connect.innovateuk.org/c/document_library/get_file?uuid=d468f129-9a07-46f8-82e5-2aa7512f4d59&groupId=47465>. Acesso em: Janeiro de 2013.

MEDINA, A. C.; CHWIF, L. **Modelagem e simulação de eventos discretos**. 3.ed. [S.l.]: Leonardo Chwif, 2010. ISBN 8590597830.

MFEM. **Finite Element Discretization Library**. [S.l.: s.n.], 2011. <Disponível em: https://code.google.com/p/mfem/>. Acesso em: Junho de 2013.

MICROSOFT. **DirectX**. [S.l.: s.n.], 1995. <Disponível em: http://windows.microsoft.com/pt-BR/Windows7/products/features/directx-11>. Acesso em: Abril de 2013.

MICROSOFT. **Microsoft DirectCompute**. [S.l.: s.n.], 2008. <Disponível em: http://www.microsoft.com/en-us/download/details.aspx?id=16995>. Acesso em: Abril de 2013.

MOORKAMP, M. et al. **Massively parallel forward modeling of scalar and tensor gravimetry data**. [S.l.: s.n.], 2010.

NEPTEL. **Energy Methods**. [S.l.: s.n.], 2013. <Disponível em: http://www.nptel.iitm.ac.in/courses/Webcourse-contents/IIT-ROORKEE/strength\%20of\%20materials/lects\%20\&\%20picts/image/lect38/lecture38.htm>. Acesso em: Abril de 2013.

NIEMANN, C.; HUDERT, S.; EYMANN, T. On Computer Simulation as a Component in Information Systems Research. **Global Perspectives on Design Science Research**, [S.l.], 2010. 167-179 p. ISBN 978-3-642-13335-0.

NVIDIA. **Compute Unified Device Architecture**. [S.l.: s.n.], 2007. <Disponível em: http://www.nvidia.com/object/cuda_home_new.html>. Acesso em: Abril de 2013.

NVIDIA. **What is GPU Computing?** [S.l.: s.n.], 2013. <Disponível em: http://www.nvidia.com/object/what-is-gpu-computing.html>. Acesso em: Janeiro de 2013.

NVIDIA. **NVIDIA Visual Profiler**. [S.l.: s.n.], 2013. <Disponível em: https://developer.nvidia.com/nvidia-visual-profiler>. Acesso em: Outubro de 2013.

RAHMADDIANSYAH; RASHID, N. A. **SPEEDING UP INDEX CONSTRUCTION WITH GPU FOR DNA DATA SEQUENCES**. [S.l.: s.n.], 2011. <Disponível em: http://www.icoci.cms.net.my/proceedings/2011/papers/92.pdf>. Acesso em: Novembro de 2013.

RIBEIRO, F. Introdução ao Método dos Elementos Finitos–Notas de Aula do Prof. **Fernando LB Ribeiro. COPPE–UFRJ–Programa de Engenharia Civil. Rio de Janeiro**, [S.l.], 2004.

SANDERS, J.; KANDROT, E. **Cuda By Example**: an introduction to general-purpose gpu programming. 1.ed. [S.l.]: Addison-Wesley Professional, 2010. ISBN-13 978-0131387683.

SILVA, S. da. **Introdução ao Método dos Elementos Finítos**. [S.l.]: Universidade Estadual do Oeste do Paraná. UNIOESTE/Campus de Foz do Iguaçu, 2009. <Disponível em: http://www.inf.unioeste.br/~rogerio/Analise-Estruturas1.pdf>. Acesso em: Março de 2013.

SMITH, W. F.; HASHEMI, J. **Fundamentos da engenharia e ciência dos materiais**. [S.l.]: Bookman, 2010. ISBN 978007352940.

TANENBAUM, A. S. **Modern Operating Systems**. 2.ed. [S.l.]: Prentice Hall of India, 2007. ISBN-10 8120320638.

TEACHENGINEERING. **Mechanics of Elastic Solids**. [S.l.: s.n.], 2013. <Disponível em: http://www.teachengineering.org/view_lesson.php?url=collection/cub_/lessons/cub_surg/cub_surg_lesson02.xml>. Acesso em: Maio de 2013.

WHITEHEAD, N.; FIT-FLOREA, A. **Precision & Performance**: floating point and ieee 754 compliance for nvidia gpus. [S.l.: s.n.], 2011. <Disponível em: http://developer.download.nvidia.com/assets/cuda/files/NVIDIA-CUDA-Floating-Point.pdf>. Acesso em: Novembro de 2013.

ApêndiceA

A.1 Implementação do MEF para um elemento quadrilátero

Nesta seção será apresentada uma implementação em MATLAB (MATHWORKS, 2013) para o exemplo descrito na seção 3.5.

Trecho de Código A.1: Função para calcular a matriz Jacobiana

```
1         % e, n são os valores das coordenadas (x,y) no plano do elemento,
              definido entre [1,-1]
2         % x1,y1, x2,y2, x3,y3, x4,y4 são as coordenadas físicas do elemento no
              plano da simulação
3         function [ J ] = GerarMatriz_Jacobiana(e,n, x1,y1, x2,y2, x3,y3, x4,y4 )
4            % matriz com as derivadas parciais da funções de forma, sobre as
                 coordenadas naturais no plano do elemento
5             dpJ = [n-1 1-n 1+n -n-1; e-1 -e-1 1+e 1-e];
6
7             quadNodePos = [x1, y1; x2, y2; x3, y3; x4, y4];
8
9             J = 1/4 * dpJ * quadNodePos;
10        end
```

Trecho de Código A.2: Função para calcular o determinante da matriz Jacobiana

```
1         % e, n são os valores das coordenadas (x,y) no plano do elemento,
              definido entre [1,-1]
2         % x1,y1, x2,y2, x3,y3, x4,y4 são as coordenadas físicas do elemento no
              plano da simulação
3         function [ dJ ] = CalcularDet_Jacobiana(e,n,x1,y1, x2,y2, x3,y3, x4,y4 )
4                 J = GerarMatriz_Jacobiana(e,n,  x1,y1, x2,y2, x3,y3, x4,y4 );
5                 dJ = det(J);
6         end
```

Trecho de Código A.3: Função para gerar a matriz D

```
1         % E, v são, respectivamente, o módulo de Young e o coeficiente de
              Poisson
2         function [ D ] = GerarMatriz_D(E,v)
3                 D = E/(1-v^2)*[1 v 0; v 1 0; 0 0 (1-v)/2];
4         end
```

Trecho de Código A.4: Função para calcular a matriz B

```
% e, n são os valores das coordenadas (x,y) no plano do elemento,
    definido entre [1,-1]
% x1,y1, x2,y2, x3,y3, x4,y4 são as coordenadas físicas do elemento no
    plano da simulação
function [ B ] = GerarMatriz_B(e,n, x1,y1, x2,y2, x3,y3, x4,y4)
        J = GerarMatriz_Jacobiana(e,n,  x1,y1, x2,y2, x3,y3, x4,y4 );
        invJ = inv(J);

        b = invJ * (1/4) * dpJ;

        B = [  b(1,1) 0 b(1,2) 0 b(1,3) 0 b(1,4) 0;
                0 b(2,1) 0 b(2,2) 0 b(2,3) 0 b(2,4);
                b(2,1) b(1,1) b(2,2) b(1,2) b(2,3) b(1,3) b(2,4) b(1,4)
          ];
end
```

Trecho de Código A.5: Função para aplicar a condição de contorno de Dirichlet em uma linha da matriz

```
% K é a matriz global de rigidez e rowNum e a linha onde será aplicado a
     condição de contorno de Dirichlet
function [ Kd ] = Dirichlet( K, rowNum )
    K(rowNum, :) = 0;
    K(rowNum, rowNum) = 1;

    Kd = K;
end
```

Trecho de Código A.6: Função para calcular a matriz K

```
% E, v são, respectivamente, o módulo de Young e o coeficiente de
    Poisson
% x1,y1, x2,y2, x3,y3, x4,y4 são as coordenadas físicas do elemento no
    plano da simulação
function [ K ] = GerarMatriz_K(E, v, x1,y1, x2,y2, x3,y3, x4,y4)
    % Parâmetros para aplicar a Quadratura de Gauss
        QG_x = [-1/sqrt(3), 1/sqrt(3), -1/sqrt(3), 1/sqrt(3)];
        QG_y = [-1/sqrt(3), -1/sqrt(3), 1/sqrt(3), 1/sqrt(3)];

        K = zeros(8,8);
        for i=1:4
                B = GerarMatriz_B(QG_x(i), QG_y(i), x1,y1, x2,y2, x3,y3,
                     x4,y4 );
                K = K + B' * GerarMatriz_D(E, v) * B *
                    CalcularDet_Jacobiana(QG_x(i), QG_y(i), x1,y1, x2,y2
                    , x3,y3, x4,y4);
        end
end
```

Trecho de Código A.7: Resolvendo o exemplo apresentado na Seção 3.5

```
% Descobrindo as deformações para um elemento quadrilátero, considerando
     uma força de 10N
% aplicadas a aresta 3-4
```

```
 3         K = GerarMatriz_K(3 * 10^7, 0.3, 0,0, 1,0, 1,1, 0,1);
 4         F = [0 0 0 0 0 -10 0 -10];
 5
 6         dirichletLinhas = [1,2,7,8];
 7         for i = 1:4
 8             K = Dirichlet(K, dirichletLinhas(i));
 9             F(dirichletLinhas(i)) = 0;
10         end
11
12         % Resultado da deformação
13         u = K \ F'
```

A.2 Implementação do Gradiente Conjugado em *CUDA* na MFEM

Trecho de Código A.8: Arquivo *header* NVCGSolver.cuh

```
 1 // Copyright (c) 2010, Lawrence Livermore National Security, LLC. Produced at
 2 // the Lawrence Livermore National Laboratory. LLNL-CODE-443211. All Rights
 3 // reserved. See file COPYRIGHT for details.
 4 //
 5 // This file is part of the MFEM library. For more information and source code
 6 // availability see http://mfem.googlecode.com.
 7 //
 8 // MFEM is free software; you can redistribute it and/or modify it under the
 9 // terms of the GNU Lesser General Public License (as published by the Free
10 // Software Foundation) version 2.1 dated February 1999.
11 //
12 // Edited by Lucas Signor Schwochow (lsschwoc@ucs.br)
13
14 #include <iostream>
15 #include <iomanip>
16 #include "vector.hpp"
17 #include "matrix.hpp"
18 #include "sparsemat.hpp"
19
20 void NV_CG( const SparseMatrix &A, const Vector &b, Vector &x,
21          int print_iter=0, int max_num_iter=1000,
22          double RTOLERANCE=10e-12, double ATOLERANCE=10e-24);
```

Trecho de Código A.9: Arquivo *source* NVCGSolver.cu

```
 1
 2 // Copyright (c) 2010, Lawrence Livermore National Security, LLC. Produced at
 3 // the Lawrence Livermore National Laboratory. LLNL-CODE-443211. All Rights
 4 // reserved. See file COPYRIGHT for details.
 5 //
 6 // This file is part of the MFEM library. For more information and source code
 7 // availability see http://mfem.googlecode.com.
 8 //
 9 // MFEM is free software; you can redistribute it and/or modify it under the
10 // terms of the GNU Lesser General Public License (as published by the Free
11 // Software Foundation) version 2.1 dated February 1999.
12 //
```

```
13 // Edited by Lucas Signor Schwochow (lsschwoc@ucs.br)
14
15 #include <stdio.h>
16 #include <limits>
17 #include <cmath>
18
19 #include <cuda_runtime.h>
20 #include <cusparse_v2.h>
21 #include <cublas_v2.h>
22
23 #include "NVCGSolver.cuh"
24
25
26 void myCudaCheck (cudaError_t r) {
27     if (r == cudaSuccess) {
28         printf("OK\n");
29     } else {
30         cudaDeviceReset();
31         printf("FAIL\n");
32         exit(1);
33     }
34 }
35
36 __global__ void _matrixByVector(const double *A, const int *I, const int *J,
37                         const double *x, double *y, int size) {
38
39
40     __shared__ double sums[THREAD_PER_BLOCK];
41
42     double tempY = 0;
43     int i = 0;
44     int rowIndex = (THREAD_PER_BLOCK * blockIdx.x + threadIdx.x) >> 5;
45
46     if (rowOffset >= size) return;
47
48     int rowOffset = I[rowIndex];
49     int rowLimit = I[rowIndex + 1];
50     int nIdx = threadIdx.x % 32;
51
52     for (i = rowOffset + nIdx; i < rowLimit; i += 32 ){
53         tempY += A[i] * x[J[i]];
54     }
55
56     sums[threadIdx.x] = tempY;
57
58     if ( (threadIdx.x >> 4) % 2 == 0) {
59         sums[threadIdx.x] += sums[threadIdx.x + 16];
60         sums[threadIdx.x] += sums[threadIdx.x + 8];
61         sums[threadIdx.x] += sums[threadIdx.x + 4];
62         sums[threadIdx.x] += sums[threadIdx.x + 2];
63         sums[threadIdx.x] += sums[threadIdx.x + 1];
64     }
65
66     if (threadIdx.x % 32 == 0) {
67         y[rowIndex] = sums[threadIdx.x];
68     }
69
70 }
71
```

```
72 #define THREAD_PROD 512
73 __global__ void _dotProduct(const double *v1, const double *v2, double *r, int
       size) {
74          __shared__ double sums[THREAD_PROD];
75
76     double tempY = 0;
77     int i = 0;
78     int rowOffset = 0;
79     int rowWidth = size;
80
81     for (i = rowOffset + threadIdx.x; i < rowWidth; i += THREAD_PROD ){
82         tempY += v1[i] * v2[i];
83     }
84
85     sums[threadIdx.x] = tempY;
86     __syncthreads();
87
88
89     if (threadIdx.x < (THREAD_PROD >> 1)) {
90         for (i = (THREAD_PROD >> 1); i > 0; i >>= 1) {
91
92             sums[threadIdx.x] += sums[threadIdx.x + i];
93         }
94     }
95
96     if (threadIdx.x == 0) {
97         (*r) = sums[0];
98     }
99 }
100
101
102 __global__ void _addVector(const double* v1, const double alpha, const double*
       v2, double *r, int size) {
103     int idx = blockDim.x * blockIdx.x + threadIdx.x;
104     double a,b;
105     if (idx < size) {
106         a = v1[idx];
107         b = v2[idx];
108         r[idx] = a + alpha * b;
109     }
110 }
111
112
113 void NV_CG( const SparseMatrix &A, const Vector &b, Vector &x,
114                       int print_iter, int max_num_iter,
115                       double RTOLERANCE, double ATOLERANCE){
116
117        int i, dim = x.Size();
118        double den, nom, nom0, betanom, alpha, beta, r0;
119        Vector r(dim), d(dim), Ad(dim);
120
121        double *d_x, *d_b, *d_r, *d_d, *d_Ad, *d_A;
122        int *d_I, *d_J;
123        int *I = A.GetI();
124        int *J = A.GetJ();
125        double *d_temp;
126
127
128        myCudaCheck(cudaMalloc(&d_A, sizeof(double) * I[A.Size()]));
```

```
129         myCudaCheck(cudaMalloc(&d_I, sizeof(int) * (A.Size() + 1)));
130         myCudaCheck(cudaMalloc(&d_J, sizeof(int) * I[A.Size()]));
131         myCudaCheck(cudaMalloc(&d_b, sizeof(double) * b.Size()));
132         myCudaCheck(cudaMalloc(&d_x, sizeof(double) * x.Size()));
133         myCudaCheck(cudaMalloc(&d_r, sizeof(double) * dim));
134         myCudaCheck(cudaMalloc(&d_d, sizeof(double) * dim));
135         myCudaCheck(cudaMalloc(&d_Ad, sizeof(double) * dim));
136         myCudaCheck(cudaMalloc(&d_temp, sizeof(double)));
137
138
139         myCudaCheck(cudaMemcpy(d_A, A.GetData(), sizeof(double) * I[A.Size()],
                cudaMemcpyHostToDevice));
140         myCudaCheck(cudaMemcpy(d_I, I, sizeof(int) * (A.Size() + 1),
                cudaMemcpyHostToDevice));
141         myCudaCheck(cudaMemcpy(d_J, J, sizeof(int) * I[A.Size()],
                cudaMemcpyHostToDevice));
142         myCudaCheck(cudaMemcpy(d_x, x.GetData(), sizeof(double) * x.Size(),
                cudaMemcpyHostToDevice));
143         myCudaCheck(cudaMemcpy(d_b, b.GetData(), sizeof(double) * b.Size(),
                cudaMemcpyHostToDevice));
144         cudaMemset(d_r, 0.0, sizeof(double) * dim);
145         cudaMemset(d_d, 0.0, sizeof(double) * dim);
146         cudaMemset(d_Ad, 0.0, sizeof(double) * dim);
147         cudaMemset(d_temp, 0.0, sizeof(double));
148
149
150         _matrixByVector<<<ceil(A.Size() / (THREAD_PER_BLOCK/32)) + 1,
                THREAD_PER_BLOCK>>>(d_A, d_I, d_J, d_x, d_r, A.Size());
151
152         // r = b - A x
153         _addVector<<<ceil(dim/512) + 1, 512>>>(d_b, -1, d_r, d_r, dim);
154
155         //d = r;
156         cudaMemcpy(d_d, d_r, sizeof(double) * dim, cudaMemcpyDeviceToDevice);
157
158
159         _dotProduct<<<1, THREAD_PROD>>>(d_r, d_r, d_temp, dim);
160          cudaMemcpy(&nom0, d_temp, sizeof(double), cudaMemcpyDeviceToHost);
161         nom = nom0;
162
163
164         if (print_iter == 1)
165            cout << "   Iteration : " << setw(3) << 0 << "  (r, r) = "
166                 << nom << endl;
167
168         if ( (r0 = nom * RTOLERANCE) < ATOLERANCE) r0 = ATOLERANCE;
169         if (nom < r0)
170            return;
171
172         _matrixByVector<<<ceil(A.Size() / (THREAD_PER_BLOCK/32)) + 1,
                THREAD_PER_BLOCK>>>(d_A, d_I, d_J, d_d, d_Ad, A.Size());
173
174
175         _dotProduct<<<1, THREAD_PROD>>>(d_d, d_Ad, d_temp, dim);
176         cudaMemcpy((void *)&den, d_temp, sizeof(double), cudaMemcpyDeviceToHost);
177
178         if (den <= 0.0) {
179            if (nom0 > 0.0)
180               cout <<"Operator A is not postive definite. (Ar,r) = "
```

```
181                   << den << endl;
182            return;
183         }
184
185
186         // start iteration                              //  d = r, Ad = A r
187
188         for(i = 1; i<max_num_iter ;i++) {
189            alpha= nom/den;                               // alpha = (r_o,r_o)/(Ar_o,
                   r_o)
190
191
192            _addVector<<<ceil(x.Size()/512) + 1, 512>>>(d_x, alpha, d_d, d_x, x.
                   Size());
193            _addVector<<<ceil(r.Size()/512) + 1, 512>>>(d_r, -alpha, d_Ad, d_r, r.
                   Size());
194
195
196             // betanom = (r_o, r_o)
197            _dotProduct<<<1, THREAD_PROD>>>(d_r, d_r, d_temp, dim);
198             cudaMemcpy((void *)&betanom, d_temp, sizeof(double),
                       cudaMemcpyDeviceToHost);
199
200            if (print_iter == 1)
201               cout << "   Iteration : " << setw(3) << i << "  (r, r) = "
202                    << betanom << endl;
203
204            if ( betanom < r0) {
205               if (print_iter == 2)
206                  cout << "Number of CG iterations: " << i << endl;
207               else
208                  if (print_iter == 3)
209                     cout << "(r_0, r_0) = " << nom0 << endl
210                          << "(r_N, r_N) = " << betanom << endl
211                          << "Number of CG iterations: " << i << endl;
212               break;
213            }
214
215            beta = betanom/nom;                           // beta = (r_n,r_n)/(r_o,r_o
                   )
216
217            //  d = r_n + beta * d
218            _addVector<<<ceil(d.Size()/512) + 1, 512>>>(d_r, beta, d_d, d_d, d.
                   Size());
219
220
221            //   Ad = A d
222            _matrixByVector<<<ceil(A.Size() / (THREAD_PER_BLOCK/32)) + 1,
                   THREAD_PER_BLOCK>>>(d_A, d_I, d_J, d_d, d_Ad, A.Size());
223
224
225            //den = d * Ad;                                 //  den = (d , A d)
226            _dotProduct<<<1, THREAD_PROD>>>(d_d, d_Ad, d_temp, dim);
227             cudaMemcpy((void *)&den, d_temp, sizeof(double),
                       cudaMemcpyDeviceToHost);
228
229            if (den <= 0.0)
230            {
231                double dtest = 0;
```

```
232              _dotProduct<<<1, THREAD_PROD>>>(d_d, d_Ad, d_temp, dim);
233              cudaMemcpy((void *)&dtest, d_temp, sizeof(double),
                     cudaMemcpyDeviceToHost);
234              if (dtest > 0.0)
235                 cout <<"Operator A is not postive definite. (Ad,d) = "
236                      << den << endl;
237           }
238           nom = betanom;                              //  nom = (r_n, r_n)
239        }
240        if (i == max_num_iter && print_iter >= 0)
241        {
242           cerr << "CG: No convergence!" << endl;
243           cout << "(r_0, r_0) = " << nom0 << endl
244                << "(r_N, r_N) = " << betanom << endl
245                << "Number of CG iterations: " << i << endl;
246        }
247
248        cudaMemcpy(x.GetData(), d_x, sizeof(double) * x.Size(),
               cudaMemcpyDeviceToHost);
249
250       cudaFree(d_temp);
251       cudaFree(d_A);
252       cudaFree(d_I);
253       cudaFree(d_J);
254       cudaFree(d_b);
255       cudaFree(d_x);
256       cudaFree(d_r);
257       cudaFree(d_d);
258       cudaFree(d_Ad);
259
260
261     }
```

A.3 Implementação sequencial do Gradiente Conjugado da MFEM

Trecho de Código A.10: Implementação sequencial do Gradiente Conjugado

```
// Copyright (c) 2010, Lawrence Livermore National Security, LLC. Produced at
// the Lawrence Livermore National Laboratory. LLNL-CODE-443211. All Rights
// reserved. See file COPYRIGHT for details.
//
// This file is part of the MFEM library. For more information and source code
// availability see http://mfem.googlecode.com.
//
// MFEM is free software; you can redistribute it and/or modify it under the
// terms of the GNU Lesser General Public License (as published by the Free
// Software Foundation) version 2.1 dated February 1999.
//

#include <iostream>
#include <iomanip>
#include "vector.hpp"
#include "matrix.hpp"
#include "sparsemat.hpp"
```

```
// Conjugate Gradient solver

void CG( const Operator &A, const Vector &b, Vector &x,
         int print_iter=0, int max_num_iter=1000,
         double RTOLERANCE=10e-12, double ATOLERANCE=10e-24){

   int i, dim = x.Size();
   double den, nom, nom0, betanom, alpha, beta, r0;
   Vector r(dim), d(dim), Ad(dim);

   A.Mult( x, r);
   subtract( b, r, r);                          // r = b - A x
   d = r;
   nom0 = nom = r * r;

   if (print_iter == 1)
      cout << "   Iteration : " << setw(3) << 0 << "  (r, r) = "
           << nom << endl;

   if ( (r0 = nom * RTOLERANCE) < ATOLERANCE) r0 = ATOLERANCE;
   if (nom < r0)
      return;

   A.Mult( d, Ad);
   den = d * Ad;


   if (den <= 0.0) {
      if (nom0 > 0.0)
         cout <<"Operator A is not postive definite. (Ar,r) = "
              << den << endl;
      return;
   }

   // start iteration                           //  d = r, Ad = A r
   for(i = 1; i<max_num_iter ;i++) {
      alpha= nom/den;                            // alpha = (r_o,r_o)/(Ar_o,r_o)

      add(x, alpha, d, x);                       //   x =   x + alpha * d
      add(r, -alpha, Ad, r);                     // r_n = r_o - alpha * Ad
      betanom = r*r;                             // betanom = (r_o, r_o)

      if (print_iter == 1)
         cout << "   Iteration : " << setw(3) << i << "  (r, r) = "
              << betanom << endl;

      if ( betanom < r0) {
         if (print_iter == 2)
            cout << "Number of CG iterations: " << i << endl;
         else
            if (print_iter == 3)
               cout << "(r_0, r_0) = " << nom0 << endl
                    << "(r_N, r_N) = " << betanom << endl
                    << "Number of CG iterations: " << i << endl;
         break;
      }

      beta = betanom/nom;                        // beta = (r_n,r_n)/(r_o,r_o)
```

```
      add(r, beta, d, d);                          //    d = r_n + beta * d
      A.Mult(d, Ad);                               //   Ad = A d
      den = d * Ad;                                //  den = (d , A d)
      if (den <= 0.0)
      {
         if (d * d > 0.0)
            cout <<"Operator A is not postive definite. (Ad,d) = "
                 << den << endl;
      }
      nom = betanom;                               //  nom = (r_n, r_n)
   }
   if (i == max_num_iter && print_iter >= 0)
   {
      cerr << "CG: No convergence!" << endl;
      cout << "(r_0, r_0) = " << nom0 << endl
           << "(r_N, r_N) = " << betanom << endl
           << "Number of CG iterations: " << i << endl;
   }
}
```

A.4 Malha utilizada na análise de desempenho

Trecho de Código A.11: Formato do arquivo da malha

```
dimension
3

# <número de elementos>
# <atributo do elemento> <tipo da geometria> <indice do vértice 1> ... <indice
    do vértice m>

elements
8
2 5 0 1 10 9 18 19 28 27
2 5 1 2 11 10 19 20 29 28
2 5 2 3 12 11 20 21 30 29
1 5 3 4 13 12 21 22 31 30
2 5 4 5 14 13 22 23 32 31
2 5 5 6 15 14 23 24 33 32
2 5 6 7 16 15 24 25 34 33
2 5 7 8 17 16 25 26 35 34

#<número de elementos de fronteira>
#<atributo do elemento de fronteira> <tipo da geometria> < índice do vértice 1>
    ... <índice do vértice n>

boundary
34
1 3 9 10 1 0
2 3 0 1 19 18
3 3 10 9 27 28
3 3 9 0 18 27
3 3 18 19 28 27
3 3 10 11 2 1
3 3 1 2 20 19
3 3 11 10 28 29
```

```
3 3 19 20 29 28
2 3 11 12 3 2
2 3 2 3 21 20
2 3 12 11 29 30
2 3 20 21 30 29
2 3 12 13 4 3
2 3 3 4 22 21
2 3 13 12 30 31
2 3 21 22 31 30
2 3 13 14 5 4
2 3 4 5 23 22
2 3 14 13 31 32
2 3 22 23 32 31
2 3 14 15 6 5
2 3 5 6 24 23
3 3 15 14 32 33
3 3 23 24 33 32
3 3 15 16 7 6
3 3 6 7 25 24
3 3 16 15 33 34
3 3 24 25 34 33
3 3 16 17 8 7
3 3 7 8 26 25
3 3 8 17 35 26
2 3 17 16 34 35
1 3 25 26 35 34

#<número de vértices>
#<dimensão>
#<coordenada 1> ... <coordenada <dimensão>>

vertices
36
3
0 0 0
1 0 0
2 0 0
3 0 0
4 0 0
5 0 0
6 0 0
7 0 0
8 0 0
0 1 0
1 1 0
2 1 0
3 1 0
4 1 0
5 1 0
6 1 0
7 1 0
8 1 0
0 0 1
1 0 1
2 0 1
3 0 1
4 0 1
5 0 1
6 0 1
```

```
7 0 1
8 0 1
0 1 1
1 1 1
2 1 1
3 1 1
4 1 1
5 1 1
6 1 1
7 1 1
8 1 1
```

A.5 Gráficos

A.5.1 Multiplicação matriz-vetor

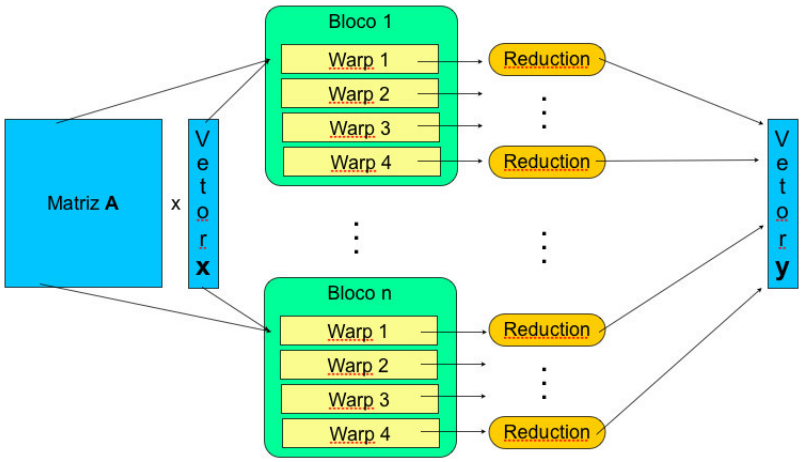

Figura A.1: Representação da multiplicação matriz-vetor implementada em *GPU*

A.5.2 Produto escalar

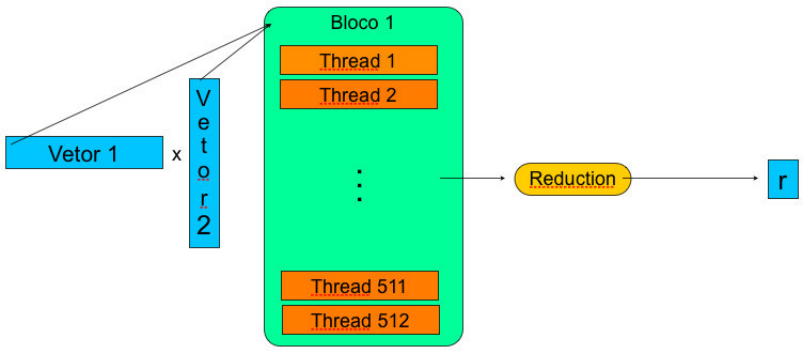

Figura A.2: Representação do produto escalar implementado em *GPU*

A.5.3 Soma de vetores

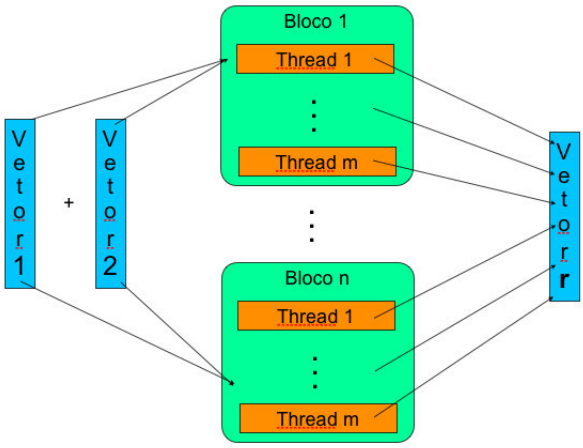

Figura A.3: Representação da soma de vetores implementada em *GPU*