UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE TECNOLOGIA E GEOCIÊNCIAS
DEPARTAMENTO DE ENGENHARIA CIVIL E AMBIENTAL
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA CIVIL

CICERO VITOR CHAVES JUNIOR

ACELERAÇÃO DE SIMULAÇÕES COMPUTACIONAIS EM PROBLEMAS DE DINÂMICA ESTRUTURAL E DE ESCOAMENTO EM MEIOS POROSOS

Recife
2020

CICERO VITOR CHAVES JUNIOR

ACELERAÇÃO DE SIMULAÇÕES COMPUTACIONAIS EM PROBLEMAS DE DINÂMICA ESTRUTURAL E DE ESCOAMENTO EM MEIOS POROSOS

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Civil da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do Título de Mestre em Engenharia Civil.

Área de concentração: Estruturas.

Orientador: Prof. Dr. Paulo Marcelo Vieira Ribeiro.

Recife
2020

Catalogação na fonte
Bibliotecária Margareth Malta, CRB-4 / 1198

C512a Chaves Junior, Cicero Vitor.
Aceleração de simulações computacionais em problemas de dinâmica estrutural e de escoamento em meios porosos / Cicero Vitor Chaves Junior. – 2020.
87 folhas, il., gráfs., tabs.

Orientador: Prof. Dr. Paulo Marcelo Vieira Ribeiro.

Dissertação (Mestrado) – Universidade Federal de Pernambuco. CTG. Programa de Pós-Graduação em Engenharia Civil, 2020.
Inclui Referências e Apêndices.

1. Engenharia Civil. 2. Elementos finitos. 3. Computação paralela. 4. GPGPU. 5. CUDA. 6. Simulação dinâmica. I. Ribeiro, Paulo Marcelo Vieira. (Orientador). II. Título.

UFPE

624 CDD (22. ed.) BCTG/2020-74

CICERO VITOR CHAVES JUNIOR

ACELERAÇÃO DE SIMULAÇÕES COMPUTACIONAIS EM PROBLEMAS DE DINÂMICA ESTRUTURAL E DE ESCOAMENTO EM MEIOS POROSOS

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Civil da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do Título de Mestre em Engenharia Civil.

Área de concentração: Estruturas.

Aprovado em 19 / 02 / 2020.

BANCA EXAMINADORA

Prof. Dr. Paulo Marcelo Vieira Ribeiro (Orientador)
Universidade Federal de Pernambuco

Prof. Dr. Leonardo José do Nascimento Guimarães (Examinador Interno)
Universidade Federal de Pernambuco

Prof. Dr. Marcus Vinícius Girão de Morais (Examinador Externo)
Universidade de Brasília

AGRADECIMENTOS

Agradeço a Deus.

Agradeço aos meus pais, Cirlene e Cicero, pelo amor demonstrado na minha criação e sustento, além do incentivo dado aos meus estudos.

À minha irmã, Camila, por ter me aturado e rido junto a mim.

À minha namorada, Riadny, por ter ficado ao meu lado em todos os momentos, me dando força e carinho. E por todos os momentos que passamos juntos.

Agradeço ao meu orientador, Professor Paulo Marcelo, por todos os conselhos, pela paciência e ajuda nesse período.

Aos meus amigos do mestrado que passaram pelo mesmo junto comigo.

À NVIDIA pela doação da placa aceleradora gráfica Titan XP (mediante o GPU Grant Program) utilizada nesta pesquisa.

À Energi Simulation (Alberta, Canadá) pelo apoio financeiro, através da FADE-UFPE, para realização deste trabalho de pesquisa.

RESUMO

Simulações de elementos finitos com o uso de modelos em larga escala estão se tornando mais frequentes. Entre esses problemas temos a simulação de sismos em barragens e a simulação geomecânica de reservatórios de petróleo. Estratégias modernas aplicadas a problemas transitórios aproveitam a montagem eficiente de matrizes globais, bem como a solução rápida de sistemas de equações em modelos com dezenas de milhões de graus de liberdade. O uso da computação de uso geral em unidades de processamento gráfico (GPGPU) permite extrema paralelização e aceleração desses processos. Nesta direção, o presente trabalho apresenta várias aplicações de problemas de mecânica computacional usando a linguagem de programação C ++ acoplada ao NVCC (NVIDIA CUDA Compiler) para a plataforma CUDA. Essas simulações requerem apenas funções nativas do C ++, sem dependências externas. O código foi desenvolvido em uma estrutura modular, com a implementação híbrida de sub-rotinas em CPU e GPU (Graphical Processing Units). Um solucionador iterativo com o método de gradiente conjugado é apresentado e pode ser acoplado a códigos desenvolvidos em outras linguagens de programação para soluções GPU dedicadas. Duas formas de integração direta são apresentadas para a evolução no tempo. Vários benchmarks são discutidos, incluindo o uso do OpenMP para computação paralela e cálculos na GPU de precisão dupla e única, além de diferentes núcleos de GPU para multiplicação esparsa de vetores matriciais (SpMV). Os resultados obtidos usando as estratégias propostas revelam que os cálculos usando as rotinas descritas são eficazes e fornecem acelerações significativas em relação aos cálculos de thread único.

Palavras-chave: Elementos finitos. Computação paralela. GPGPU. CUDA. Simulação dinâmica.

ABSTRACT

Finite element simulations with the use of large scale models are becoming more frequent. These problems include simulation of earthquakes in dams and geomechanical simulation of petroleum reservoirs. Modern strategies applied to transient problems seize efficient assembly of global matrices as well as the fast solution of system of equations in models with hundreds of millions of degrees of freedom. The use of General-Purpose computing on Graphics Processing Units (GPGPU) enables extreme parallelization and acceleration of these processes. In this direction, the present work presents several applications of computational mechanics problems using the C ++ programming language coupled to the NVCC (NVIDIA CUDA Compiler) for the CUDA platform. These simulations require only native C ++ functions, without external dependencies. The code was developed in a modular structure, with the hybrid implementation of subroutines in CPU and Graphical Processing Units (GPU). An iterative solver with the conjugate gradient method is presented and can be coupled to codes developed in other programming languages for dedicated GPU solution. Two forms of direct integration are presented for evolution over time. Several benchmarks are discussed, including the use of OpenMP for parallel computing and computations on the GPU using double and single precision accuracy, as well as different GPU kernels for sparse matrix-vector multiplication (SpMV). Results obtained using the proposed strategies reveal that computations using the described routines are effective and provide significant speedups over single-threaded computations.

Keywords: Finite elements. Parallel computing. GPGPU. CUDA. Dynamic simulation.

LISTA DE FIGURAS

LISTA DE TABELAS

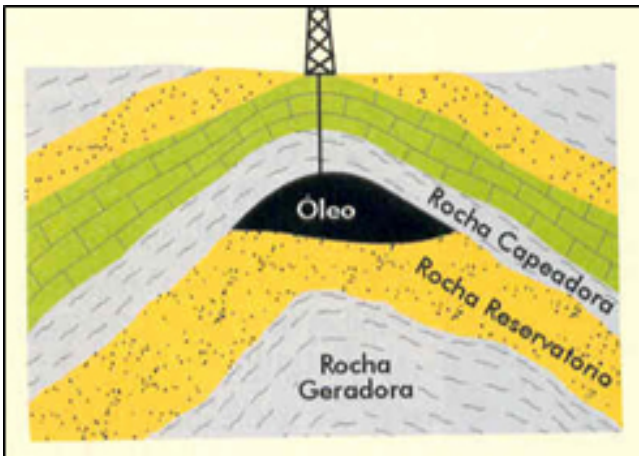
SUMÁRIO

1 INTRODUÇÃO

Há um crescente interesse na modelagem computacional de processos dinâmicos acoplando duas físicas diferentes em todas as áreas da engenharia pela necessidade de simulações mais próximas da realidade. Um destes é o acoplamento entre a análise mecânica e de fluxo em elementos finitos, onde há um grande número de problemas que podem ser simulados. Entre eles temos barragens (Figura 2(a)) e reservatórios de petróleo (Figura 2(b)).

Figura 1 – Exemplos de problemas

(a) Barragem em arco Hoover(EUA).
Fonte: WIKIPEDIA, 2019.



(b) Reservatório de petróleo.
Fonte: TEIXEIRA et al., 2000.

A análise dinâmica de uma estrutura resulta em informações de grande importância sobre ela. Como a estrutura responde aos carregamentos dinâmicos aplicados a ela e se atende aos pré-requisitos funcionais, por exemplo. Com isso determina-se os parâmetros estruturais que mais influenciam a resposta da estrutura e a mesma pode ser modificada. Uma das direções a se caminhar na análise dinâmica de estruturas é o aprimoramento da resolução de grandes sistemas

de equações (MARKELE, 2000). Simulações dinâmicas em barragens são um exemplo dessa necessidade de grandes sistemas, quanto maior for a precisão da resposta da simulação maior a segurança da estrutura. Uma ação excepcional ocorrente em barragens é a ação de sismos, que decorrem da acomodação das camadas da crosta terrestre, podendo ser provocada pelas grandes cargas hidráulicas no reservatório (CIFU, 2011).

A simulação de reservatórios de petróleo visa prever o comportamento do mesmo sob diferentes condições de pressão devido a produção, isto é importante para a otimização econômica da exploração. Mas apenas isto não é o suficiente, esta otimização deve considerar a integridade do reservatório. A ruptura de qualquer parte do reservatório pode acarretar em danos a vidas humanas, seja por ferimentos ou a ocorrência de fatalidades, e danos materiais, seja ao patrimônio próprio ou de terceiros (ANP, 2017). Além de que, deslocamentos excessivos podem provocar a ruptura de dutos, trazendo danos ao meio ambiente devido ao vazamento de óleo ou gás. Ao simular apenas o fluxo não há como considerar estes fatores.

Segundo Logan (2012) o método dos elementos finitos (MEF) é um método numérico muito utilizado na resolução de problemas da engenharia e da física matemática, como: análise estrutural, transferência de calor e escoamento de um fluido. Ele consiste em discretizar o domínio em elementos e achar soluções diferentes, mas de mesma forma, para cada elemento. Umas das formulações deste método é a de resíduos ponderados, que consistem em multiplicar a função aproximada por funções ponderadoras e impor que a média em todo o domínio seja igual a zero (KWON; BANG, 1996). Zienkiewicz e Taylor (2000) listam os três métodos dos resíduos ponderados mais comuns: colocação pontual, onde o resíduo é exatamente zero nos pontos escolhidos; colocação por subdomínio, onde o a média ponderada do resíduo é tida como zero em cada subdomínio; e Galerkin, onde as funções de ponderação são derivadas da função aproximada. A formulação dos resíduos ponderados de Galerkin é utilizada tanto no problema mecânico quanto no de fluxo.

Rutqvist (2017) percebeu que o crescente interesse na modelagem de processos acoplados de fluxo e geomecânicos levou a um grande número de modelos. Ele identificou que a maioria dos modelos utilizados acoplam simuladores de fluxo e mecânicos sequencialmente. Geralmente é utilizado um simulador de fluxo comercial, podendo o simulador mecânico ser comercial ou um código próprio em elementos finitos. Relatou que uma das principais motivações para o desenvolvimento de código próprio é o acesso ao código fonte e ao meio de acoplamento.

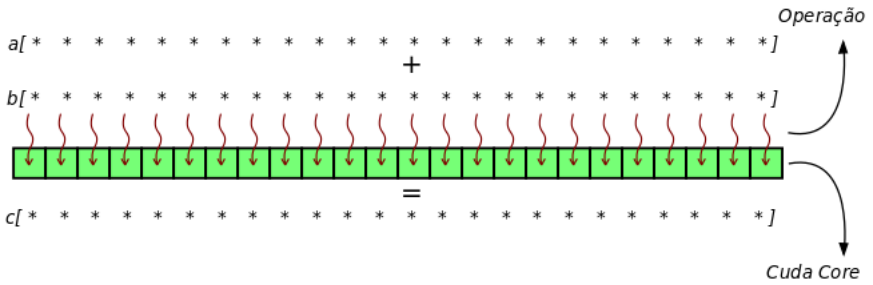
A busca de maiores precisões para as simulações em grandes domínios levou a discretizações cada vez maiores, aumentando em muito a escala do problema. Quanto maior for a escala do problema mais memória será necessário para armazena-lo no computador. Em casos onde a memória é um fator limitador, não é possível a resolução do sistema linear de forma direta, pois esta exige uma grande quantidade de memória da máquina utilizada. Logo, em problemas de grande escala o sistema é resolvido por métodos iterativos, onde a precisão depende da quantidade das iterações realizadas. O método iterativo mais utilizado é o método dos gradientes conjugados.

Problemas de grande escala têm um custo computacional muito elevado, principalmente quando são realizadas análises não lineares ou transientes. Em análises não lineares geralmente são aplicadas técnicas iterativas, onde realizadas diversas análises até a convergência da solução (ZIENKIEWICZ; TAYLOR, 2000). O mesmo pode ser feito em análises transientes, onde a cada avanço temporal a solução é recalculada levando em consideração a solução do tempo anterior.

O custo se torna ainda maior se a não linearidade do problema for muito expressiva, pois desta forma será necessária uma maior quantidade de iterações até a convergência. No caso de problemas transientes o que mensura o custo são o tamanho do passo de tempo e o tempo completo e simulação. O tamanho do passo de tempo tem influência do método de integração temporal escolhido e da variação das cargas externas com o tempo.

A demora das simulações devido ao grande número de iterações a serem realizados, levou a busca de métodos para acelera-las. Um meio de aceleração é a paralelização dos procedimentos. Esta paralelização pode ser feita em CPU através da estrutura OpenMP, ou do pacote MPI, ou outra forma. Mas a algum tempo estruturas de hardware com arquiteturas amplamente paralelizadas vêm sendo utilizadas, são as GPUs (unidades de processamento gráfico). As GPUs foram projetadas inicialmente apenas com o foco no mercado de jogos e vídeos, onde é necessário que vários processadores realizem o mesmo comando. Há poucos anos houve uma mudança nesse segmento e elas começaram a ter arquiteturas que as permitissem ser utilizadas para computação de alto desempenho de propósito geral através de linguagens de programação especializadas. Como a CUDA (*Compute Unified Device Architecture*), projetada pela fabricante de GPUs NVIDIA, ou a OpenCL, mantida pelo grupo Khronos (BARTEZZAGHI et al., 2015). Os dispositivos de GPU atuais são construídos com até centenas de processadores, de maneira que se pode realizar uma grande quantidade de operações de pontos flutuantes paralelamente; um esquema demonstrando a soma de dois vetores é mostrado na Figura 2. Com isso, simulações numéricas podem ser feitas de forma mais eficiente.

Figura 2 – Distribuição das operações paralelas em uma soma de vetores na GPU



Fonte: O Autor, 2020.

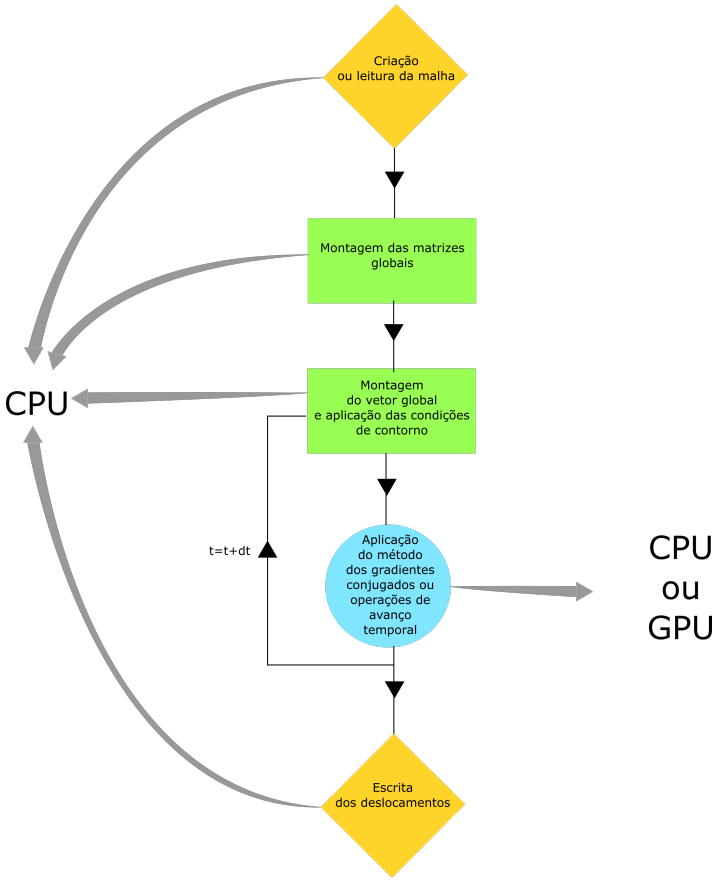
O foco deste trabalho é o desenvolvimento de técnicas de aceleração em GPU de simulações dinâmicas mecânicas e acopladas a simulações de fluxo através do método dos elementos finitos. Para tal foi utilizado a linguagem CUDA em integração com a linguagem C++. Foram

realizadas aplicações estáticas e dinâmicas para avaliar a aceleração. Nas aplicações as estratégias utilizadas foram:

1. resolução em CPU de forma sequencial com o código em C++;
2. resolução em CPU de forma sequencial com o código em MATLAB;
3. resolução em CPU de forma paralela com o código em C++ (OpenMP);
4. resolução em GPU com o código em CUDA sendo chamada pela CPU com o código em C++; e
5. resolução em GPU sendo chamada pela CPU, ambos com o código em MATLAB.

A Figura 3 mostra onde cada parte do código proposto pode é processado.

Figura 3 – Fluxograma com distinção de onde os processos são realizados



Fonte: O Autor, 2020.

1.1 Objetivos

O objetivo geral é estudar o desempenho com a utilização da GPU na aceleração de problemas dinâmicos, bidimensionais e tridimensionais, mecânicos acoplados ao problema de

fluxo utilizando integração explícita e implícita no tempo. E destacam os seguintes objetivos específicos:

- Desenvolver um código para a resolução de problemas em MEF de elasticidade 2D e 3D estáticos em CPU e GPU, e avaliar a aceleração obtida;
- Avaliar a aceleração obtida pela GPU em um problema dinâmico com os diferentes esquemas de integração no tempo;
- Desenvolver código para o acoplamento com o fluxo para problemas em reservatórios de petróleo e avaliar a aceleração obtida;
- Avaliar os ganhos e as limitações de cada estratégia utilizada.

1.2 Metodologia

Foi realizada uma revisão bibliográfica acerca de simulações numéricas mecânicas afim de obter informações sobre o que está sendo feito. As simulações mecânicas têm sido muito utilizadas acopladas a simulações de fluxo não só para analisar como o meio em que o escoamento ocorre se deforma, mas também para que os resultados das simulações de fluxo se aproximem mais da realidade. Na maioria das vezes esse acoplamento consiste na simulação sequencial do problema de fluxo e do mecânico, onde o problema de fluxo alimenta o problema mecânico do mesmo passo de tempo. Logo, acelerar um simulador mecânico além de ser de grande valia para quem tem interesse em problemas mecânicos, também se torna de valor para quem tem interesse em simulações de fluxo transiente acopladas ao problema mecânico.

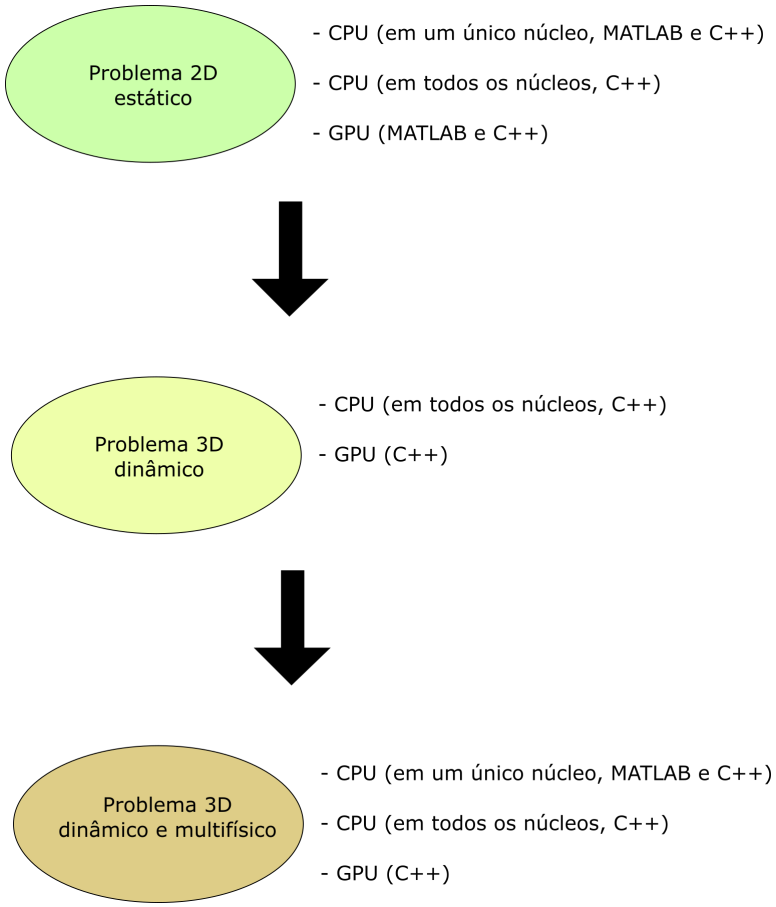
Uma revisão sobre implementações em GPU também foi realizada para identificar os possíveis caminhos a serem tomados. Em todas as implementações de simuladores numéricos foram utilizados métodos iterativos na solução, e na maioria deles o método usado foi o dos gradientes conjugados. A implementação de um solucionador na GPU consiste na implementação de cada uma das operações realizadas por este solucionador.

Os códigos na GPU foram escritos na linguagem CUDA e os códigos na CPU na linguagem C++, pela facilidade de integração com o CUDA e por ser uma linguagem livre de programação. Primeiro foi utilizado o método dos elementos finitos para modelar os problemas mecânicos 2D e foram escritos os códigos para a análise estática para a realização de testes de eficiência. Em seguida foram modelados os problemas mecânicos 3D e os códigos foram escritos para a análise dinâmica no domínio do tempo e novos testes de eficiência foram realizados. Por último foi realizada a modelagem em MEF do escoamento em meios porosos em 3D e o acoplamento unidirecional com os códigos mecânicos. A Figura 4 mostra o progresso dos casos realizados e as estratégias utilizadas.

A fim de tornar mais cômodo a aplicação dos problemas, foi criado um gerador de malha em C++ para domínios retangulares e cúbicos. Porém, as malhas podem ser criadas em um

pré-processador e depois serem lidas pelo programa através de um arquivo. O elemento utilizado no problema de elasticidade em duas dimensões foi o triângulo de tensão constante (CST), já para os problemas de elasticidade mecânica e de fluxo em 3 dimensões foi utilizado o tetraedro de 4 nós.

Figura 4 – Progressão dos casos e linguagens utilizadas



Fonte: O Autor, 2020.

Foi utilizado o método dos gradientes conjugados para a obtenção da solução dos problemas. O algoritmo para este método foi implementado na CPU, de forma a utilizar um único núcleo ou todos os núcleos (através da estrutura OpenMP), e na GPU de três formas diferentes que são explicadas no capítulo 4. No MATLAB foi utilizado a função *pcg* tanto na CPU quanto na GPU.

1.3 Abrangências e limitações

Não são realizadas análises não lineares tanto no problema mecânico como no problema de escoamento. E ambos os problemas possuem simplificações.

Para o problema mecânico o meio foi considerado elástico, linear e sobre o regime de pequenas deformações. As formulações foram realizadas a partir do problema de elasticidade

mecânica e na modelagem pelo método dos elementos finitos foi considerado o elemento CST (*Constatnt Stress Triangle*), para análise 2D, e Tetraedro de tensão constante, para análise 3D.

Para o problema de fluxo em meio poroso, foi considerado o meio completamente saturado e um fluxo monofásico com o fluido pouco compressível. A formulação foi realizada através da equação de balanço de massa em um meio poroso e o escoamento regido pela lei de Darcy. A modelagem pelo método dos elementos finitos foi realizada com o elemento tetraédrico de 4 nós para os problemas 3D.

O acoplamento foi realizado de forma unidirecional (*One-Way*), onde as pressões vindas do problema de fluxo alimentam o problema mecânico. Desta forma, o escoamento não é influenciado pelo problema mecânico.

A aplicações foram processadas em um computador com o processador Intel Core I7 8700 de 3,20 GHz com 6 núcleos de processamento, uma memória RAM de 32 GB e uma GPU NVIDIA TITAN Xp, que possui uma arquitetura Pascal, com 3840 núcleos CUDA e 12 GB de memória. A memória da GPU não foi utilizada até o esgotamento, mas todos os núcleos da GPU foram utilizados.

Por ser um primeiro trabalho de estudo em CUDA no grupo, não foram utilizadas bibliotecas de álgebra linear em CUDA, foram utilizadas apenas as funções mais básicas.

1.4 Organização do trabalho

No capítulo 2 são descritos como algumas simulações de terremotos em barragens e simulações geomecânicas em reservatórios vêm sendo realizadas no país.

No capítulo 3 são apresentados: o desenvolvimento do problema de elasticidade mecânica e escoamento em meios porosos 2D e 3D; a aplicação do método dos resíduos ponderados com o método de Galerkin para a resolução pelo método dos elementos finitos; e os métodos de integração e discretização no tempo.

No capítulo 4 se encontram todos os desenvolvimentos computacionais, os algoritmos utilizados nas implementações em CPU e na GPU, e discussões sobre o uso da arquitetura otimizada da GPU para obter o melhor desempenho na paralelização.

O capítulo 5 traz as aplicações realizadas. Foram feitas 2 aplicações estáticas em elasticidade 2D e 2 aplicações dinâmicas em elasticidade 3D, comparando os métodos de integração. As velocidades das aplicações em GPU foram comparadas às velocidades das aplicações em CPU utilizando programação paralela em OpenMP, para o uso completo do processador.

As considerações finais são apresentadas no capítulo 6.

2 ESTADO DA ARTE

Neste capítulo estão relacionados alguns trabalhos que abordam integração direta no tempo, acoplamento hidromecânico e utilização de GPU em computação de alto desempenho. No âmbito nacional são apresentados trabalhos com simulações computacionais em análise dinâmica estrutural de barragens e análise geomecânica de reservatórios de petróleo.

2.1 Utilização da GPU em Programação de Alto Desempenho

Com a mudança no desenvolvimento de GPUs de alguns anos atrás, ampliando o alcance para o mercado de programação de alto desempenho, estas vêm sendo utilizadas em diversas áreas. Aplicações gerais, como a paralelização de algoritmos de álgebra linear, vêm sendo desenvolvidos.

Clark et al. (2010) usaram a plataforma CUDA na implementação da multiplicação matriz esparsa-vetor para resolver a equação de Dirac discreta em uma aplicação de cromo dinâmica quântica. Usaram os métodos iterativos CG (Gradientes Conjugados) e BiCGstab (Gradientes Biconjugados Estabilizados) com pontos flutuantes de precisões meia, simples e dupla. Além disso implementou um solucionador BiCGstab de precisão mista baseado em uma correção de defeitos e atualizações confiáveis com o cálculo residual e o acúmulo da solução em alta precisão. Tudo em uma GPU GeForce GTX 280 da NVIDIA. Alcançaram um *speedup* de 3 com a utilização de precisão simples e de até 4 com a utilização de meia precisão, em relação a utilização da precisão dupla em todo o algoritmo.

Um *speedup* de até 10 é alcançado por Helfenstein e Koko (2012) em uma implementação na GPU de um algoritmo de gradientes conjugados pré-condicionados proposto pelos mesmos. A matriz esparsa foi armazenada no formato CSR e aplicação foi realizada através da plataforma CUDA em uma GPU Tesla T100 da NVIDIA, com 240 núcleos CUDA.

Ma et al. (2019) implementaram em GPU um algoritmo que realiza a multiplicação tensor-matriz densa. Apresentaram o formato sCOO (semi-Coordinate) para tensor esparso. Utilizando precisão simples em uma GPU NVIDIA P100, eles conseguiram acelerar suas aplicações, em média, em até 30 vezes com relação a um algoritmo paralelo em CPU com 12 núcleos utilizando o OpenMP.

As implementações em GPU também vêm sendo realizadas nas mais variadas áreas da engenharia e da física. Uma revisão de cada etapa envolvida em simulações em MEF na GPU é apresentada por Pikle, Sathe e Vyavhare (2018), demonstrando as dificuldades enfrentadas nas implementações. Entre as discussões vê-se a integração numérica, montagem e formato de matrizes globais, tipos de solucionadores e precondicionadores e a utilização de métodos livres da montagem da matriz global.

Além de um algoritmo de dinâmica de colisão de multi-partículas otimizado para a GPU, Howard, Z. e Nikoubashman (2018) fizeram um modelo de precisão mista para a melhoria de

desempenho em relação a um modelo completamente em precisão dupla. Utilizaram o modelo de programação CUDA para demonstrar que a utilização de precisão mista torna o algoritmo 1,7 vezes mais rápido na GPU Tesla 100 e 1,5 vezes mais rápido na GPU Geforce GTX 1080.

Kim e Park (2019) propuseram uma arquitetura mista CPU-GPU para controle de ruído, ao comparar com um algoritmo baseado apenas em CPU obtiveram diferentes aumentos de velocidades, indo de 4,1 em algumas partes do processo a 241,6 em outras, de forma que o processo inteiro foi acelerado em 82 vezes.

Um método de otimização de enxame de partículas discretas baseado em GPU foi projetado por Shao et al. (2019) para investigar ligas de nanopartículas. Utilizando uma GPU GeForce GTX 1080 da NVIDIA eles propõem a utilização da memória de registradores e um tamanho ótimo para os blocos de *threads* para melhorar o desempenho da GPU. Demonstraram a melhoria alcançada através de diversas aplicações, em uma delas a utilização da GPU sem as melhorias proposto levou a um *speedup* de até 36,28; enquanto que com as melhorias proporcionaram um *speedup* de até 179,82; em relação ao algoritmo em CPU.

Cheng e Gen (2019) deram uma visão geral de vários tipos de algoritmos genéticos implementados em GPU selecionados desde 2010. Listaram como os três princípios para a otimização de uma implementação em GPU por ordem de importância sendo: exposição da suficiente do paralelismo, otimização do acesso a memória, e otimização da execução das instruções.

Na mecânica estrutural e na geomecânica também se tem avanços com a utilização da GPU. Yang, Yang e Hsieh (2014) utilizaram um programa de análise estrutural já conhecido e acoplaram um módulo que calcula as matrizes de rigidez e vetores de forças dos elementos em GPU, as montagens da matriz e do vetor globais foram feitas sequencialmente na CPU, e obtiveram acelerações de 7,6 vezes em uma simulação com 15048 graus de liberdade e de 8,3 vezes em uma simulação com 23088 graus de liberdade.

Um solucionador estrutural dinâmico utilizando o método explícito em GPU foi desenvolvido por Bartezzaghi et al. (2015) para placas finas. Por se tratar de um método explícito, não há um sistema a ser resolvido e com isso foi utilizado uma estratégia onde não se faz necessária a montagem da matriz global de rigidez. Em uma comparação com um algoritmo em CPU foi obtido uma aceleração de aproximadamente 50 vezes utilizando precisão simples, e aproximadamente 20 vezes com a utilização de precisão dupla.

2.2 Acoplamento Hidromecânico

Segundo Zienkiewicz e Taylor (2000), sistemas acoplados podem ser classificados em duas categorias. Na primeira, o acoplamento ocorre na interface dos domínios dos dois sistemas envolvidos através das condições de contorno. Geralmente cada domínio possui uma física diferente, mas pode ocorrer o acoplamento de domínios com físicas iguais. Já na segunda, os domínios dos sistemas envolvidos se sobrepõem completamente ou parcialmente. Nesta classe o

acoplamento é feito através das equações diferenciais que governam os fenômenos físicos de cada sistema.

O acoplamento pode ser feito de forma sequencial como Minkoff et al. (2003), onde as pressões nos poros resultantes da simulação de fluxo se tornavam o vetor de forças da simulação mecânica, e as deformações vindas da simulação mecânica atualizavam a porosidade do meio, alterando a matriz de fluxo. Isto os permitiu utilizar dois simuladores comerciais diferentes em elementos finitos. Concluíram que a simulação mecânica não é necessária em todos os passos de tempos, mas deve ser feita com certa frequência para não prejudicar os resultados. O mesmo foi feito por Conell (2009) para investigar a drenagem de gás em camadas de carvão. Usando o mesmo tipo de acoplamento Moradi, Shamloo e Dezfuli (2017) desenvolveram códigos próprios tanto para a simulação mecânica quanto para a simulação de fluxo. Analisando assim reservatórios fraturados.

Para um problema de interação fluido estrutura Nilsson e Tornberg (2019) descreveram três métodos de resolução. Um deles consistia no acoplamento de um simulador CFD com um simulador mecânico em MEF de maneira iterativa transferindo cargas e deslocamento entre os simuladores. A cada passo de tempo este processo iterativo foi realizado. Em outro método, ambos os domínios foram simulados em MEF e de maneira completamente acoplada, onde os problemas são resolvidos em um único sistema. Este método possibilita o cálculo de frequências. Utilizaram métodos implícitos de integração no tempo: o método de Newmark com e sem dissipação numérica, e o método de Bathe. Isto para garantir a estabilidade sem depender do passo de tempo.

Um método de volumes finitos para a análise dinâmica de estruturas em resposta a movimentos de fluido é proposto por Xia e Lin (2008). O acoplamento é realizado com as pressões resultantes do domínio fluido entrando como força no domínio mecânico e os deslocamentos vindos do domínio mecânico passam por um tratamento através do método de malha dinâmica e serve de entrada no problema fluido.

Minkoff et al. (1970) descreveu e realizou testes com o acoplamento sequencial unidirecional. Onde as pressões resultantes do problema de fluxo alimentam o problema mecânica, mas o problema mecânico não alimenta o problema de fluxo. Este acoplamento foi utilizado como teste inicial para o acoplamento bidirecional, em que os deslocamentos resultantes do problema mecânico modificam a porosidade do problema de fluxo.

Neste trabalho o acoplamento hidromecânico foi realizado apenas em problemas da segunda categoria. O acoplamento utilizado foi o sequencial unidirecional, onde o problema mecânico não influencia o problema de fluxo.

2.2.1 Simulação computacional de terremotos em barragem

Podem ser utilizados muitos métodos numéricos na simulação dinâmica de uma barragem, os mais comuns são o método dos elementos finitos e o método das diferenças finitas. Utilizando o método dos elementos finitos Chopra e Wang (2009) calcularam os efeitos de um terremoto

registrado em duas barragens incluindo a interação com a água e a fundação. Comparando as respostas computadas com as registradas para identificar as variações de acordo com os parâmetros do terremoto e da barragem.

Com a utilização do método dos elementos finitos através do programa SAP2000, Gutstein (2011) realiza uma análise pseudo-estática para avaliar a estrutura em resposta aos sismos, apesar de considerar esta análise muito conservadora. Com o mesmo método, através do programa ANSYS 11.0, Mendes e Pedroso (2016) simularam computacionalmente um terremoto na barragem de concreto em arco *Morrow Point*, situada nos EUA. Avaliaram em duas situações: a barragem isolada e a barragem interagindo com a água em uma profundidade igual a altura da barragem. Comparando as duas situações, observaram aumentos nos deslocamentos pela interação com a água de até 3.8 vezes.

Com a utilização do programa de elementos finitos FLAC 2D, Loayza (2009) analisou barragens de rejeitos sob a ação de sismos e identificou um tempo de processamento muito longo quando se tratava de grandes deformações. Com o mesmo programa, Fajardo (2015) analisou o comportamento dinâmico da barragem de Breapampa no Peru, no processo foi implementado rotinas na linguagem no próprio programa para tornar a modelagem mais prática. Utilizando o programa Plaxis 2D, Collantes (2015) realizou simulações de terremoto afim de identificar e apresentar métodos de como analisar obras de terra sob ações sísmicas.

Acoplando o problema com a fundação e com água, Mendes (2018) utilizou o ANSYS 14 para a simulação de terremotos em barragens pelo método dos elementos finitos, além de realizar as análises modal e estática em cada caso. A combinação da solicitação dinâmica com a estática gera resultados mais próximos aos casos reais. Santos (2018) analisou uma barragem em arco de concreto através do método dos elementos finitos. Foi utilizado o *software Diana* com o elemento tetraédrico CTE30. Foram realizadas análises estática, modal e sísmica.

A utilização de *softwares* comerciais para a simulação é bastante cômoda e tem a garantia de uma simulação precisa, a depender da discretização utilizada. Alguns destes programas bastante conhecidos possuem vários tipos de elementos e várias formas de aplicação das cargas dinâmicas. A maioria destes já possuem módulos para a utilização de mais de um núcleo da CPU, e alguns já desenvolveram ou estão desenvolvendo módulos para o uso da GPU. Porém a atualização destes módulos para uso de novas tecnologias desenvolvidas em GPU não pode ser realizada assim que lançada essas novas tecnologias, deve-se esperar que o programa seja atualizado, se ele for atualizado. E não terá acesso ao código caso queira modificar algum modo de simulação. A utilização de código próprio pode ser mais complexa, mas em compensação a atualização para novas tecnologias e modificações em simulação podem ser feitas sem espera. E, se tratando de um código aberto, as modificações podem ser realizadas pelo próprio usuário.

2.2.2 Simulação computacional geomecânica em reservatório de petróleo

Um simulador em elementos finitos de reservatórios acoplado com a geomecânica foi desenvolvido por Triana (2017) utilizando técnicas de multi-escala e utilizando uma aproximação

com base reduzida através do ambiente de programação NeoPZ. Comparando este simulador a simuladores comerciais com e sem acoplamento geomecânico, os resultados obtidos com o simulador proposto foram considerados de alta qualidade.

Falcão (2002) comparou duas formas de acoplamento, cada uma utilizando *softwares* diferentes. Na primeira foi utilizado um acoplamento entre o simulador de fluxo IMEX e o simulador geomecânico STARS, ambos da CMG. Já na segunda, foi utilizado o simulador FPORO, desenvolvido pela PUC-Rio, totalmente acoplado, ou seja, resolve as pressões e os deslocamentos em um único sistema. Os simuladores da CMG se mostraram mais completos e algumas melhorias foram propostas para o FPORO.

Atualmente na PUC-Rio está sendo utilizado um acoplamento iterativo entre o *software* comercial IMEX com o programa desenvolvido na universidade, o Chronos. Este programa foi escrito em C++ e CUDA, para a utilização da GPU, e usa paralelismo em CPU com a estrutura OpenMP. Em alternativa a este programa é utilizado o simulador comercial Abaqus. Albuquerque (2015), Seabra (2017), Bizzo (2017) e Righetto (2018) utilizaram este acoplamento para simulações geomecânicas mais recentemente.

O Laboratório de Métodos Computacionais em Geomecânica da UFPE utiliza o programa CODE_BRIGHT, que foi desenvolvido na Universidade Politécnica da Catalunha e possui várias implementações realizadas pelo grupo. Este programa foi desenvolvido em Fortran e tem versões que capazes de utilizar 2, 4 ou 6 núcleos. É capaz de analisar fenômenos 3D termo-hidromecânicos acoplados. Silva (2018), Assis (2019), Rodrigues (2019) e Melo (2019) utilizaram o programa em trabalhos mais recentes. No mesmo grupo Lima (2019) utilizou o *software* IMEX acoplado a um código em MATLAB para comparar métodos de acoplamento. O acoplamento foi feito através no MATLAB, onde este recebia as pressões do simulador IMEX e calculava as deformações através de um código mecânico, utilizando as porosidades como entrada no simulador de fluxo.

Para este tipo de simulação é mais comum o acoplamento entre simuladores de fluxo e mecânicos, onde os simuladores de fluxo são comercias e os mecânicos podem ser comerciais ou não. Para uma mesma malha o problema mecânico é mais custoso que um problema de fluxo, e é necessária a utilização de uma estratégia de aceleração, o que é mais simples com a utilização de código próprio do que com um simulador comercial.

2.3 Integração Direta no Tempo

Existem dois tipos de integração direta no tempo. A integração implícita, onde é calculada uma matriz de rigidez equivalente e um vetor de forças equivalente e é realizado a resolução do sistema. E a integração explícita, onde as acelerações são calculadas em relação aos deslocamentos e velocidades de passos anteriores e então o cálculo dos novos deslocamentos e velocidades em relação as acelerações obtidas. Vários esquemas de integração implícita e explícita foram desenvolvidos e vêm sendo ainda hoje.

Segundo Bathe (1996) o método explícito mais comum utilizado é o método da diferença central, que consiste em aproximar a derivada no tempo utilizando o método das diferenças finitas centrais de forma que o deslocamento atual dependa dos deslocamentos passados. Já o método mais utilizado para integração direta implícita é o de Newmark, que calcula o deslocamento futuro como a soma da velocidade atual com a multiplicação do passo de tempo por uma velocidade média ponderada da atual com a futura e o mesmo é feito para o calculo da velocidade. Os ponderadores da média podem ser escolhidos de modo que gerem resultados estáveis.

Um novo esquema implícito de integração no tempo foi desenvolvido por Wen et al. (2017b). Compararam a eficiência e o consumo de tempo deste novo esquema com os esquemas de Bathe e de α generalizado. E demonstraram que o esquema proposto obteve um desempenho desejável, sendo melhor que o esquemas anteriores.

Wen et al. (2017a) descreveram e realizaram um estudo comparativo com três esquemas implícitos compostos. O esquema de Bathe, onde na primeira etapa é utilizada a regra do trapézio e na segunda o esquema de Euler inverso de três pontos. Um esquema que eles chamaram de TTBDF e o esquema de Wen, ambos baseados no esquema anterior. Apesar do melhor desempenho dos dois últimos, a viabilidade ainda precisa ser melhor estudadas. Enquanto que o esquema de Bathe possui um bom desempenho e é mais viável que os outros dois.

Dois esquemas de integração explícita foram propostos por Zhang e Xing (2019). Estes esquemas se baseiam nos deslocamentos e velocidades, os vetores de aceleração não são armazenados. O primeiro esquema proposto se trata de um esquema de passo único, já o segundo de um esquema de passo duplo. O desempenho foi comparado a outros esquemas. O primeiro esquema possui precisão um pouco abaixo da desejável, enquanto o segundo demonstrou uma maior precisão e uma maior flexibilidade em relação aos métodos existentes.

Para um problema de propagação de ondas, Noh e Bathe (2013) propuseram um esquema implícito que também pode ser utilizado em dinâmica estrutural. Mas, segundo os autores, frequentemente métodos implícitos demonstram serem mais eficazes em problemas estruturais.

Baseado no método da colocação Kim e Lee (2018) desenvolveram um esquema de integração explícita no tempo para análises lineares e não lineares. Para este novo esquema o passo de crítico foi considerado 10% do menor período do sistema.

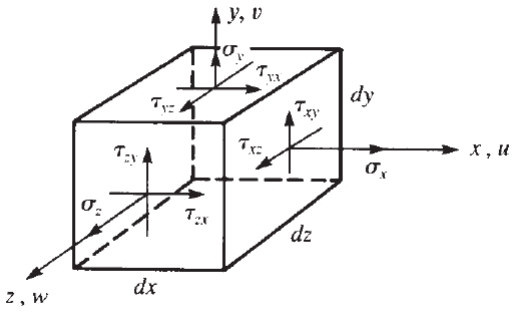
3 FORMULAÇÃO DO PROBLEMA

Neste capítulo são apresentadas as formulações para os problemas de elasticidade mecânica, fluxo em meios porosos e para o acoplamento hidromecânico. Com a utilização do método dos elementos finitos para sua resolução.

3.1 Elasticidade Mecânica

O equilíbrio de tensões em um elemento infinitesimal de um meio contínuo é feito a partir de seu diagrama de corpo livre, Figura 5.

Figura 5 – Diagrama de corpo livre de um elemento infinitesimal.



Fonte: LOGAN, 2012.

Na Equação (3.1) é feito o equilíbrio de forças na direção x:

$$\begin{aligned}\sum F_x &= (\sigma_x + \frac{\partial \sigma_x}{\partial x}dx)dy\,dz - \sigma_x dy\,dz + (\tau_{xy} + \frac{\partial \tau_{xy}}{\partial y}dy)dx\,dz - \tau_{xy}dx\,dz + \\ &\quad (\tau_{zx} + \frac{\partial \tau_{zx}}{\partial z}dz)dx\,dy - \tau_{zx}dx\,dy + f_x\,dx\,dy\,dz + f_{t,x}\,dx\,dy\,dz = 0 \qquad (3.1)\end{aligned}$$

onde tem-se:

- σ_x, σ_y **e** σ_z , as tensões normais nos planos;

- τ_{xy}, τ_{yz} **e** τ_{zx} , as tensões cisalhantes nos planos;

- f_x, f_y **e** f_z , as forças de corpo por unidade de volume;

- $f_{t,x}$, $f_{t,y}$ **e** $f_{t,z}$, as forças inerciais de corpo por unidade de volume.

Simplificando (3.1) obtém-se a Equação (3.2). E analogamente as Equações (3.3) e (3.4) para as direções y e z respectivamente. Tem-se assim as Equações diferenciais do equilíbrio.

$$\frac{\partial \sigma_x}{\partial x} + \frac{\partial \tau_{xy}}{\partial y} + \frac{\partial \tau_{zx}}{\partial z} + f_x + f_{t,x} = 0 \tag{3.2}$$

$$\frac{\partial \sigma_y}{\partial y} + \frac{\partial \tau_{xy}}{\partial x} + \frac{\partial \tau_{yz}}{\partial z} + f_y + f_{t,y} = 0 \tag{3.3}$$

$$\frac{\partial \sigma_z}{\partial z} + \frac{\partial \tau_{zx}}{\partial x} + \frac{\partial \tau_{yz}}{\partial y} + f_z + f_{t,z} = 0 \tag{3.4}$$

As forças inerciais $f_{t,e}$ são:

$$f_{t,e} = -a_e\,\gamma = -\frac{\partial^2 u_e}{\partial t^2}\,\gamma \tag{3.5}$$

onde γ é o peso específico do domínio, a_e e u_e são a aceleração e a deformação na direção e respectivamente. O sinal negativo se dá pelo sentido oposto à aceleração.

As Equações (3.2), (3.3) e (3.4) formam um sistema que pode ser posto na forma matricial:

$$\begin{bmatrix} \frac{\partial}{\partial x} & 0 & 0 & \frac{\partial}{\partial y} & 0 & \frac{\partial}{\partial z} \\ 0 & \frac{\partial}{\partial y} & 0 & \frac{\partial}{\partial x} & \frac{\partial}{\partial z} & 0 \\ 0 & 0 & \frac{\partial}{\partial z} & 0 & \frac{\partial}{\partial y} & \frac{\partial}{\partial x} \end{bmatrix} \begin{pmatrix} \sigma_x \\ \sigma_y \\ \sigma_z \\ \tau_{xy} \\ \tau_{yz} \\ \tau_{zx} \end{pmatrix} + \begin{pmatrix} f_x \\ f_y \\ f_z \end{pmatrix} + \begin{pmatrix} f_{t,x} \\ f_{t,y} \\ f_{t,z} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \tag{3.6}$$

Uma forma mais sucinta para a Equação (3.6) é a Equação (3.7), onde a matriz $\boldsymbol{A}$ é o operador sobre o vetor de tensões $\boldsymbol{\sigma}$.

$$\boldsymbol{A}\,\boldsymbol{\sigma} + \boldsymbol{f} + \boldsymbol{f_t} = \boldsymbol{0} \tag{3.7}$$

Segundo Vilaça e Garcia (1998) as Equações constitutivas caracterizam o comportamento do material, pois relacionam as componentes de tensão com as componentes de deformação. Estas relações estão descritas abaixo.

$$\begin{aligned} \epsilon_x &= \tfrac{\sigma_x}{E} + \tfrac{\nu}{E}(\sigma_y + \sigma_z) \qquad & \gamma_{xy} &= \tfrac{\tau_{xy}}{G} \\ \epsilon_y &= \tfrac{\sigma_y}{E} + \tfrac{\nu}{E}(\sigma_z + \sigma_x) \qquad & \gamma_{yz} &= \tfrac{\tau_{yz}}{G} \\ \epsilon_z &= \tfrac{\sigma_z}{E} + \tfrac{\nu}{E}(\sigma_x + \sigma_y) \qquad & \gamma_{zx} &= \tfrac{\tau_{zx}}{G} \end{aligned} \tag{3.8}$$

sendo:

- ϵ_x, ϵ_y **e** ϵ_z , as deformações axiais;

- γ_{xy}, γ_{yz} **e** γ_{zx} , as distorções angulares;

- E , o módulo de elasticidade ou módulo de Young;

- ν , o coeficiente de Poisson;

- G , o módulo de elasticidade transversal que pode ser calculado por:

$$G = \frac{E}{2(1+\nu)}$$

O sistema formado pelas Equações das relações constitutivas pode ser colocado em forma matricial como em (3.9).

$$\begin{pmatrix} \epsilon_x \\ \epsilon_y \\ \epsilon_z \\ \gamma_{xy} \\ \gamma_{yz} \\ \gamma_{zx} \end{pmatrix} = \frac{1}{E} \begin{bmatrix} 1 & -\nu & -\nu & 0 & 0 & 0 \\ -\nu & 1 & -\nu & 0 & 0 & 0 \\ -\nu & -\nu & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2(1+\nu) & 0 & 0 \\ 0 & 0 & 0 & 0 & 2(1+\nu) & 0 \\ 0 & 0 & 0 & 0 & 0 & 2(1+\nu) \end{bmatrix} \begin{pmatrix} \sigma_x \\ \sigma_y \\ \sigma_z \\ \tau_{xy} \\ \tau_{yz} \\ \tau_{zx} \end{pmatrix} \tag{3.9}$$

Modificando (3.9) para se ter as tensões em função das deformações:

$$\begin{pmatrix} \sigma_x \\ \sigma_y \\ \sigma_z \\ \tau_{xy} \\ \tau_{yz} \\ \tau_{zx} \end{pmatrix} = \frac{E}{(1-2\nu)(1+\nu)} \begin{bmatrix} 1-\nu & -\nu & -\nu & 0 & 0 & 0 \\ -\nu & 1-\nu & -\nu & 0 & 0 & 0 \\ -\nu & -\nu & 1-\nu & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1-2\nu}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1-2\nu}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1-2\nu}{2} \end{bmatrix} \begin{pmatrix} \epsilon_x \\ \epsilon_y \\ \epsilon_z \\ \gamma_{xy} \\ \gamma_{yz} \\ \gamma_{zx} \end{pmatrix} \tag{3.10}$$

sendo $\boldsymbol{D}$ a matriz de elasticidade do material, a Equação (3.10) pode ser escrita de forma mais compacta:

$$\boldsymbol{\sigma} = \boldsymbol{D}\boldsymbol{\epsilon} \tag{3.11}$$

Além das Equações de equilíbrio e das Equações constitutivas é necessário utilizar Equações que relacionam deformações e deslocamentos, relações cinéticas. Estas Equações traduzem relações de natureza geométrica e para estabelece-las deve-se admitir a hipótese de pequenas mudanças de configurações (VILAÇA; GARCIA, 1998). Segundo a qual as deformações e rotações são consideradas muito menores que a unidade em questão. Estas relações estão descritas a seguir.

$$\begin{aligned} \epsilon_x &= \frac{\partial u}{\partial x} \qquad & \gamma_{xy} &= \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \\ \epsilon_y &= \frac{\partial v}{\partial y} \qquad & \gamma_{yz} &= \frac{\partial v}{\partial z} + \frac{\partial w}{\partial y} \\ \epsilon_z &= \frac{\partial w}{\partial z} \qquad & \gamma_{zx} &= \frac{\partial w}{\partial x} + \frac{\partial u}{\partial z} \end{aligned} \tag{3.12}$$

onde u, v e w são os deslocamentos nos eixos x, y e z respectivamente. Passando o sistema formado por (3.12) para forma matricial:

$$\begin{pmatrix} \epsilon_x \\ \epsilon_y \\ \epsilon_z \\ \gamma_{xy} \\ \gamma_{yz} \\ \gamma_{zx} \end{pmatrix} = \begin{bmatrix} \frac{\partial}{\partial x} & 0 & 0 \\ 0 & \frac{\partial}{\partial y} & 0 \\ 0 & 0 & \frac{\partial}{\partial z} \\ \frac{\partial}{\partial y} & \frac{\partial}{\partial x} & 0 \\ 0 & \frac{\partial}{\partial z} & \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} & 0 & \frac{\partial}{\partial x} \end{bmatrix} \begin{pmatrix} u \\ v \\ w \end{pmatrix} \tag{3.13}$$

Visto que o operador sobre vetor de deslocamentos em (3.13) é a transposta do operador sobre o vetor de tensões em (3.6), pode-se reescrever (3.13):

$$\boldsymbol{\epsilon} = \boldsymbol{A}^T \boldsymbol{u} \tag{3.14}$$

As Equações (3.7), (3.11) e (3.14) devem ser satisfeitas no domínio do problema de elasticidade. Já as Equações ditas condições de contorno devem ser satisfeitas no contorno do domínio em que são aplicadas. Segundo Kwon e Bang (1996) as condições de contorno podem ser deslocamentos prescritos (condições essenciais) ou forças prescritas (condições naturais), esta última pode ser expressa da forma:

$$\begin{aligned} q_x &= \sigma_x n_x + \tau_{xy} n_y + \tau_{zx} n_z = \overline{q}_x \\ q_y &= \sigma_y n_y + \tau_{xy} n_x + \tau_{yz} n_z = \overline{q}_y \\ q_z &= \sigma_z n_z + \tau_{yz} n_y + \tau_{zx} n_x = \overline{q}_z \end{aligned} \tag{3.15}$$

onde tem-se:

- n_x**,** n_y **e** n_z , os cossenos diretores nas direções x, y e z respectivamente;

- $\overline{q}_x$**,** $\overline{q}_y$ **e** $\overline{q}_z$, as tensões prescritas nas direções x, y e z respectivamente.

3.1.1 Formulação MEF no Problema Mecânico

Aplicando o método dos resíduos ponderados nas Equações de equilíbrio e condições de contorno:

$$\int_V \begin{pmatrix} \omega_1(\frac{\partial \sigma_x}{\partial x} + \frac{\partial \tau_{xy}}{\partial y} + \frac{\partial \tau_{zx}}{\partial z} + f_x + f_{t,x}) \\ \omega_2(\frac{\partial \sigma_y}{\partial y} + \frac{\partial \tau_{xy}}{\partial x} + \frac{\partial \tau_{yz}}{\partial z} + f_y + f_{t,y}) \\ \omega_3(\frac{\partial \sigma_z}{\partial z} + \frac{\partial \tau_{yz}}{\partial y} + \frac{\partial \tau_{zx}}{\partial x} + f_z + f_{t,z}) \end{pmatrix} dV + \int_A \begin{pmatrix} \omega_1(\overline{q}_x - q_x) \\ \omega_2(\overline{q}_y - q_y) \\ \omega_3(\overline{q}_z - q_z) \end{pmatrix} dA = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \tag{3.16}$$

sendo:

- ω_1**,** ω_2 **e** ω_3 , as funções de ponderação nas direções x, y e z respectivamente;

- V , o domínio do problema;

- A , a superfície do domínio onde a condição de contorno é aplicada.

A Equação (3.16 é chamada de forma forte do problema, pois com a substituição das Equações (3.8 e (3.12 são formados termos de derivada de alta ordem que obrigam que a função de aproximação tenha uma ordem elevada. Para possibilitar que a função de aproximação tenha uma ordem mais baixa deve-se passar a Equação para a forma fraca, eliminando o termo de derivada de alta ordem. Resolvendo alguns termos da primeira integral da Equação (3.16 por partes para o uso da forma fraca, temos:

$$\int_V \begin{pmatrix} \omega_1(\frac{\partial \sigma_x}{\partial x} + \frac{\partial \tau_{xy}}{\partial y} + \frac{\partial \tau_{zx}}{\partial z}) \\ \omega_2(\frac{\partial \sigma_y}{\partial y} + \frac{\partial \tau_{xy}}{\partial x} + \frac{\partial \tau_{yz}}{\partial z}) \\ \omega_3(\frac{\partial \sigma_z}{\partial z} + \frac{\partial \tau_{yz}}{\partial y} + \frac{\partial \tau_{zx}}{\partial x}) \end{pmatrix} dV = \int_V \begin{pmatrix} \frac{\partial(\sigma_x \omega_1)}{\partial x} + \frac{\partial(\tau_{xy}\omega_1)}{\partial y} + \frac{\partial(\tau_{zx}\omega_1)}{\partial z} \\ \frac{\partial(\sigma_y \omega_2)}{\partial y} + \frac{\partial(\tau_{xy}\omega_2)}{\partial x} + \frac{\partial(\tau_{yz}\omega_2)}{\partial z} \\ \frac{\partial(\sigma_z \omega_3)}{\partial z} + \frac{\partial(\tau_{yz}\omega_3)}{\partial y} + \frac{\partial(\tau_{zx}\omega_3)}{\partial x} \end{pmatrix} dV - \int_V \begin{pmatrix} \frac{\partial \omega_1}{\partial x}\sigma_x + \frac{\partial \omega_1}{\partial y}\tau_{xy} + \frac{\partial \omega_1}{\partial z}\tau_{zx} \\ \frac{\partial \omega_2}{\partial y}\sigma_y + \frac{\partial \omega_2}{\partial x}\tau_{xy} + \frac{\partial \omega_2}{\partial z}\tau_{yz} \\ \frac{\partial \omega_3}{\partial z}\sigma_z + \frac{\partial \omega_3}{\partial y}\tau_{yz} + \frac{\partial \omega_3}{\partial x}\tau_{zx} \end{pmatrix} dV \tag{3.17}$$

Aplicando o Teorema de Stokes na segunda integral da Equação (3.17:

$$\int_V \begin{pmatrix} \frac{\partial(\sigma_x \omega_1)}{\partial x} + \frac{\partial(\tau_{xy}\omega_1)}{\partial y} + \frac{\partial(\tau_{zx}\omega_1)}{\partial z} \\ \frac{\partial(\sigma_y \omega_2)}{\partial y} + \frac{\partial(\tau_{xy}\omega_2)}{\partial x} + \frac{\partial(\tau_{yz}\omega_2)}{\partial z} \\ \frac{\partial(\sigma_z \omega_3)}{\partial z} + \frac{\partial(\tau_{yz}\omega_3)}{\partial y} + \frac{\partial(\tau_{zx}\omega_3)}{\partial x} \end{pmatrix} dV = \int_A \begin{pmatrix} \sigma_x\omega_1 n_x + \tau_{xy}\omega_1 n_y + \tau_{zx}\omega_1 n_z \\ \sigma_y\omega_2 n_y + \tau_{xy}\omega_2 n_x + \tau_{yz}\omega_2 n_z \\ \sigma_z\omega_3 n_z + \tau_{yz}\omega_3 n_y + \tau_{zx}\omega_3 n_x \end{pmatrix} dA \tag{3.18}$$

Substituindo (3.18 em (3.17 e o resultado em (3.16:

$$-\int_V \begin{pmatrix} \frac{\partial \omega_1}{\partial x}\sigma_x + \frac{\partial \omega_1}{\partial y}\tau_{xy} + \frac{\partial \omega_1}{\partial z}\tau_{zx} \\ \frac{\partial \omega_2}{\partial y}\sigma_y + \frac{\partial \omega_2}{\partial x}\tau_{xy} + \frac{\partial \omega_2}{\partial z}\tau_{yz} \\ \frac{\partial \omega_3}{\partial z}\sigma_z + \frac{\partial \omega_3}{\partial y}\tau_{yz} + \frac{\partial \omega_3}{\partial x}\tau_{zx} \end{pmatrix} dV + \int_V \begin{pmatrix} \omega_1 f_x \\ \omega_2 f_y \\ \omega_3 f_z \end{pmatrix} dV + \int_V \begin{pmatrix} \omega_1 f_{t,x} \\ \omega_2 f_{t,y} \\ \omega_3 f_{t,z} \end{pmatrix} dV + \int_A \begin{pmatrix} \omega_1 \overline{q}_x \\ \omega_2 \overline{q}_y \\ \omega_3 \overline{q}_z \end{pmatrix} dA = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \tag{3.19}$$

A Equação (3.19) é a forma fraca do problema. Colocando-a de forma matricial:

$$-\int_V (\boldsymbol{A}^T \boldsymbol{W})^T \boldsymbol{\sigma} dV + \int_V \boldsymbol{W}^T \boldsymbol{f} dV + \int_V \boldsymbol{W}^T \boldsymbol{f_t} dV + \int_A \boldsymbol{W}^T \overline{\boldsymbol{q}} dA = \boldsymbol{0} \tag{3.20}$$

Onde:

$$\boldsymbol{W} = \begin{bmatrix} \omega_1 & 0 & 0 \\ 0 & \omega_2 & 0 \\ 0 & 0 & \omega_3 \end{bmatrix}$$

$$\overline{\boldsymbol{q}} = \begin{pmatrix} \overline{q}_x \\ \overline{q}_y \\ \overline{q}_z \end{pmatrix}$$

Substituindo (3.14), (3.11) e (3.5) em (3.20):

$$-\int_V (\boldsymbol{A}^T\boldsymbol{W})^T\boldsymbol{D}\boldsymbol{A}^T\boldsymbol{u}dV + \int_V \boldsymbol{W}^T\boldsymbol{f}dV - \int_V \boldsymbol{W}^T\frac{\boldsymbol{\partial^2 u}}{\boldsymbol{\partial t^2}}\gamma dV + \int_A \boldsymbol{W}^T\overline{\boldsymbol{q}}dA = \boldsymbol{0} \quad (3.21)$$

Discretizando o domínio em elementos onde os deslocamentos u, v e w dentro de cada elemento são aproximadas em função dos deslocamentos nos nós do mesmo. Sendo H_i, as funções de forma que compõem as funções aproximadas, e n o número de nós em cada elemento:

$$\tilde{u}(x,y,z) = \sum_{i=0}^{n} H_i(x,y,z)u_i \quad (3.22)$$

$$\tilde{v}(x,y,z) = \sum_{i=0}^{n} H_i(x,y,z)v_i \quad (3.23)$$

$$\tilde{w}(x,y,z) = \sum_{i=0}^{n} H_i(x,y,z)w_i \quad (3.24)$$

Segundo Kwon e Bang (1996) no método de Galerkin as funções de ponderação vêm da função aproximada escolhida, da forma:

$$\omega_{1,i} = \frac{\partial\tilde{u}}{\partial u_i}; \quad \omega_{2,i} = \frac{\partial\tilde{v}}{\partial v_i}; \quad \omega_{3,i} = \frac{\partial\tilde{w}}{\partial w_i}. \quad (3.25)$$

O sistema formado pelas Equações (3.22), (3.23) e (3.24) pode ser colocado na forma matricial:

$$\boldsymbol{u} = \boldsymbol{H}\boldsymbol{d} \quad (3.26)$$

onde:

$$\boldsymbol{H} = \left[\begin{array}{cccccc} \boldsymbol{W}_1 & ... & \boldsymbol{W}_i & ... & \boldsymbol{W}_n \end{array} \right];$$

$$\boldsymbol{W}_i = \left[\begin{array}{ccc} \omega_{1,i} & 0 & 0 \\ 0 & \omega_{2,i} & 0 \\ 0 & 0 & \omega_{3,i} \end{array} \right];$$

$$\boldsymbol{d} = \left[\begin{array}{ccccccccccc} u_1 & v_1 & w_1 & ... & u_i & v_i & w_i ... & u_n & v_n & w_n \end{array} \right]^T.$$

Modificando (3.21) para o domínio discretizado de um elemento e inserindo (3.26):

$$-\int_e (\boldsymbol{A}^T\boldsymbol{H})^T\boldsymbol{D}\boldsymbol{A}^T\boldsymbol{H}\boldsymbol{d}\,dV + \int_e \boldsymbol{H}^T\boldsymbol{f}dV - \int_V \boldsymbol{H}^T\boldsymbol{H}\frac{\boldsymbol{\partial^2 d}}{\boldsymbol{\partial t^2}}\gamma dV + \int_e \boldsymbol{H}^T\overline{\boldsymbol{q}}dA = \boldsymbol{0} \quad (3.27)$$

sendo $\boldsymbol{B} = \boldsymbol{A}^T\boldsymbol{H}$, $\boldsymbol{K}_e$ a matriz de rigidez do elemento, $\boldsymbol{M}_e$ a matriz de massa do elemento, $\boldsymbol{F}_{c,e}$ o vetor de forças de corpo do elemento e $\boldsymbol{F}_{q,e}$ o vetor de forças externas do elemento:

$$-\overbrace{\int_e \boldsymbol{B}^T\boldsymbol{D}\boldsymbol{B}\,dV}^{\boldsymbol{K}_e}\boldsymbol{d} + \overbrace{\int_e \boldsymbol{H}^T\boldsymbol{f}dV}^{\boldsymbol{F}_{c,e}} - \overbrace{\gamma\int_e \boldsymbol{H}^T\boldsymbol{H}\,dV}^{\boldsymbol{M}_e}\frac{\boldsymbol{\partial^2 d}}{\boldsymbol{\partial t^2}} + \overbrace{\int_e \boldsymbol{H}^T\overline{\boldsymbol{q}}dA}^{\boldsymbol{F}_{q,e}} = \boldsymbol{0} \quad (3.28)$$

Ou seja, pra cada elemento temos:

$$\boldsymbol{K}_e\boldsymbol{d} + \boldsymbol{M}_e\frac{\boldsymbol{\partial^2 d}}{\boldsymbol{\partial t^2}} = \boldsymbol{F}_{c,e} + \boldsymbol{F}_{q,e} \quad (3.29)$$

No caso particular do elemento utilizado ser tetraédrico e de tensão constante, tem-se a matriz $\boldsymbol{B}$ constante, e considerando que as propriedades do material não se alteram dentro do elemento, a matriz $\boldsymbol{D}$ também será constante, a matriz de rigidez do elemento será:

$$\boldsymbol{K}_e = \boldsymbol{B}^T\boldsymbol{D}\boldsymbol{B}\,V_e \quad (3.30)$$

em que V_e é o volume do elemento.

Em outro caso, análogo ao anterior, em que temos um problema de elasticidade em duas dimensões e o domínio foi discretizado em elementos triangulares de tensão constante (CST) a formulação segue o mesmo padrão para se encontrar a matriz $\boldsymbol{B}$, de forma que a matriz de rigidez do elemento será:

$$\boldsymbol{K}_e = \boldsymbol{B}^T\boldsymbol{D}\boldsymbol{B}\,A_e t \quad (3.31)$$

onde temos a área A_e do elemento e sua espessura t. Já a matriz $\boldsymbol{D}$, caso esteja-se admitindo o estado plano de tensões, será:

$$\boldsymbol{D} = \frac{E}{1-\nu^2}\left[\begin{array}{ccc} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{array} \right] \quad (3.32)$$

E caso esteja-se admitindo o estado plano de deformações, será:

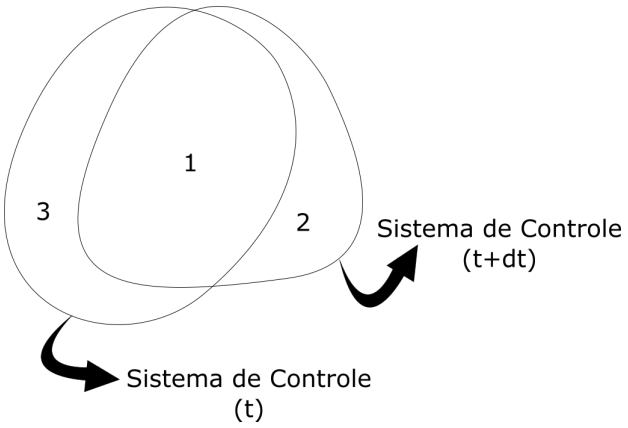
$$\boldsymbol{D} = \frac{E}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1-\nu & \nu & 0 \\ \nu & 1-\nu & 0 \\ 0 & 0 & \frac{1-2\nu}{2} \end{bmatrix} \tag{3.33}$$

3.2 Problema de Fluxo em Meios Porosos

Utilizando a abordagem Euleriana, o problema de escoamento em meio poroso é desenvolvido através do balanço de massa em um volume de controle, onde podemos encontrar a Equação da conservação de massa. Conforme a Figura 6 temos o Volume 1 o de controle, o balanço de massa no tempo é descrito pela equação:

$$M_1(t+\Delta t) + M_2(t+\Delta t) - M_1(t) - M_3(t) = \Delta M_{sistema}. \tag{3.34}$$

Figura 6 – Volume de controle



Fonte: O Autor, 2020.

onde temos que M_i; com $i = 1$, 2 e 3; sendo a massa dos volumes 1, 2 e 3. Dividindo a Equação (3.34) por Δt, estando o Volume 2 representando a massa que sai do volume de controle e o Volume 3 a massa prestes a entrar:

$$\underbrace{\frac{M_1(t+\Delta t) - M_1(t)}{\Delta t}}_{\frac{\Delta M_{vc}}{\Delta t}} - \underbrace{\frac{M_3(t)}{\Delta t}}_{\dot{M}_{entra}} + \underbrace{\frac{M_2(t+\Delta t)}{\Delta t}}_{\dot{M}_{sai}} = \frac{\Delta M_{sistema}}{\Delta t}. \tag{3.35}$$

Podendo existir alguma recarga de massa ou um sumidouro no sistema, temos:

$$\frac{\Delta M_{sistema}}{\Delta t} = \int_V \rho r dV. \tag{3.36}$$

onde temos que ρ é a densidade do fluido e r o termo de recarga ou sumidouro. Os fluxos de entrada e saída do volume de controle podem ser escritos da forma:

$$\dot{M}_{entra} - \dot{M}_{sai} = -\int_A \rho \boldsymbol{V}_r . \boldsymbol{n} dA. \tag{3.37}$$

onde temos que $\boldsymbol{V}_r$ é a velocidade do fluido em relação ao volume de controle e $\boldsymbol{n}$ o vetor normal a superfície do volume de controle. Aplicando o teorema da divergência a esta Equação:

$$-\int_A \rho \boldsymbol{V}_r \boldsymbol{n} dA = -\int_V \nabla(\rho \boldsymbol{V}_r) dV. \tag{3.38}$$

Além disto temos que:

$$\frac{\Delta M_{vc}}{\Delta t} = \int_V \frac{\partial(\phi\rho)}{\partial t} dV. \tag{3.39}$$

onde ϕ é a porosidade do meio sólido, que se encontra completamente saturado, no qual o escoamento. Substituindo as Equações (3.36), (3.37), (3.38) e (3.39) na Equação (3.35) temos a Equação de conservação de massa:

$$\int_V \frac{\partial(\phi\rho)}{\partial t} dV + \int_V \nabla(\rho \boldsymbol{V}_r) dV = \int_V \rho r dV. \tag{3.40}$$

E sua forma diferencial:

$$\frac{\partial(\phi\rho)}{\partial t} + \nabla(\rho \boldsymbol{V}_r) = \rho r. \tag{3.41}$$

A velocidade do fluido em relação ao volume de controle pode ser expressa como:

$$\boldsymbol{V}_r = \boldsymbol{V}_f + \boldsymbol{V}_{vc}. \tag{3.42}$$

sendo $\boldsymbol{V}_f$ a velocidade do fluido em relação a um referencial, e $\boldsymbol{V}_{vc}$ a velocidade do volume de controle a este mesmo referencial. A velocidade do fluido em um meio poroso completamente saturado pode ser obtida pela lei de Darcy (BEAR, 1972):

$$\boldsymbol{V}_f = -\frac{\boldsymbol{\kappa}}{\mu}(\nabla p - \rho \boldsymbol{g}); \tag{3.43}$$

$$\boldsymbol{\kappa} = \begin{bmatrix} \kappa_x & 0 & 0 \\ 0 & \kappa_y & 0 \\ 0 & 0 & \kappa_z \end{bmatrix}. \tag{3.44}$$

onde $\boldsymbol{\kappa}$ é a matriz de permeabilidade intrínseca do meio, μ é a viscosidade do fluido, p a pressão do fluido e $\boldsymbol{g}$ o vetor de gravidade. Já a velocidade do volume de controle é considerada como a velocidade com que os poros se deformam e pode ser escrita como:

$$\boldsymbol{V}_{vc} = \phi \frac{\partial \boldsymbol{u}}{\partial t}. \tag{3.45}$$

Desconsiderando o efeito desta última parcela e substituindo as Equações (3.42) e (3.43 n)a Equação (3.41:

$$\frac{\partial(\phi\rho)}{\partial t} + \nabla(\rho(-\frac{\boldsymbol{\kappa}}{\mu}(\nabla p - \rho\boldsymbol{g}))) = \rho r. \tag{3.46}$$

A densidade do fluido pode se relacionar com a pressão da forma:

$$\rho = \rho_0 e^{c_f(p-p_0)}, \tag{3.47}$$

onde ρ_0 é a densidade média do fluido.

Considerando que a densidade do fluido pouco se altera, ou seja, que a densidade pode ser considerada igual a densidade média, a compressibilidade do fluido c_f pode ser definida como:

$$c_f = \frac{1}{\rho}\frac{\partial\rho}{\partial p}. \tag{3.48}$$

Admitindo que a porosidade ϕ tem uma variação muito pequena com o tempo, que o fluido é pouco compressível e introduzindo a Equação (3.48) na Equação (3.46):

$$\phi\rho c_f\frac{\partial p}{\partial t} + \rho\nabla((-\frac{\boldsymbol{\kappa}}{\mu}(\nabla p - \rho\boldsymbol{g}))) = \rho r. \tag{3.49}$$

Dividindo a Equação (3.49) encontramos a Equação a ser resolvida em função das pressões do fluido:

$$\phi c_f\frac{\partial p}{\partial t} + \nabla((-\frac{\boldsymbol{\kappa}}{\mu}(\nabla p - \rho\boldsymbol{g}))) = r. \tag{3.50}$$

Como condição de contorno do problema de fluxo podemos ter pressões prescritas (condições essenciais) e fluxos prescritos (condições naturais). As condições naturais se apresentam como:

$$q = \boldsymbol{V}_f.\boldsymbol{n} = \overline{q}. \tag{3.51}$$

onde q é o fluxo real e $\overline{q}$ é o fluxo prescrito.

3.2.1 Formulação MEF no Problema de Fluxo

Aplicando o método dos resíduos ponderados, assim como no problema mecânico, nas Equações (3.50) e (3.51):

$$\int_V \omega\phi c_f\frac{\partial p}{\partial t}dV + \int_V \omega\nabla((-\frac{\boldsymbol{\kappa}}{\mu}(\nabla p - \rho\boldsymbol{g})))dV + \int_A \omega(\overline{q} - q)dA = \int_V \omega r dV. \tag{3.52}$$

onde ω é a função de ponderação. Reduzindo a derivada da segunda integral da Equação acima para passar para a forma fraca:

$$\int_V \omega\nabla((-\frac{\boldsymbol{\kappa}}{\mu}(\nabla p-\rho\boldsymbol{g})))dV = \int_V \nabla(\omega(-\frac{\boldsymbol{\kappa}}{\mu}(\nabla p-\rho\boldsymbol{g})))dV+\int_V \nabla\omega(\frac{\boldsymbol{\kappa}}{\mu}(\nabla p-\rho\boldsymbol{g}))dV. \quad (3.53)$$

Pelo teorema da divergência temos que:

$$\int_V \nabla(\omega(-\frac{\boldsymbol{\kappa}}{\mu}(\nabla p-\rho\boldsymbol{g})))dV = \int_A \omega(-\frac{\boldsymbol{\kappa}}{\mu}(\nabla p-\rho\boldsymbol{g}))\boldsymbol{n}dA. \quad (3.54)$$

Ainda temos que:

$$\int_A \omega(-\frac{\boldsymbol{\kappa}}{\mu}(\nabla p-\rho\boldsymbol{g}))\boldsymbol{n}dA = \int_A \omega\boldsymbol{V}_f\boldsymbol{n}dA = \int_A \omega q dA. \quad (3.55)$$

Substituindo (3.53), (3.54) e (3.55) em (3.52) obtemos a forma fraca do problema de fluxo:

$$\int_V \omega\phi c_f\frac{\partial p}{\partial t}dV + \int_V \nabla\omega(\frac{\boldsymbol{\kappa}}{\mu}(\nabla p-\rho\boldsymbol{g}))dV - \int_A \omega\overline{q}dA = \int_V \omega r dV. \quad (3.56)$$

Discretizando o domínio em elementos onde as pressões p dentro de cada elemento são aproximadas em função das pressões nos nós do mesmo. Sendo H_i, as funções de forma associadas a cada nó i que compõem a função aproximada, e n o número de nós em cada elemento:

$$\tilde{p}(x,y,z) = \sum_{i=1}^{n} p_i H_i(x,y,z). \quad (3.57)$$

Da mesma forma que se procedeu para o problema mecânico, no método de Galerkin as funções de ponderação vêm da função aproximada escolhida, logo:

$$\omega_i = \frac{\partial\tilde{p}}{\partial p_i} = H_i. \quad (3.58)$$

colocando a Equação (3.57) na forma matricial:

$$\tilde{p} = \boldsymbol{H}\boldsymbol{p}. \quad (3.59)$$

onde:

$$\boldsymbol{H} = \begin{bmatrix} H_1 & ... & H_i & ... & H_n \end{bmatrix}; \quad (3.60)$$

$$\boldsymbol{p}^T = \begin{bmatrix} p_1 & ... & p_i & ... & p_n \end{bmatrix}. \quad (3.61)$$

Substituindo (3.58), (3.59) e $\boldsymbol{B} = \nabla \boldsymbol{H}$em (3.56):

$$\overbrace{\int_V \boldsymbol{H}^T \phi c_f \boldsymbol{H} dV}^{\boldsymbol{C}_f^e} \frac{\partial \boldsymbol{p}}{\partial t} + \overbrace{\int_V \boldsymbol{B}^T \frac{\boldsymbol{\kappa}}{\mu} \boldsymbol{B} dV}^{\boldsymbol{K}_f^e} \boldsymbol{p} = \overbrace{\int_V \boldsymbol{H}^T r dV + \int_V \boldsymbol{B}^T \frac{\boldsymbol{\kappa}}{\mu} \rho \boldsymbol{g} dV + \int_A \boldsymbol{H}^T \overline{q} dA}^{\boldsymbol{F}_f^e}. \tag{3.62}$$

Mudando a notação de $\frac{\partial \boldsymbol{p}}{\partial t}$ para $\dot{\boldsymbol{p}}$:

$$\boldsymbol{K}_f^e \boldsymbol{p} + \boldsymbol{C}_f^e \dot{\boldsymbol{p}} = \boldsymbol{F}_f^e \tag{3.63}$$

Utilizando o elemento tetraédrico de quatro nós a matriz $\boldsymbol{B}$ será composta de elementos constantes de forma que a matriz $\boldsymbol{K}_f^e$ será:

$$\boldsymbol{K}_f^e = \boldsymbol{B}^T \frac{\boldsymbol{\kappa}}{\mu} \boldsymbol{B} V_e. \tag{3.64}$$

onde V_e é o volume do elemento.

3.3 Acoplamento Hidromecânico

O conceito de tensão efetiva introduzido por Terzaghi em 1923 diz que em um meio poroso saturado com um fluido ao sofrer atuação de um carregamento externo, parte deste carregamento será suportado pelo meio sólido e parte será suportado pelo meio fluido. Sendo $\boldsymbol{\sigma}'$ a tensão efetiva,$\boldsymbol{\sigma}$ as tensões totais, $p\boldsymbol{I}$ as pressões do fluido nos poros e α o coeficiente de Biot:

$$\boldsymbol{\sigma}' = \boldsymbol{\sigma} + \alpha p \boldsymbol{L}. \tag{3.65}$$

$$\boldsymbol{L} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \end{bmatrix}^T \tag{3.66}$$

onde $\alpha = 1$ no caso dos grãos do meio sólido serem incompressíveis. Substituindo a Equação (3.65) na Equação (3.7):

$$\boldsymbol{A}\,\boldsymbol{\sigma}' - \boldsymbol{A}\alpha p \boldsymbol{L} + \boldsymbol{f} + \boldsymbol{f_t} = \boldsymbol{0} \tag{3.67}$$

onde o termo $-\boldsymbol{A}\alpha p \boldsymbol{L}$ pode ser incluso no vetor de forças $\boldsymbol{f}$. O método dos elementos finitos para o caso mecânico será aplicado utilizando as tensões totais e posteriormente será realizada a separação. O acoplamento sequencial será da forma unidirecional (*One-Way*), onde o problema de fluxo gera pressões que entram como ações no problema mecânico (Equação (3.67)) de mesmo passo de tempo.

3.3.1 Formulação MEF no Acoplamento Hidromecânico

Para o acoplamento hidromecânico, a primeira integral da Equação (3.20) será substituída por:

$$-\int_V (\boldsymbol{A}^T\boldsymbol{W})^T\boldsymbol{\sigma}dV = -\int_V (\boldsymbol{A}^T\boldsymbol{W})^T\boldsymbol{\sigma}'dV + \int_V (\boldsymbol{A}^T\boldsymbol{W})^T\alpha p\boldsymbol{L}dV. \tag{3.68}$$

E, prosseguindo as operações como feito anteriormente até chegar na Equação (3.28), temos:

$$-\overbrace{\int_e \boldsymbol{B}^T\boldsymbol{D}\boldsymbol{B}\,dV}^{\boldsymbol{K}_e}\boldsymbol{d}+\overbrace{\int_e \boldsymbol{B}^T\alpha\,dV\,\boldsymbol{L}p_{med}}^{\boldsymbol{F}_{p,e}}+\overbrace{\int_e \boldsymbol{H}^T\boldsymbol{f}dV}^{\boldsymbol{F}_{c,e}}-\overbrace{\gamma\int_e \boldsymbol{H}^T\boldsymbol{H}\,dV}^{\boldsymbol{M}_e}\frac{\boldsymbol{\partial^2 d}}{\boldsymbol{\partial t^2}}+\overbrace{\int_e \boldsymbol{H}^T\overline{\boldsymbol{q}}dA}^{\boldsymbol{F}_{q,e}}=\boldsymbol{0} \tag{3.69}$$

onde p_{med} é a pressão média no elemento.

4 ASPECTOS COMPUTACIONAIS

Neste capítulo serão apresentados os algoritmos utilizados nas implementações em CPU e em GPU, assim como também alguns detalhes da programação em GPU.

4.1 Problema estático

Nas implementações em CPU estão descritos os algoritmos para o cálculo das matrizes dos elementos, utilizando como referência o elemento CST, e a montagem da matriz global esparsa em dois formatos de armazenamento, além disso é apresentado o algoritmo de gradientes conjugados utilizado. Já nas implementações em GPU estão descritos detalhes da programação em GPU e os algoritmos utilizados nas operações internas do método dos gradientes conjugados.

4.1.1 Implementações em CPU

Como demonstrado na seção anterior, tudo que precisamos é calcular as matrizes de rigidez e os vetores de forças de cada elemento e agrupá-los na matriz e no vetor globais, como visto na Equação 4.1, para depois resolver o sistema formado. Nesta seção, será apresentado como foram feitas as montagens das matrizes e dos vetores, globais e locais.

$$\mathbf{K}^g\mathbf{d}^g = \mathbf{f}^{c,g} + \mathbf{f}^{q,g}. \tag{4.1}$$

A linguagem C++ foi escolhida para ser utilizada neste trabalho por ser uma linguagem de programação livre e por sua integração com a linguagem CUDA, além de permitir um código compilado de alto desempenho em estruturas de repetição. A linguagem C++ possui várias bibliotecas padrões e nestas bibliotecas não estão inclusas funções de álgebra linear, como funções de multiplicação de matrizes e de produto interno, também não estão inclusas funções para manipulação de matrizes esparsas. Dessa forma, há dois caminhos a seguir: utilizar bibliotecas externas, como a biblioteca Eigen ou a biblioteca Armadillo, que possuam funções e classes necessárias; ou criar funções com as bibliotecas padrões. O segundo foi o mais explorado, para o desenvolvimento de códigos para problemas mais específicos.

Ao optar por não utilizar bibliotecas externas, o Algoritmo 1 foi utilizado para a criação da matriz de rigidez de um elemento CST, onde a maior parte dele é destinada para as duas multiplicações de matrizes. Note que, na primeira multiplicação, a matriz $\mathbf{B}$ deve ser transposta, isto já está sendo considerado pela inversão dos índices na multiplicação.

Algoritmo 1 Montagem da Matriz de rigidez local $\mathbf{K}^e$ de elemento CST

Entrada: coordenadas dos nós do elemento, espessura t do elemento, E e ν
Saída: $\mathbf{K}^e$
Cálculo da área A^e do elemento
Montagem da matriz $\mathbf{D}_{3x3}$ utilizando E e ν;
Montagem da matriz $\mathbf{B}_{3x6}$ utilizando as coordenadas dos nós
Inicialização de $\mathbf{BD}_{6x3}$ e $\mathbf{K}^e_{6x6}$ como matrizes nulas;
for $i = 0 : 5$ **do**
  for $j = 0 : 2$ **do**
    for $k = 0 : 2$ **do**
      $BD_{i,j} = BD_{i,j} + B_{k,i}D_{k,j}$;
    end for
  end for
end for
for $i = 0 : 5$ **do**
  for $j = 0 : 5$ **do**
    for $k = 0 : 2$ **do**
      $K^e_{i,j} = K^e_{i,j} + BD_{i,k}B_{k,j}$;
    end for
    $K^e_{i,j} = K^e_{i,j}A^e t$;
  end for
end for

Esta função de criação de matriz de rigidez local é chamada dentro de outra função, que monta a matriz de rigidez global em formato esparso, em um laço por elemento. Isto é visto no Algoritmo 2.

A matriz esparsa pode ser armazenada em vários formatos, 2 deles foram escolhidos para serem utilizados nesta dissertação: o CRS (*Compressed Row Storage*) e o ELLPACK. No formato CRS, a matriz esparsa é armazenada em três vetores: o primeiro armazena os valores não nulos da matriz ordenados da esquerda pra direita e de cima pra baixo; o segundo são armazenados as posições das colunas desses valores não nulos na mesma ordem do primeiro vetor; e o terceiro vetor armazena a posição que inicia cada linha nos vetores anteriores, seu último valor é o número total de valores não nulos da matriz. Abaixo segue um exemplo.

$$\mathbf{M} = \begin{bmatrix} 2 & 0 & 4 & 0 \\ 0 & 7 & 9 & 0 \\ 3 & 18 & 0 & 5 \\ 0 & 20 & 0 & 0 \end{bmatrix} \longrightarrow \quad \begin{matrix} \mathbf{m}^1 = \begin{bmatrix} 2 & 4 & 7 & 9 & 3 & 18 & 5 & 20 \end{bmatrix} \\ \mathbf{m}^2 = \begin{bmatrix} 0 & 2 & 1 & 2 & 0 & 1 & 3 & 1 \end{bmatrix} \\ \mathbf{m}^3 = \begin{bmatrix} 0 & 2 & 4 & 7 & 8 \end{bmatrix} \end{matrix} . \tag{4.2}$$

No Algoritmo 2, vê-se como é feito o armazenamento da matriz global no formato CRS. Em um laço por elemento, de forma que cada matriz de rigidez local é incluída na global por vez. Neste laço, também são feitas transformações das coordenadas locais em globais. Então é

verificado se a posição já foi inclusa por outro elemento, caso afirmativo os valores são somados e, caso negativo, a posição nova é incluída.

Já no formato ELLPACK, primeiramente a matriz é compactada de modo que a matriz compacta tenha o número de colunas igual ao maior número de elementos não nulos entre as linhas da matriz original. Os elementos não nulos de cada linha são colocados nas primeiras colunas e as demais colunas são preenchidas com valores nulos. Desta forma, são armazenados dois vetores: o primeiro possui os elementos da matriz compacta coluna por coluna; o segundo vetor armazena a coluna da matriz original dos elementos do primeiro vetor.

$$\mathbf{M} = \begin{bmatrix} 2 & 0 & 4 & 0 \\ 0 & 7 & 9 & 0 \\ 3 & 18 & 0 & 5 \\ 0 & 20 & 0 & 0 \end{bmatrix} \longrightarrow \mathbf{M}_c = \begin{bmatrix} 2 & 4 & 0 \\ 7 & 9 & 0 \\ 3 & 18 & 5 \\ 20 & 0 & 0 \end{bmatrix}$$

$$\longrightarrow \quad \begin{array}{l} \mathbf{m}^1 = \begin{bmatrix} 2 & 7 & 3 & 20 & 4 & 9 & 18 & 0 & 0 & 0 & 5 & 0 \end{bmatrix} \\ \mathbf{m}^2 = \begin{bmatrix} 1 & 2 & 1 & 2 & 3 & 3 & 2 & 0 & 0 & 0 & 4 & 0 \end{bmatrix} \end{array} . \tag{4.3}$$

O formato ELLPACK tem vantagem sobre o CRS quando a matriz possui os números de elementos não nulos em cada linha aproximadamente iguais. Além disso o formato ELLPACK organiza a memória em uma melhor posição para ser acessada pela GPU. Se a quantidade de não zeros em cada linha for muito diferente o formato CRS é a melhor escolha já que é um formato mais geral.

O armazenamento da matriz no formato ELLPACK se faz por um algoritmo muito semelhante ao do armazenamento no formato CRS. Existe três diferenças: a primeira está na não criação do terceiro vetor, a segunda é que os dois primeiros serão do tamanho do número de graus de liberdade multiplicado pelo maior número de elementos não nulos entre as linhas; e a última consiste em que os dois laços mais internos são invertidos.

Algoritmo 2 Montagem da Matriz de rigidez global $\mathbf{K}^g$ no formato CRS

Entrada: coordenadas dos nós, conectividade dos elementos, espessura t dos elementos, E e ν.
Saída: $\mathbf{K}^g$ na forma dos três vetores $\mathbf{k}^{g,1}$, $\mathbf{k}^{g,2}$ e $\mathbf{k}^{g,3}$, como visto na equação 4.2.
ngl : número de graus de liberdade;
non : número de elementos não nulos da matriz global;
Inicializar vetores $\mathbf{k}^{g,1}_{non}$, $\mathbf{k}^{g,2}_{non}$ e $\mathbf{k}^{g,3}_{ngl+1}$ como vetores nulos;
for $e = 0$: número total de elementos -1 **do** %Laço nos elementos
  Cálculo da matriz de rigidez do elemento $\mathbf{K}^e$;
  Cálculo do vetor de graus de liberdade globais $\mathbf{gl}_6$;
  for $i = 0 : 5$ **do** %Laço nas linhas da matriz local
    $lg = gl_i$; posição global da linha da matriz local
    for $j = 0 : 5$ **do** %Laço nas colunas da matriz local
      $cg = gl_j$; posição global da coluna da matriz
      Buscar cg dentre os elementos correspondentes a linha lg em $\mathbf{k}^{g,2}$;
      if For encontrado **then**
        h recebe a posição de $\mathbf{k}^{g,2}$ correspondente a cg;
        $\mathbf{k}^{g,1}_h = \mathbf{k}^{g,1}_h + \mathbf{K}^e_{i,j}$
      else
        Reorganizar $\mathbf{k}^{g,1}$ e $\mathbf{k}^{g,2}$;
        h recebe a posição de $\mathbf{k}^{g,2}$ correspondente a posição (lg, cg);
        $k^{g,1}_h = K^e_{i,j}$;
        $k^{g,2}_h = cg$;
        $k^{g,3}_{lg} = k^{g,3}_{lg} + 1$;
      end if
    end for
  end for
end for

Caso haja preferência em trabalhar com as bibliotecas externas, esses algoritmos ficam mais simples, porém deve-se trabalhar com as classes disponibilizadas por essas bibliotecas. No caso da biblioteca Eigen, é disponibilizada uma classe para matriz densa (*Matrix*) e outra para esparsa (*SparseMatrix*), com vetores sendo casos especiais destas. Neste caso, a multiplicação e a divisão por escalar com vetores densos ou com matrizes densas pode ser realizadas como feito escalar por escalar. Disponibiliza também a função membro *transposeInPlace()* para a realização de transpostas de matrizes densas. E a última melhoria disponibilizada pela Eigen, para o Algoritmo 1, é que a multiplicação entre matrizes densas pode ser escrita exatamente como a multiplicação escalar por escalar. Logo, este pode ser escrito sem nenhum laço, de forma que facilite o entendimento do código e diminua o esforço do programador.

Para a manipulação de matrizes esparsas, a biblioteca Eigen permite comprimir a matriz no formato CRS. Para a montagem da matriz esparsa, esta biblioteca fornece uma outra classe chamada *Triplet* que armazena a linha, a coluna e o valor não nulo. Ao ser armazenado todos os valores das matrizes de rigidez locais em um vetor de *Triplets* $\mathbf{t}$ e declarado $\mathbf{M}$ como uma matriz esparsa, a expressão $\mathbf{M}.setFromTriplets(\mathbf{t}.begin(), \mathbf{t}.end())$ faz de $\mathbf{M}$ uma matriz esparsa no

formato CRS onde todos os valores não nulos em $\mathbf{t}$ com posições iguais são somados. Logo, o trabalho é armazenar os valores de todos elementos das matrizes de rigidez dos elementos em um vetor de *Triplets* com as coordenadas globais das linhas e colunas.

Em Eigen, também é possível encontrar classe *ConjugateGradient* e com ela pode-se resolver sistemas lineares com matrizes esparsas e densas. Basta usar algumas funções membros para se solucionar um sistema. Na criação, o primeiro argumento é o tipo de matriz do sistema e o segundo o tipo de precondicionador a ser usado. Então, utiliza-se a expressão $cg.compute(\mathbf{A})$ para inserir a matriz do problema $\mathbf{A}$ no objeto cg. A expressão $\mathbf{x} = cg.solve(\mathbf{b})$ armazena no vetor $\mathbf{x}$ o resultado do sistema $\mathbf{Ax} = \mathbf{b}$. Para mais informações acessar a referência MediaWiki (2019).

4.1.2 Implementações em GPU

Alguns conceitos de programação em GPU precisam ser introduzidos para o melhor entendimento das explicações:

host - CPU onde o código será implementado e de onde o código em GPU será chamado;

***device* ou dispositivo** - GPU onde será implementado o código paralelizado;

kernel - função a ser processada na GPU;

thread - cada realização de uma *kernel*, a chamada de cada *kernel* deve especificar o número de realizações a serem realizadas, cada realização é feita em um núcleo CUDA diferente da GPU;

bloco de *threads* - o número de realizações, *threads*, de uma *kernel* deve ser organizada em blocos, que devem possuir no máximo 1024 *threads*;

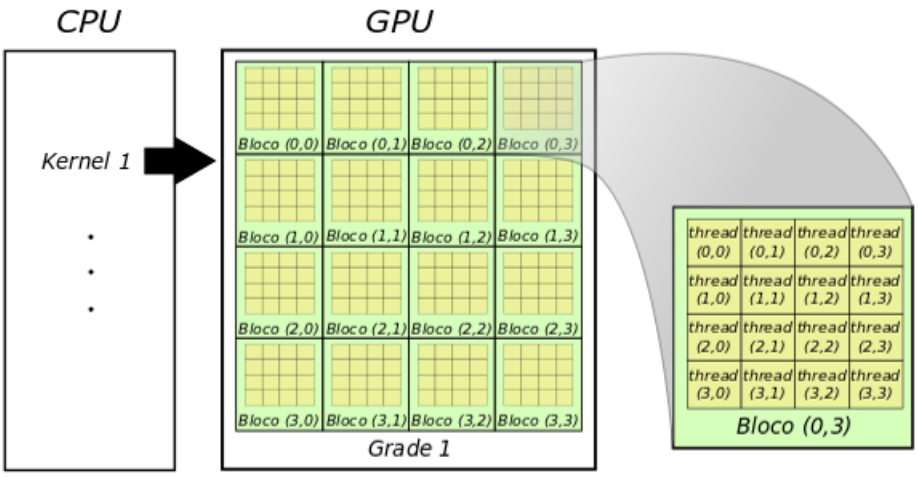
grade - os blocos de *threads* devem ser organizados em uma grade.

Para a utilização dos recursos da GPU, parte do código deve ser executado no *host* (CPU). que chamará a execução da parte que estará no dispositivo (GPU). A inicialização acontece no *host* que então chamará as partes, chamadas de *kernels*, a serem executadas no dispositivo. Cada *kernel* é uma função do tipo SPMD (*Single Program Multiple Data*) e deve ser especificada quantas vezes será executada em paralelo, essas execuções (*threads*) são organizadas em blocos que possuem a mesma quantidade de execuções e podem ser organizadas em até três dimensões. Os blocos de *threads* podem ser organizados em uma grade com até três dimensões, cada grade executa um *kernel*. Um esquema de organização de *threads* e blocos pode ser visto na Figura 8(a).

A implementação foi feita em uma GPU da NVIDIA com capacidade computacional 6.1 através da linguagem CUDA na versão 10, desenvolvida pela própria NVIDIA, em integração com a linguagem C++. Uma GPU possui dezenas de SM (*Streaming Multiprocessors*), e cada

um destes possui geralmente um número de núcleos CUDA múltiplo de 32. Um esquema básico da arquitetura de um SM pode ser visto na Figura 8(b). Cada SM pode ser responsável pela execução de um ou mais blocos de *threads*. Os *threads* no mesmo bloco devem executar as mesmas operações, de forma que análises condicionais podem bloquear a execução de alguns *threads* temporariamente.

Figura 7 – Esquemas de organização e arquitetura CUDA



(a) Organização dos *threads* e blocos.
Fonte: O Autor, 2020.

(b) SM de um processador gráfico GP100.
Fonte: NVIDIA, 2016.

Antes da utilização do código no dispositivo, todas as variáveis devem ser transferidas para sua memória global, dado o fato que o dispositivo não acessa a memória do *host*. As variáveis que serão calculadas no dispositivo e serão necessárias no *host* devem ser pré alocadas na memória global do dispositivo pelo *host*. Já existe uma função CUDA que pré aloca uma posição de memória que pode ser modificada no *host* e no dispositivo, com isso não há a preocupação de transferir a memória, bastando a sincronização de ambos quando for utilizar a

variável no *host*. Temos no dispositivo: a memória global, que pode ser acessada por todos os *threads* utilizados; a memória constante, uma parte da memória global mais próxima dos SMs para um acesso mais rápido a informação; a memória local, particular a cada *thread* e por isso pequena; a memória compartilhada, que pode ser acessada por qualquer *thread* no mesmo bloco e tem um acesso bem mais rápido que a memória global. Destas, apenas a global e a constante, por ser uma parte da global, são mantidas ao fim do *kernel*.

Em várias aplicações como somatório de um vetor ou máximo valor de um vetor se fazem necessárias a sincronização e a troca de informação entre os *threads* para não criar uma condição de corrida, onde um *thread* utiliza um valor desatualizado e que está sendo atualizado por outro *thread* no mesmo momento. A sincronização dos *threads* no mesmo bloco pode ser feita dentro do *kernel* sem problemas e a troca de informação se dá pela memória compartilhada. Porém, a sincronização de todos os *threads* não é possível dentro de um mesmo *kernel* e o único modo de trocar informações é pela memória global. A forma encontrada para a sincronização de todos os *threads* é com o encerramento do *kernel*.

Como descrito na primeira seção, optou-se apenas por solucionar o sistema de equações lineares, parte mais dispendiosa do código, na GPU. A solução do sistema linear foi feita pelo Algoritmo 3 do método dos gradientes conjugados. Basicamente, tem-se 3 tipos de operações em cada iteração: produto de matriz esparsa com vetor denso; operações de soma e subtração entre vetores; e produto interno.

Algoritmo 3 Método dos gradientes conjugados

Entrada: $\mathbf{K}^g$, $\mathbf{f}^g$, $\mathbf{u}_0$, tol, $maxit$
Saída: $\mathbf{u}$
$\mathbf{r} = \mathbf{f}^g - \mathbf{K}^g\mathbf{u}_0$;
$\mathbf{d} = \mathbf{r}$;
$i = 0$; $erro = \infty$;
while $erro > tol$ ou $i < maxit$ **do**
  $\alpha = \frac{\mathbf{r}^T\mathbf{d}}{\mathbf{d}^T\mathbf{K}^g\mathbf{d}}$;
  $\mathbf{r} = \mathbf{r} - \alpha\mathbf{K}^g\mathbf{d}$;
  $\mathbf{u}_0 = \mathbf{u}_0 + \alpha\mathbf{d}$;
  $\beta = \frac{-\mathbf{r}^T\mathbf{K}^g\mathbf{d}}{\mathbf{d}^T\mathbf{K}^g\mathbf{d}}$;
  $\mathbf{d} = \mathbf{r} + \beta\mathbf{d}$;
  $erro = \frac{\sqrt{\mathbf{r}^T\mathbf{r}}}{\sqrt{\mathbf{F}^{g,T}\mathbf{F}^g}}$;
  $i = i + 1$;
end while
$\mathbf{u} = \mathbf{u}_0$;

Um meio de executar o Algoritmo 3 na GPU é tornar cada uma dessas operações um *kernel*. Para a multiplicação de matriz esparsa por vetor denso, cada *thread* receberá uma linha da matriz e fará um laço pelos elementos não-nulos desta linha, multiplicando-os pelo valor correspondente a suas posições no vetor e somando os resultados em um único valor. Já para operações entre vetores, elemento por elemento, cada *thread* receberá os valores correspondentes

a mesma posição em ambos os vetores, fará a operação e armazenará em um terceiro na mesma posição. Os Algoritmos de 4 a 6 demonstram o funcionamento destes *kernels*. Os Algoritmos de 4 e 6 podem ser encontrados no Apêndice A na linguagem C++ e no Apêndice B na linguagem CUDA junto com o Algoritmo 5.

Algoritmo 4 Multiplicação entre matriz esparsa no formato CRS e vetor denso na GPU

Entrada: M - matriz esparsa no formato CRS como demonstrado na equação 4.2, b, dim - dimensão
Saída: x
$tidx =$ índice do *thread*; indica qual linha cada *thread* irá receber
if $tidx < dim$ **then**
    $soma = 0$;
    for $i = m^3_{tidx} : m^3_{tidx+1} - 1$ **do**
        $soma = soma + m^1_i * b_{m^2_i}$;
    end for
    $x_{tidx} = soma$;
end if

Algoritmo 5 Multiplicação entre matriz esparsa no formato ELLPACK e vetor denso na GPU

Entrada: M - matriz esparsa no formato ELLPACK como demonstrado na equação 4.3, b, dim - dimensão, ncm - número de colunas da matriz compacta
Saída: x
$tidx =$ índice do *thread*; indica qual linha cada *thread* irá receber
if $tidx < dim$ **then**
    $soma = 0$;
    for $i = 0 : ncm - 1$ **do**
        $j = tidx + ngl * i$;
        $soma = soma + m^1_j * b_{m^2_j}$;
    end for
    $x_{tidx} = soma$;
end if

Algoritmo 6 Operações entre vetores, elemento por elemento, exemplificado com a soma na GPU

Entrada: a, b, dim - dimensão
Saída: c
$tidx =$ índice do *thread*;
if $tidx < dim$ **then**
    $c_{tidx} = a_{tidx} + b_{tidx}$;
end if

O produto interno é a parte mais complexa, já que é uma redução que precisa da comunicação entre *threads* de blocos diferentes. Essa comunicação exige uma sincronização entre todos os *threads* e, como já foi falado antes, será feita através do encerramento de um *kernel*.

Outro meio de se fazer estas comunicações é através das funções atômicas disponibilizadas pela linguagem CUDA. O somatório é feito dois a dois até que a dimensão seja reduzida a 1. O Algoritmo 7 mostra o *kernel* de um passo desta redução e o Algoritmo 8 mostra como a CPU o chama sucessivamente.

Algoritmo 7 Passo para o somatório dos elementos do vetor $\mathbf{a}$ na GPU (necessário para o produto interno)

Entrada: $\mathbf{a}$, dim - dimensão
$tidx =$ índice do *thread*;
if $tidx < dim$ **then**
    $a_{tidx} = a_{tidx} + a_{tidx+dim}$;
end if

Algoritmo 8 Somatório do vetor $\mathbf{a}$ na GPU (necessário para o produto interno)

Entrada: $\mathbf{a}$, dim - dimensão
while $dim > 1$ **do**
    if dim é ímpar **then**
        $dim = \frac{dim}{2} + 1$;
    else
        $dim = \frac{dim}{2}$;
    end if
    Algoritmo 6 com entradas $\mathbf{a}$ e dim;
end while

Algumas modificações foram feitas para a melhor utilização da GPU. O algoritmo de gradientes conjugados foi rearranjado para possibilitar a diminuição de chamadas de *kernels* com o agrupamento de algumas funções. O produto da matriz esparsa pelo vetor denso é feito apenas uma vez por iteração e armazenado em um vetor auxiliar para a reutilização. Essa reorganização do Algoritmo 3 é mostrada no Algoritmo 9.

Algoritmo 9 Método dos gradientes conjugados

Entrada: $\mathbf{K}^g$, $\mathbf{f}^g$, $\mathbf{u}_0$, tol, $maxit$
Saída: $\mathbf{u}$
$\mathbf{r} = \mathbf{f}^g - \mathbf{K}^g\mathbf{u}_0$;
$\beta = 0$;
$mf = \mathbf{f}^{g,T}\mathbf{f}^g$
$i = 0$; $erro = \infty$;
while $\sqrt{\frac{erro}{mf}} > tol$ ou $i < maxit$ **do**
  $\mathbf{d} = \mathbf{r} + \beta\mathbf{d}$;
  $\mathbf{aux} = \mathbf{K}^g\mathbf{d}$;
  $\alpha = \mathbf{r}^T\mathbf{d}$;
  $\gamma = \mathbf{d}^T\mathbf{aux}$;
  $\mathbf{r} = \mathbf{r} - \frac{\alpha}{\gamma}\mathbf{aux}$;
  $\mathbf{u}_0 = \mathbf{u}_0 + \frac{\alpha}{\gamma}\mathbf{d}$;
  $\beta = -\mathbf{r}^T\mathbf{aux}$;
  $erro = \mathbf{r}^T\mathbf{r}$;
  $i = i + 1$;
end while
$\mathbf{u} = \mathbf{u}_0$;

Desta forma o cálculo do vetor $\mathbf{aux}$ é feito no mesmo *kernel* que o cálculo dos vetores antes das reduções de α e γ, assim como as atualizações dos vetores $\mathbf{r}$ e $\mathbf{u}_0$. Os cálculos dos vetores antes das reduções de β e do $erro$ são feitos no mesmo *kernel*. As reduções necessárias para α e γ são feitas juntas, assim como as reduções para o $erro$ e β. Logo, em cada iteração é feito três chamadas de *kernels* diretas e duas reduções, que envolvem chamadas sucessivas de um mesmo *kernel*. A quantidade de chamadas do *kernel* das reduções depende do tamanho do problema.

Para melhorar o desempenho, o *kernel* responsável pela multiplicação da matriz esparsa pelo vetor foi modificado para a utilização da memória compartilhada. A memória compartilhada, como foi dito antes, possui uma maior velocidade de acesso em relação a global. Porém, é esvaziada com o fim do *kernel*. Ou seja, no início de cada *kernel* as variáveis na memória global devem ser transferidas para a compartilhada. Isso só será vantagem se dentro do mesmo *kernel* essa variável for acessada mais de uma vez, pois passar da memória global para a compartilhada já é um acesso. Como na multiplicação de matriz esparsa por vetor alguns *threads* do mesmo bloco irão acessar as posições iguais do vetor, convém passar as posições do vetor utilizadas pelos *threads* do bloco para a memória compartilhada. Após a passagem das variáveis para a memória compartilhada, deve ser feita a sincronização dos *threads* residentes no mesmo bloco.

Outro meio de acelerar a aplicação é pelo meio da utilização da precisão simples em algumas operações. Em processadores gráficos como o GP100 com SMs que possuem uma organização favorável para a utilização de precisão dupla (Figura 8(b)) a aplicação já é bastante eficiente em precisão dupla. Porém muitos deles não possuem esta arquitetura. Todos os processadores de arquitetura Maxwell, por exemplo, são 32 vezes mais lentos em precisão dupla,

enquanto que, o GP100 é apenas duas (NVIDIA, 2016). O grande problema da utilização da precisão simples é a possibilidade de acarretar uma não convergência do método dos gradientes conjugados. Para analisar a possibilidade do uso da precisão simples em alguns *kernels*, aplicações foram feitas em ambas as precisões comparando os resultados e velocidades.

4.2 Análise Dinâmica

Nesta seção são apresentados a formulação e os algoritmos utilizados na integração direta no tempo. Assim como detalhes das implementações, como o cálculo do passo de tempo do método explícito para a garantia da estabilidade do método e o uso da matriz de massa consistente.

4.2.1 Problema Mecânico

A equação 3.29 pode ser escrita da forma:

$$\boldsymbol{K}\boldsymbol{d} + \boldsymbol{M}\ddot{\boldsymbol{d}} = \boldsymbol{F}. \tag{4.4}$$

onde temos:

$$\ddot{\boldsymbol{d}} = \frac{\boldsymbol{\partial^2 d}}{\boldsymbol{\partial t^2}}.$$

Com as condições iniciais:

$$\boldsymbol{d}(t=0) = \boldsymbol{d}_0, \qquad \dot{\boldsymbol{d}}(t=0) = \dot{\boldsymbol{d}}_0.$$

Segundo Barros et al. (2002), os processos de integração numérica podem ser por métodos diretos ou por superposição modal. Os métodos de integração direta se dividem em explícitos, onde não há a inversão da matriz de rigidez, e implícitos, onde há a inversão da matriz de rigidez. Os métodos de integração direta consistem na discretização do domínio do tempo em intervalos de tempo, que pode ser de mesmo tamanho ou não, aproximando os deslocamentos, as velocidades e acelerações em cada intervalo de tempo.

Craig Jr. e Kurdilla (2006) apresenta o método da diferença central, onde é usado diferenças finitas centrais para a aproximação das derivadas no tempo. Substituindo a aproximação por diferenças finitas centrais na equação 4.4:

$$\boldsymbol{K}\boldsymbol{d}_i + \boldsymbol{M}\frac{\boldsymbol{d}_{i-1} - 2\boldsymbol{d}_i + \boldsymbol{d}_{i+1}}{\Delta t^2} = \boldsymbol{F}_i. \tag{4.5}$$

Rearranjando a equação acima de forma que o deslocamento a ser calculado seja o deslocamento futuro:

$$\boldsymbol{M}\boldsymbol{d}_{i+1} = \boldsymbol{F}_i\Delta t^2 - \boldsymbol{M}\boldsymbol{d}_{i-1} + (2\boldsymbol{M} - \Delta t^2\boldsymbol{K})\boldsymbol{d}_i. \tag{4.6}$$

Este método consiste em calcular um passo futuro com o conhecimento de dois passos anteriores. Como a matriz a ser invertida não é a de rigidez, trata-se de método explícito. Nestes tipos de métodos a escolha do tamanho de passo de tempo é muito importante, passos de tempo elevados podem ocasionar respostas muito erradas. Belytschchko, Liu e Moran (2000) definiram um passo de tempo que garante a estabilidade do método para elementos de tensão constante:

$$\Delta t = \alpha \, \Delta t_{crit} \quad , \quad \Delta t_{crit} = \frac{2}{\omega_{max}}. \tag{4.7}$$

onde Δt_{crit} é o passo de tempo crítico, ω_{max} é a maior frequência do sistema e uma boa escolha para α se encontra entre:

$$0,8 \leq \alpha \leq 0,98.$$

Para evitar a preocupação com o passo de tempo, pode-se, em vez de utilizar a diferença finita central, a diferença finita regressiva:

$$\boldsymbol{K}\boldsymbol{d}_i + \boldsymbol{M}\frac{\boldsymbol{d}_{i-2} - 2\boldsymbol{d}_{i-1} + \boldsymbol{d}_i}{\Delta t^2} = \boldsymbol{F}_i. \tag{4.8}$$

Desta forma, em vez de utilizar um método explícito, utilizamos um método implícito. Neste caso há a inversão da matriz de rigidez, o que torna o método mais estável e menos dependente do tamanho do passo de tempo.

$$(\boldsymbol{K} + \frac{1}{\Delta t^2}\boldsymbol{M})\boldsymbol{d}_i = \boldsymbol{F}_i - \boldsymbol{M}\frac{\boldsymbol{d}_{i-2} - 2\boldsymbol{d}_{i-1}}{\Delta t^2}. \tag{4.9}$$

Na condição de contorno de velocidade é aplicada a aproximação por diferença finita central ficando assim conhecido o deslocamento do tempo duas vezes anterior ao que se deseja calcular.

$$\boldsymbol{d}_{-1} = \boldsymbol{d}_1 - 2\Delta t\dot{\boldsymbol{d}}_0. \tag{4.10}$$

Substituindo na equação 4.9 temos para o primeiro passo de tempo:

$$(\boldsymbol{K} + \frac{2}{\Delta t^2}\boldsymbol{M})\boldsymbol{d}_1 = \boldsymbol{F}_1 - \boldsymbol{M}\frac{-2\Delta t\dot{\boldsymbol{d}}_0 - 2\boldsymbol{d}_0}{\Delta t^2}. \tag{4.11}$$

E, para o método explícito, substituindo 4.10 em 4.6:

$$2\boldsymbol{M}\boldsymbol{d}_1 = \boldsymbol{F}_0\Delta t^2 + 2\Delta t\boldsymbol{M}\dot{\boldsymbol{d}}_0 + (2\boldsymbol{M} - \Delta t^2\boldsymbol{K})\boldsymbol{d}_0. \tag{4.12}$$

O Algoritmo 10 demonstra a forma que foi feita a integração implícita neste trabalho:

Algoritmo 10 Método implícito de integração do tempo

Entrada: $\mathbf{K}^g$, $\overline{\boldsymbol{M}}^g$ e $\mathbf{f}_t^g$ em todos o tempos a serem calculados;
Saída: $\mathbf{x}_t$ em todos os tempos a serem calculados;
ngl : número de graus de liberdade;
$\boldsymbol{K}_{eq} = \boldsymbol{K}^g + \frac{1}{\Delta t^2}\overline{\boldsymbol{M}}^g$;
for $t = 1 : \Delta t : t_{final}$ **do**
  if t=1 **then**
    $\boldsymbol{f}_{eq} = \boldsymbol{f}_t^g - \overline{\boldsymbol{M}}^g \frac{-2\Delta t \dot{\boldsymbol{d}}_{t-1} - 2\boldsymbol{d_{t-1}}}{\Delta t^2}$;
    $\boldsymbol{K}_{eq} = \boldsymbol{K}_{eq} + \frac{1}{\Delta t^2}\overline{\boldsymbol{M}}^g$;
    Encontrar $\boldsymbol{d}_t$ por gradientes conjugados;
    $\boldsymbol{K}_{eq} = \boldsymbol{K}_{eq} - \frac{1}{\Delta t^2}\overline{\boldsymbol{M}}^g$;
  else
    $\boldsymbol{f}_{eq} = \boldsymbol{f}_t^g - \overline{\boldsymbol{M}}^g \frac{\boldsymbol{d_{t-2}} - 2\boldsymbol{d_{t-1}}}{\Delta t^2}$;
    Encontrar $\boldsymbol{d}_t$ por gradientes conjugados;
  end if
end for

Para uma convergência mais rápida do método dos gradientes conjugados, para a resolução de cada passo de tempo usa-se como estimativa inicial o resultado do passo de tempo anterior. O quão próximo a estimativa inicial está depende da variação do vetor de forças e do passo de tempo utilizado.

Uma outra forma de realizar a integração no tempo de forma explícita é calcular uma velocidade meio passo de tempo a frente a partir da aceleração do passo anterior conhecida e da velocidade anterior conhecida. O cálculo do deslocamento futuro se dá com o conhecimento desta velocidade, e o cálculo da aceleração futura é feito pela Equação 4.4. E assim sucessivamente até o tempo desejado. O Algoritmo 11 abaixo representa esta forma.

Algoritmo 11 Integração explícita no tempo

Entrada: $\mathbf{K}^g$,$\mathbf{M}^g$, $\mathbf{f}^g$, $\mathbf{u}_0$, $\mathbf{v}_0$, $\mathbf{a}_0$,Δt,t_{fim}.
$npt = \frac{t_{fim}}{\Delta t}$;
$\mathbf{v}_m = \mathbf{v}_0 + \frac{\Delta t}{2}\mathbf{a}_0$;
for $i = 1 : npt$ **do**
  $\mathbf{u}_i = \mathbf{u}_{i-1} + \Delta t\, \mathbf{v}_m$;
  $\mathbf{a}_i = \mathbf{M}_g^{-1}(\mathbf{f}_{g,i} - \mathbf{K}_g\mathbf{u}_i)$;
  $\mathbf{v}_m = \mathbf{v}_m + \Delta t \mathbf{a}_i$;
end for

O método explícito tem apenas um inconveniente, o passo de tempo precisa ser muito pequeno para garantir a estabilidade do método. Para um mesmo período um método explícito necessita de mais passos de tempo do que métodos implícitos, a não ser que o vetor de forças varie em um passo ainda menor. Além disto, o cálculo do passo de tempo envolve o cálculo da maior frequência do sistema, que pode ser realizado pelo método da potência.

4.2.2 Problema de Fluxo

Assim como feito para o problema mecânico, a integração implícita pode ser utilizada no problema de fluxo a partir da aproximação da derivada no tempo por diferença finita regressiva. Logo a Equação 3.63 fica:

$$\boldsymbol{K}_f^e \boldsymbol{p}_i + \boldsymbol{C}_f^e \frac{\boldsymbol{p}_i - \boldsymbol{p}_{i-1}}{\Delta t} = \boldsymbol{F}_{f,i}^e \tag{4.13}$$

Isolando a pressão que se deseja calcular:

$$\boldsymbol{p}_i = (\boldsymbol{K}_f^e + \boldsymbol{C}_f^e \frac{1}{\Delta t})^{-1} (\boldsymbol{F}_{f,i}^e + \boldsymbol{C}_f^e \frac{\boldsymbol{p}_{i-1}}{\Delta t}) \tag{4.14}$$

O Algoritmo 12 descreve a forma como esta integração foi realizada:

Algoritmo 12 Método implícito de integração do tempo para o fluxo

Entrada: $\mathbf{K}^g$, $\overline{\boldsymbol{M}}^g$ e $\mathbf{f}_t^g$ em todos o tempos a serem calculados;
Saída: $\mathbf{x}_t$ em todos os tempos a serem calculados;
ngl : número de graus de liberdade;
$\boldsymbol{K}_{eq} = \boldsymbol{K}^g + \frac{1}{\Delta t}\overline{\boldsymbol{M}}^g$;
for $t = 1 : \Delta t : t_{final}$ **do**
  if t=1 **then**
    $\boldsymbol{f}_{eq} = \boldsymbol{f}_t^g - \overline{\boldsymbol{M}}^g \frac{-2\Delta t \dot{\boldsymbol{d}}_{t-1} - 2\boldsymbol{d_{t-1}}}{\Delta t}$;
    $\boldsymbol{K}_{eq} = \boldsymbol{K}_{eq} + \frac{1}{\Delta t^2}\overline{\boldsymbol{M}}^g$;
    Encontrar $\boldsymbol{d}_t$ por gradientes conjugados;
    $\boldsymbol{K}_{eq} = \boldsymbol{K}_{eq} - \frac{1}{\Delta t^2}\overline{\boldsymbol{M}}^g$;
  else
    $\boldsymbol{f}_{eq} = \boldsymbol{f}_t^g - \overline{\boldsymbol{M}}^g \frac{\boldsymbol{d_{t-2}} - 2\boldsymbol{d_{t-1}}}{\Delta t}$;
    Encontrar $\boldsymbol{d}_t$ por gradientes conjugados;
  end if
end for

Para o método explícito o procedimento é semelhante ao da segunda forma apresentada para o mecânico. A pressão do passo futuro é calculada tendo o conhecimento da velocidade e da pressão anterior. E a velocidade futuro é calculada a partir da Equação 3.63. O Algoritmo 13 abaixo apresenta o procedimento.

Algoritmo 13 Integração explícita no tempo do problema de fluxo

Entrada: $\mathbf{K}_f^g$,$\mathbf{C}_f^g$, $\mathbf{f}_f^g$, $\mathbf{p}_0$, $\dot{\mathbf{p}}_0$,Δt,t_{fim}.
$npt = \frac{t_{fim}}{\Delta t}$;
for $i = 1 : npt$ **do**
  $\mathbf{p}_i = \mathbf{p}_{i-1} + \Delta t\, \dot{\mathbf{p}}_{i-1}$;
  $\dot{\mathbf{p}}_i = (\mathbf{C}_f^g)^{-1}(\mathbf{f}_{f,i}^g - \mathbf{K}_f^g \mathbf{p}_i)$;
end for

O passo de tempo no método explícito para o problema de fluxo é calculado da mesma forma como é feito para o problema mecânico. Logo é necessário o cálculo da maior frequência do sistema. Ao se acoplar os dois problemas o passo de tempo utilizado foi o menor dos dois problemas.

4.2.3 Outros algoritmos

Para ambas as formas de integração no tempo é necessária a montagem das matrizes de massa dos elementos e da matriz de massa global. Pode optar-se por utilizar a matriz de massa consistente que deriva da integração indicada na Equação 3.28. Ou, de forma mais simples, utilizar a matriz de massa concentrada. A matriz de massa concentrada, segundo Zienkiewicz e Taylor (2000), pode ser obtida segundo a Equação 4.15.

$$\overline{\boldsymbol{M}}_{ii} = \sum_j \boldsymbol{M}_{ij}. \tag{4.15}$$

Onde $\boldsymbol{M}$ é a matriz de massa consistente e $\overline{\boldsymbol{M}}$ a matriz de massa concentrada. Esta última é uma matriz diagonal. A montagem da matriz de massa concentrada do elemento consiste na criação de um vetor preenchido com a massa do elemento dividido pelo número de nós. A montagem da matriz de massa concentrada global é demonstrada no Algoritmo 14.

Algoritmo 14 Montagem da matriz de massa concentrada global $\overline{\boldsymbol{M}}^g$

Entrada: coordenadas dos nós, conectividade dos elementos, espessura t dos elementos e a massa específica γ;
Saída: $\overline{\mathbf{M}}^g$ como um vetor nulo;
ngl : número de graus de liberdade;
Inicializar vetor $\overline{\mathbf{M}}^g_{ngl}$;
for $e = 0$: número total de elementos -1 **do** %Laço nos elementos
  $m^e = A^e t \gamma$: massa do elemento;
  Cálculo da matriz de massa concentrada do elemento $\overline{\boldsymbol{M}}^e$;
  Cálculo do vetor de graus de liberdade globais $\mathbf{gl}_6$;
  for $i = 0 : 5$ **do** %Laço nos graus de liberdade
    $\overline{\boldsymbol{M}}^g_j = \overline{\boldsymbol{M}}^g_j + \overline{\boldsymbol{M}}^e_i$;
  end for
end for

O método da potência é um método para encontrar o maior autovalor de uma matriz. Consiste na multiplicação de um vetor aleatório pela matriz e a divisão do vetor resultante pelo seu maior valor absoluto. Esse processo é realizado até que o maior valor absoluto do vetor resultante seja próximo suficiente do vetor anterior. Este valor é o maior autovalor da matriz. E a matriz da qual se quer obter o maior autovalor é a matriz $\overline{\boldsymbol{M}}^{-1}\boldsymbol{K}$. O Algoritmo 15 demonstra este processo.

Algoritmo 15 Método da potência

Entrada: $\mathbf{K}^g$ e $\overline{\boldsymbol{M}}^g$;
Saída: λ_{max} - maior autovalor;
ngl: número de graus de liberdade;
for $i = 0 : ngl$ **do**
    $\boldsymbol{Mi}_i^g = \frac{1}{\overline{\boldsymbol{M}}_i^g}$
end for
$\boldsymbol{K}_{eq} = \mathbf{K}^g \boldsymbol{Mi}^g$
$\boldsymbol{v}$: vetor onde todos os elementos são iguais a 1 do tamanho ngl;
for $i = 0 : 50$ **do**
    $\boldsymbol{v} = \boldsymbol{K}_{eq}\boldsymbol{v}$;
    $\lambda = max(\boldsymbol{v})$;
    $\boldsymbol{v} = \frac{1}{\lambda}\boldsymbol{v}$;
end for
$\lambda_{max} = \lambda$;

O passo de tempo é calculado através da Equação 4.7. Com isso, o processo segue como descrito pelo Algoritmo 11, para o problema mecânico, e pelo Algoritmo 13, para o problema de fluxo.

5 APLICAÇÕES E DISCUSSÕES

Duas aplicações estáticas e duas dinâmicas foram realizadas a fim de testar o desempenho da implementação na GPU. Para a comparação dos tempos de resolução, as aplicações também foram realizadas em um algoritmo implementado completamente na CPU. As realizações na CPU são feitas em um processador Intel Core I7 8700 de 3,20 GHz de 6 núcleos e uma memória RAM de 32 GB. Já as na GPU, utilizam uma NVIDIA TITAN Xp de 12 GB de memória. Esta GPU possui a arquitetura Pascal da NVIDIA e 3840 núcleos CUDA. As aplicações foram realizadas no ambiente do sistema operacional Windows 10.

Nas aplicações em duas dimensões foi utilizado o elemento CST para o problema mecânico. Já para as aplicações em três dimensões foram utilizados os elementos tetraédricos de quatro nós, para os problemas mecânico e de fluxo.

5.1 Aplicações estáticas

Para as aplicações estáticas, as resoluções na CPU foram feitas em um algoritmo utilizando as bibliotecas básicas do C++, outro utilizando a biblioteca Eigen, outro utilizando a estrutura OpenMP - para a utilização de todos os núcleos do processador - e por fim, outro no MATLAB. A resolução na GPU foi realizada por três algoritmos diferentes, implementados na linguagem C++, e no MATLAB. O primeiro, $GPU1$, foi uma aplicação direta do algoritmo dos gradientes conjugados apresentado no Algoritmo 3, onde cada operação foi realizada em um *kernel*; o segundo, $GPU2$, foi um rearranjo do primeiro algoritmo, para a diminuição da quantidade de *kernels*; o terceiro algoritmo, $GPU3$, além de ser um rearranjo do primeiro, realiza a cópia de algumas variáveis da memória global para a compartilhada e as utiliza a partir da segunda. Os dois últimos algoritmos têm como base o Algoritmo 9. Os algoritmos implementados na linguagem C++ podem ser utilizados com pontos flutuantes de precisão dupla (PD) e simples (PS). Porém, no MATLAB, as matrizes esparsas só podem ser utilizadas com precisão dupla. Os códigos utilizados no MATLAB podem ser encontrados no Apêndice C.

A primeira aplicação considera uma viga no estado plano de tensões e a segunda um maciço de solo no estado plano de deformações. As aplicações foram realizadas com diferentes tamanhos de malha para a análise de ganho de velocidade com diferentes números de graus de liberdade (NGL). A Tabela 1 possui os dados utilizados em cada aplicação e a Tabela 2 possui um resumo de quais algoritmos foram utilizados em cada aplicação.

Tabela 1 – Dados utilizados em cada aplicação estática

Aplicações	Forma do domínio	Base(m)	Altura(m)	t(m)	E	ν
Viga	Quadrilátero regular	10	1	1	30GPa	0,3
Solo	Quadrilátero regular	10	10	1	30MPa	0,25

Fonte: O Autor, 2020.

Tabela 2 – Algoritmos utilizados em cada aplicação estática

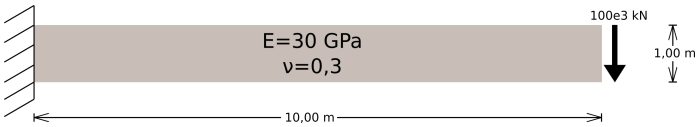
Aplicações	Precisão dupla								Precisão simples	
	CPU				GPU				GPU	
	C++	Eigen	OpenMP	MATLAB	GPU1	GPU2	GPU3	MATLAB	GPU2	GPU3
Viga	x	x	x		x	x	x			
Solo			x	x	x	x	x	x	x	x

Fonte: O Autor, 2020.

5.1.1 Viga - Estado plano de tensões

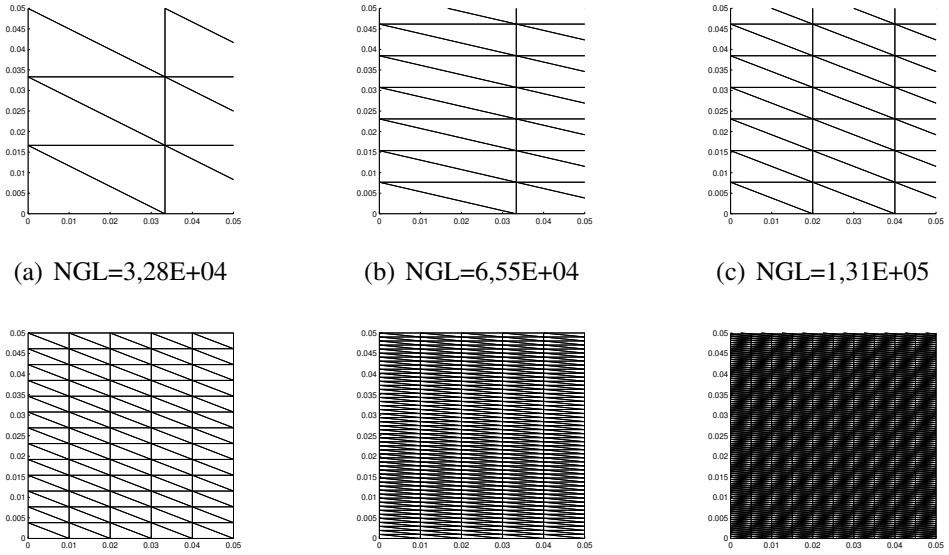
Na primeira aplicação, foram calculados os deslocamentos de uma viga engastada no bordo esquerdo e livre no bordo direito, devido a aplicação de uma carga pontual de $100\,10^3$ kN no bordo livre. Detalhes da geometria da viga e do material que é composta podem ser vistos na Tabela 1 e na Figura 8. Pelo fato de termos o estado plano de tensões, a matriz de elasticidade utilizada é descrita na Figura 3.32. O domínio foi discretizado em oito malhas diferentes de elementos CST, os detalhes de seis destas podem ser vistos na Figura 9.

Figura 8 – Detalhamento do problema da viga



Fonte: O Autor, 2020.

Figura 9 – Detalhes de seis das oito malhas utilizadas



(a) NGL=3,28E+04 (b) NGL=6,55E+04 (c) NGL=1,31E+05

(d) NGL=5,24E+05 (e) NGL=2,10E+06 (f) NGL=7,99E+06

Fonte: O Autor, 2020.

Esta aplicação foi simulada inteiramente na CPU, para a comparação, e com a montagem do problema na CPU e a resolução na GPU. Todos os três algoritmos utilizam pontos flutuantes

de precisão dupla nesta aplicação. Na CPU, foram comparados os tempos de montagem da matriz global e de resolução, para as cinco primeiras malhas, utilizando as bibliotecas básicas e utilizando a Biblioteca Eigen. Esta comparação pode ser vista na Tabela 3. Na tolerância do método foi considerado um erro de 10^{-5}.

Tabela 3 – Tempo de construção da matriz global e resolução da viga

NGL	Construção-CPU(s)	Construção-CPU Eigen(s)	Resolução-CPU(s)	Resolução-CPU Eigen(s)
3,28E+04	abaixo de 1	abaixo de 1	2	3
6,55E+04	abaixo de 1	abaixo de 1	10	11
1,31E+05	abaixo de 1	1	24	23
5,24E+05	abaixo de 1	2	181	190
1,05E+06	1	5	692	728

Fonte: O Autor, 2020.

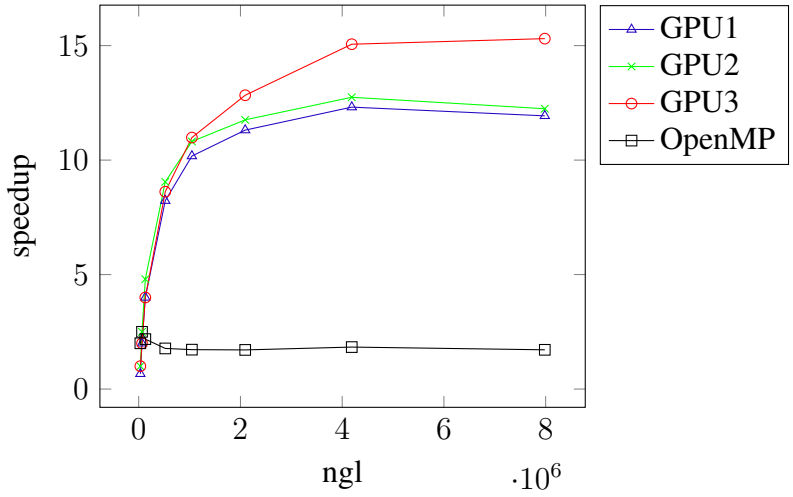
Vemos que a montagem é mais otimizada quando foram utilizadas as bibliotecas básicas, pois a biblioteca Eigen possui uma montagem de matriz esparsa mais genérica. Em relação a resolução, o desempenho no uso de ambos os algoritmos é bastante semelhante. Podem ser vistos na Tabela 4 os tempos de resolução e na Figura 10 os *speedups*, em relação ao algoritmo na CPU sequencial, alcançados pelos algoritmos utilizados na GPU para a resolução em todas as malhas. Na tolerância do método foi considerado um erro de 10^{-5}.

Tabela 4 – Tempo de resolução da viga

NGL	Iterações	CPU(s)	OpenMP(s)	GPU1(s)	GPU2(s)	GPU3(s)
3,28E+04	2890	2	1	3	2	2
6,55E+04	5506	10	4	5	4	5
1,31E+05	5753	24	11	6	5	6
5,24E+05	11341	181	102	22	20	21
1,05E+06	21435	692	402	68	64	63
2,10E+06	42232	2669	1561	236	227	208
4,19E+06	42428	5467	2918	444	429	363
7,99E+06	79744	18304	10677	1534	1495	1196

Fonte: O Autor, 2020.

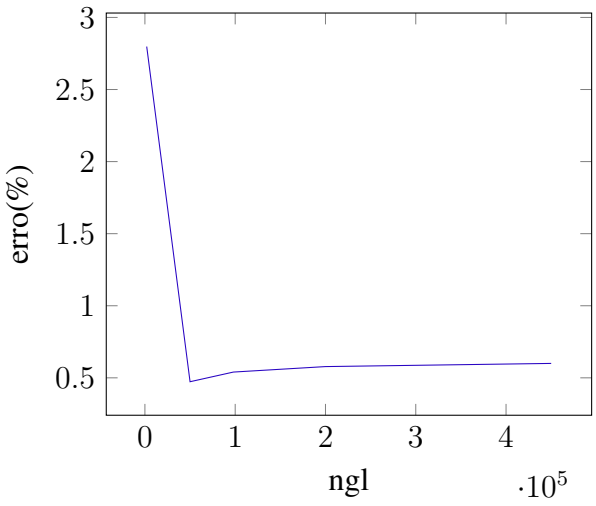
Figura 10 – *Speedups* alcançados na resolução da viga estática.



Fonte: O Autor, 2020.

A utilização de toda a CPU através do OpenMP levou a um aumento de até 2 vezes na velocidade. Já na GPU, o primeiro algoritmo, apesar de ser apenas uma utilização direta do algoritmo de gradientes conjugados passado para a GPU, leva a um aumento de até 12,3 vezes na velocidade em relação à resolução em um núcleo da CPU e 7 vezes nos 6 núcleos. Com um pequeno rearranjo esse aumento de velocidade se eleva um pouco e chega a 12,7 vezes em um núcleo e 7,1 vezes nos 6 núcleos. Ou seja, com um melhor rearranjo do acesso a memória é possível obter melhores desempenhos. O uso de memória compartilhada acelera o acesso a memória, pois, além de ser mais próxima de onde o cálculo está sendo realizado, possui uma latência maior no seu acesso. Com isso, foi possível obter aumentos na velocidade de até 15,3 vezes em um núcleo e 8,9 vezes nos 6 núcleos. A convergência do método é vista na Figura 11, comparando-se o resultado do deslocamento da extremidade livre com a resposta analítica.

Figura 11 – Erro em relação a resposta analítica

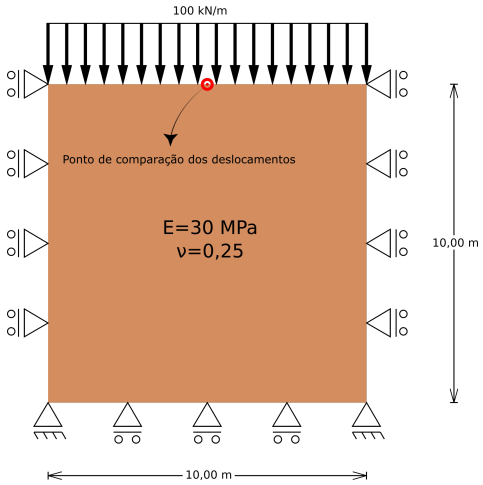


Fonte: O Autor, 2020.

5.1.2 Maciço de solo - Estado plano de deformações

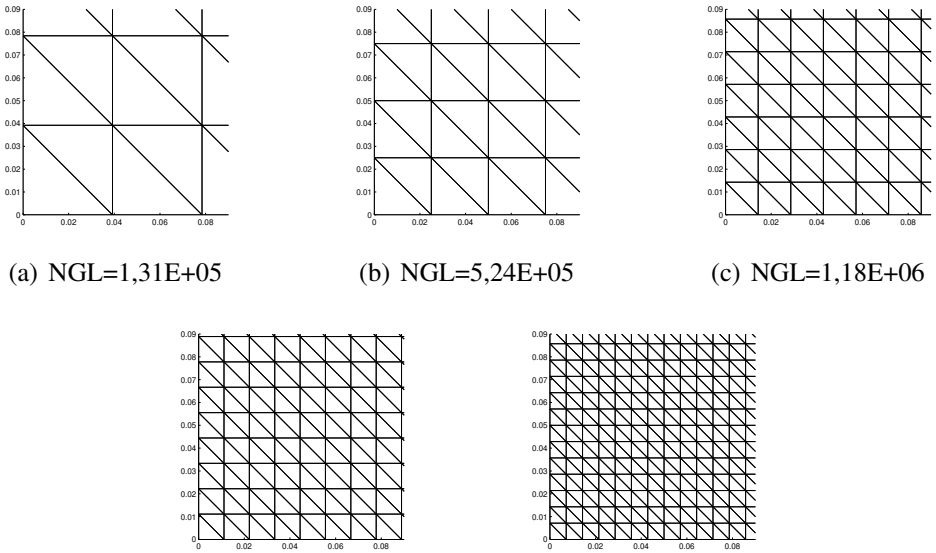
Nesta aplicação foram calculados os deslocamentos provenientes da aplicação de uma carga de 100 kN/m sobre um maciço de solo quadrado. Detalhes das dimensões do domínio e do material que é composto podem ser vistos na Tabela 1 e na Figura 12. A matriz de elasticidade utilizada é descrita pela Figura 3.33. As forças de corpo não foram consideradas. O domínio foi discretizado em cinco malhas diferentes de elementos CST, detalhes destas podem ser vistos na Figura 13.

Figura 12 – Detalhamento do problema do bloco de solo



Fonte: O Autor, 2020.

Figura 13 – Detalhes das malhas utilizadas



(a) NGL=1,31E+05 (b) NGL=5,24E+05 (c) NGL=1,18E+06

(d) NGL=2,10E+06 (e) NGL=4,72E+06

Fonte: O Autor, 2020.

Assim como a anterior, esta aplicação foi simulada inteiramente na CPU, para a com-

paração, utilizando pontos flutuantes de precisão dupla. Na GPU, foram utilizados os segundo e terceiro algoritmos da aplicação anterior, $GPU2$ e $GPU3$ respectivamente, e o MATLAB. Os algoritmos em C++ também foram considerados utilizando pontos flutuantes de precisão simples. Para uma melhor comparação dos *speedups* alcançados, os algoritmos que utilizam precisão simples foram forçados a utilizar o mesmo número de iterações que os de precisão dupla, para o mesmo número de graus de liberdade. Além disso, foi comparado o valor obtido para o deslocamento do ponto central do lado superior, como mostrado na Figura 12, para todas as malhas. Com isso, pretende-se verificar o benefício de utilizar a precisão simples, mas também o prejuízo na precisão da solução. A montagem da matriz de rigidez global foi feita em CPU tanto na linguagem C++ quanto no MATLAB, através da função *sparse*, a Tabela 5 mostra o tempo gasto nesta montagem em todas as malhas:

Tabela 5 – Tempos de construção da matriz global em CPU

NGL	C++(s)	MATLAB(s)
1,31E+05	1	1
5,24E+05	1	2
1,18E+06	1	4
2,10E+06	2	7
4,72E+06	4	17

Fonte: O Autor, 2020.

Devido ao fato da construção da matriz de rigidez global ser feita de maneira mais otimizada para o problema em C++, a velocidade do algoritmo é maior que a utilização do MATLAB, que apesar de estar vetorizado possui uma função mais genérica. Esse é um ponto positivo em optar por fazer seus próprios algoritmos. Em relação aos tempos de resolução do sistema, na Tabela 6 podem ser vistos os tempos para todos os algoritmos na linguagem C++ utilizados, na Tabela 7 os tempos no MATLAB com a utilização da função *pcg*, tanto em GPU quanto em CPU, e na Tabela 8 os tempos de uma iteração nos algoritmos utilizados na GPU. Em todos os algoritmos, sejam em C++ ou MATLAB, foi considerada a mesma quantidade de iterações para o mesmo NGL. As matrizes esparsas no MATLAB só podem ser de precisão dupla, logo não foi possível obter resultados no MATLAB utilizando precisão simples. Na tolerância do método foi considerado um erro de 10^{-5} para os algoritmos de precisão dupla.

Tabela 6 – Tempos de resolução do maciço de solo na linguagem C++

NGL	Iterações	OpenMP(s)	GPU2 PD(s)	GPU2 PS(s)	GPU3 PD(s)	GPU3 PS(s)
1,31E+05	1084	2	1	1	1	1
5,24E+05	2097	17	3	2	4	3
1,18E+06	3062	57	10	6	10	6
2,10E+06	3975	133	21	12	19	12
4,72E+06	5735	440	61	37	53	31

Fonte: O Autor, 2020.

Tabela 7 – Tempos de resolução do maciço de solo no MATLAB

NGL	Iterações	MATLAB-CPU(s)	MATLAB-GPU(s)
1,31E+05	1084	3	1
5,24E+05	2097	23	2
1,18E+06	3062	78	4
2,10E+06	3975	176	9
4,72E+06	5735	575	25

Fonte: O Autor, 2020.

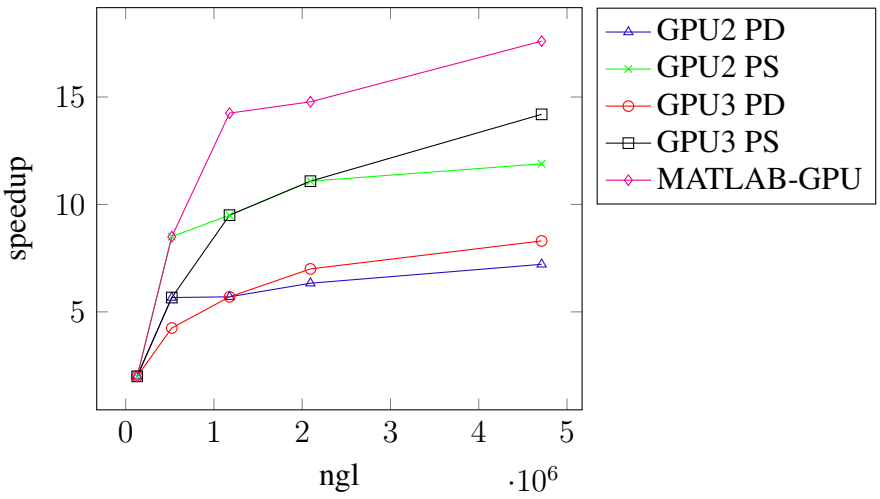
Tabela 8 – Tempo levado em cada iteração na GPU

NGL	GPU2 PD(s)	GPU2 PS(s)	GPU3 PD(s)	GPU3 PS(s)	MATLAB-GPU(s)
1,31E+05	9,23E-04	9,23E-04	9,23E-04	9,23E-04	4,97E-04
5,24E+05	1,43E-03	9,54E-04	1,91E-03	1,43E-03	8,20E-04
1,18E+06	3,27E-03	1,96E-03	3,27E-03	1,96E-03	1,37E-03
2,10E+06	5,28E-03	3,02E-03	4,78E-03	3,02E-03	2,16E-03
4,72E+06	1,06E-02	6,45E-03	9,24E-03	5,41E-03	4,29E-03

Fonte: O Autor, 2020.

A utilização da GPU em C++ trouxe uma grande redução no tempo de solução, principalmente com o uso de precisão simples. Apesar dos algoritmos utilizados na GPU em C++ serem mais rápidos do que o MATLAB em CPU, são um pouco mais lentos quando o MATLAB faz uso da GPU. Essa comparação de velocidade é melhor vista na Figura 14, onde os *speedups* de todos os algoritmos estão relacionados ao tempo do algoritmo na CPU em C++ utilizando o OpenMP:

Figura 14 – *Speedups* alcançados na resolução do maciço de solo estático



Fonte: O Autor, 2020.

Analisando agora os *speedups* em relação ao uso de toda a capacidade da CPU é possível ver, para a precisão dupla, que forma muito próximos aos da aplicação anterior. O algoritmo com

apenas o rearranjo da operação - para a otimização ao acesso a memória - acelerou a aplicação em 7,4 vezes. Já o algoritmo que otimiza o acesso a memória - através da utilização da memória compartilhada - chega a acelerar a aplicação em até 9,1 vezes.

Com a utilização de precisão simples foi possível obter maiores *speedups*, porém também houve uma queda na precisão. Com a utilização do algoritmo com o rearranjo das operações chegou-se a um aumento de 12 vezes na velocidade. Enquanto com o algoritmo que usa a memória compartilhada foi possível chegar a aumentos de velocidade de até 15,4 vezes. Como visto anteriormente - com a comparação dos tempos - o MATLAB foi um pouco mais lento na CPU. Porém, foi um pouco mais rápido na GPU. Na CPU o OpenMP foi 1,2 vezes mais rápido e na GPU o MATLAB chegou a ser 18,4 vezes mais veloz. Na Tabela 9 podem ser vistos a quantidade de iterações, o valor dos deslocamentos verticais obtidos - para o ponto indicado na Figura 12 - com ambas as precisões e suas diferenças, em relação a solução obtida pela resolução direta realizada no MATLAB, para cada uma das malhas.

Tabela 9 – Diferenças relativas no deslocamento vertical do ponto indicado na Figura 12

NGL	Deslocamento PS(m)	Deslocamento PD(m)	Erro PS(%)	Erro PD(%)
1,31E+05	3,1250093E-02	3,1249992E-02	0,0002980	0,0000256
5,24E+05	3,1250298E-02	3,1249991E-02	0,0009537	0,0000286
1,18E+06	3,1250179E-02	3,1250000E-02	0,0005722	0,0000003
2,10E+06	3,1251334E-02	3,1250004E-02	0,0042677	0,0000136
4,72E+06	3,1250283E-02	3,1249998E-02	0,0009060	0,0000053

Fonte: O Autor, 2020.

A depender da malha utilizada, as diferenças relativas podem ser consideráveis. Nas aplicações feitas, os erros devido a utilização de pontos flutuantes de precisão simples foram abaixo de 1E-3% para as cinco malhas utilizadas. Nas aplicações com pontos flutuantes de precisão dupla os erros sempre ficaram abaixo de 1E-4%.

5.2 Aplicações transientes

Para a avaliação das implementações dos algoritmos de integração no tempo foram realizadas duas aplicações. A primeira consiste na simulação de um enchimento e um esvaziamento rápido de uma barragem em arco. Já a segunda consiste na simulação de exploração de um reservatório de petróleo.

Na CPU foram utilizadas as bibliotecas básicas do C++ em um algoritmo sequencial, outro utilizando a estrutura OpenMP - para a utilização de todos os núcleos do processador - e por fim, outro no MATLAB. Já na GPU o algoritmo em C++ utilizado foi o algoritmo que se saiu melhor nas aplicações anteriores (GPU3), e o algoritmo em MATLAB. Para a integração no tempo a primeira aplicação foi realizada tanto de forma implícita como de forma explícita, mas a

segunda só foi realizada de forma implícita. Na Tabela 10 é possível ver os algoritmos utilizados em cada aplicação.

Tabela 10 – Algoritmos utilizados em cada aplicação dinâmica

Aplicações	Precisão dupla			
	CPU			GPU
	C++	OpenMP	MATLAB	GPUC++
Barragem		x		x
Reservatório	x	x	x	x

Fonte: O Autor, 2020.

Na aplicação da barragem foram utilizados os métodos implícito e explícito de integração do tempo, já na aplicação do reservatório de petróleo foi utilizado apenas o método implícito. Isso é justificado pelo tamanho do domínio no tempo ser muito superior na simulação do reservatório que na simulação da barragem. Na tabela 11 temos alguns dados do domínio e das propriedades mecânicas utilizados nas simulações.

Tabela 11 – Dados utilizados em cada aplicação dinâmica

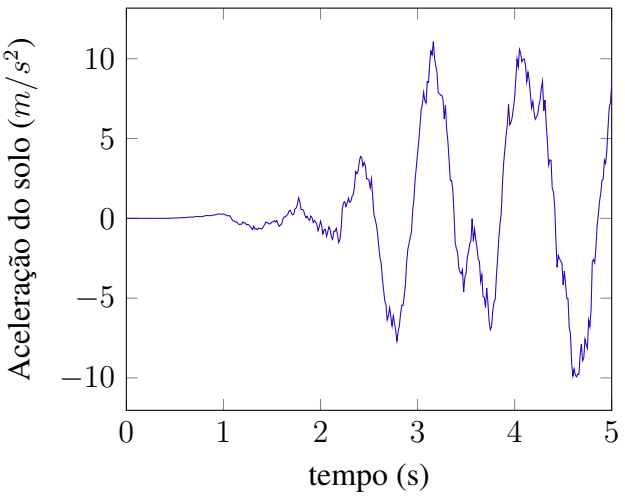
Aplicações	Seção horizontal(m x m)	Altura(m)	E	ν
Barragem	Figura 16	200	27 GPa	0,3
Reservatório	670.56 x 670.56	60.96	68.95 MPa	0,3

Fonte: O Autor, 2020.

5.2.1 Simulação de barragem submetida a um sismo

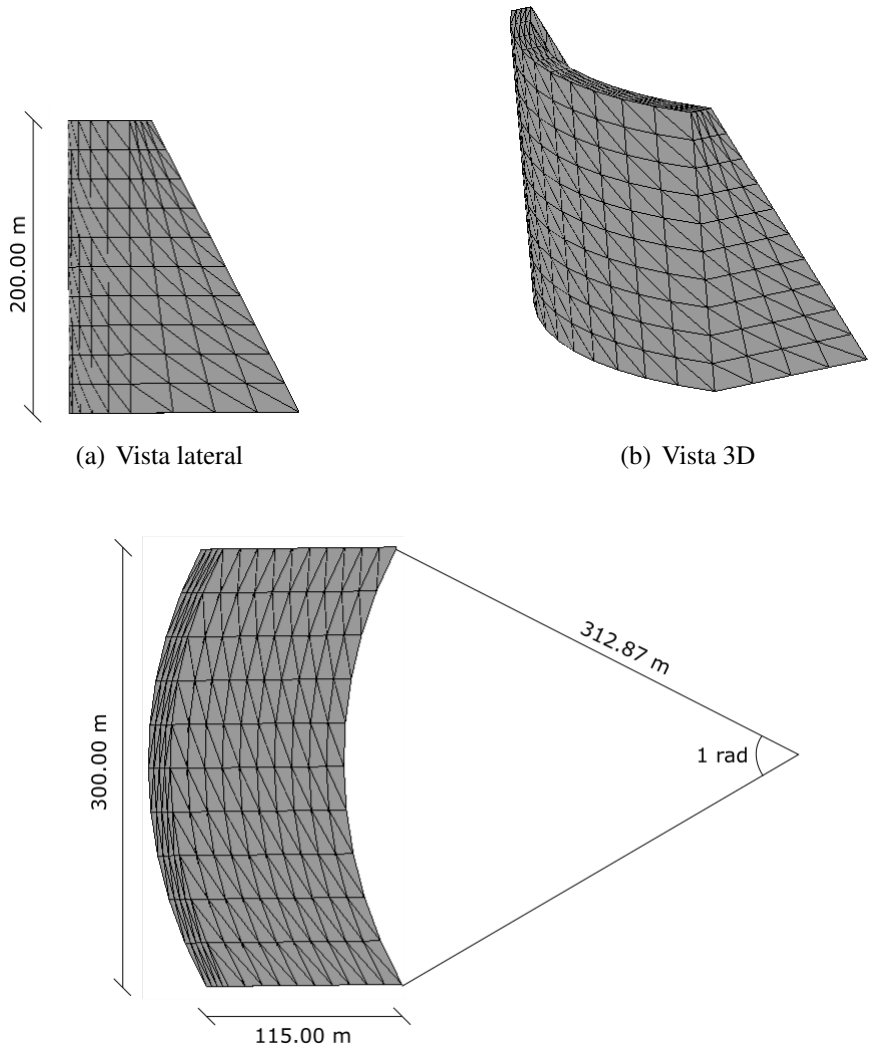
Foram simulados os $5\ s$ iniciais de um terremoto com o acelerograma, gerado artificialmente, visto na Figura 15 em uma barragem em arco para a avaliação dos algoritmos. A barragem possui $200\ m$ de altura, $15\ m$ de espessura na crista, $115\ m$ de espessura na base e sua projeção horizontal tem um comprimento de $300\ m$. O arco é gerado por um raio de $312.87\ m$ com um ângulo de $1\ rad$ de forma que a extensão da barragem é igual a $312.87\ m$. A forma da barragem pode ser vista na Figura 16. Os deslocamentos da face inferior e das faces laterais foram considerados bloqueados.

Figura 15 – Acelerograma utilizado



Fonte: O Autor, 2020.

Figura 16 – Geometria da barragem



(a) Vista lateral (b) Vista 3D

(c) Vista superior

Fonte: O Autor, 2020.

Foi considerado o módulo de elasticidade igual a $27\ GPa$, o coeficiente de Poisson igual a 0.3 e a densidade do material igual a $2500kg/m^3$. Para os algoritmos de integração implícita foi utilizado um passo de tempo igual a $0.0125\ s$ e para os algoritmos de integração explícita o passo de tempo dependeu da maior frequência do sistema, sendo calculado pela Equação 4.7 e pode ser visto na tabela 12. O domínio foi discretizado em cinco malhas de elementos tetraédricos de 4 nós com o número de graus de liberdade variando de 300 mil até 2.5 milhões.

Tabela 12 – Passos de tempo do método explícito

Malha (ngl)	3.37E+05	7.52+05	1.49E+06	1.86E+06	2.55E+06
Δt (s)	7.794E-09	5.196E-09	3.625E-09	3.243E-09	2.756E-09

Fonte: O Autor, 2020.

Para os algoritmos de integração implícita a simulação foi realizada completamente,

pois foram exigidos apenas 1024 passos de tempo. Já para os algoritmos de integração explícita seriam necessários mais de 100 milhões de passos de tempo, devido as elevadas frequências. Então, foram realizados 10001 avanços temporais para a comparação. Os tempos de cada um dos algoritmos para cada malha pode ser visto na tabela 13.

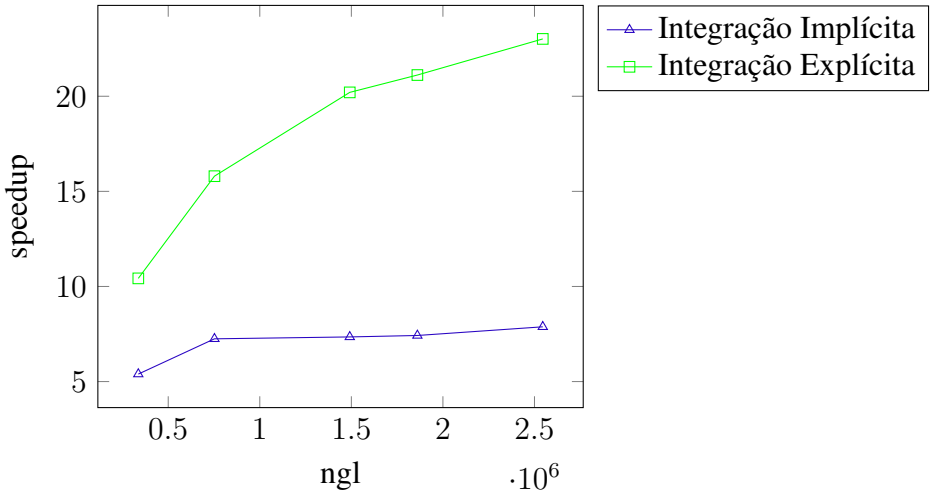
Tabela 13 – Tempo de simulação da barragem

NGL	C++			
	Integração Implícita		Integração Explícita (10001 passos de tempo)	
	OpenMP(h)	GPU(h)	OpenMP(s)	GPU(s)
3.37E+05	6.70	1.24	110.82	10.63
7.52+05	23.06	3.18	252.14	15.96
1.49E+06	52.70	7.17	504.28	24.96
1.86E+06	73.97	9.96	629,29	29.66
2.55E+06	111.23	14.12	878.91	38.19

Fonte: O Autor, 2020.

Os *speedups* observados nas implementações em GPU em relação aos algoritmos utilizando OpenMP podem ser vistos na Figura 17 para cada um dos métodos. Com a integração explícita, a GPU chega a acelerar em até 23 vezes a simulação, enquanto que com a integração implícita a aceleração foi de 7.8 vezes. Apesar de obter uma maior aceleração, o tamanho do passo de tempo é muito inferior, fazendo com que leve muito mais tempo para a simulação completa. Para a integração explícita ser vantajosa será necessário que a implementação exija um passo de tempo pequeno o suficiente, dessa forma a integração implícita usará um número de passos de tempo tão grande quanto a integração explícita.

Figura 17 – *Speedups* alcançados na simulação da barragem

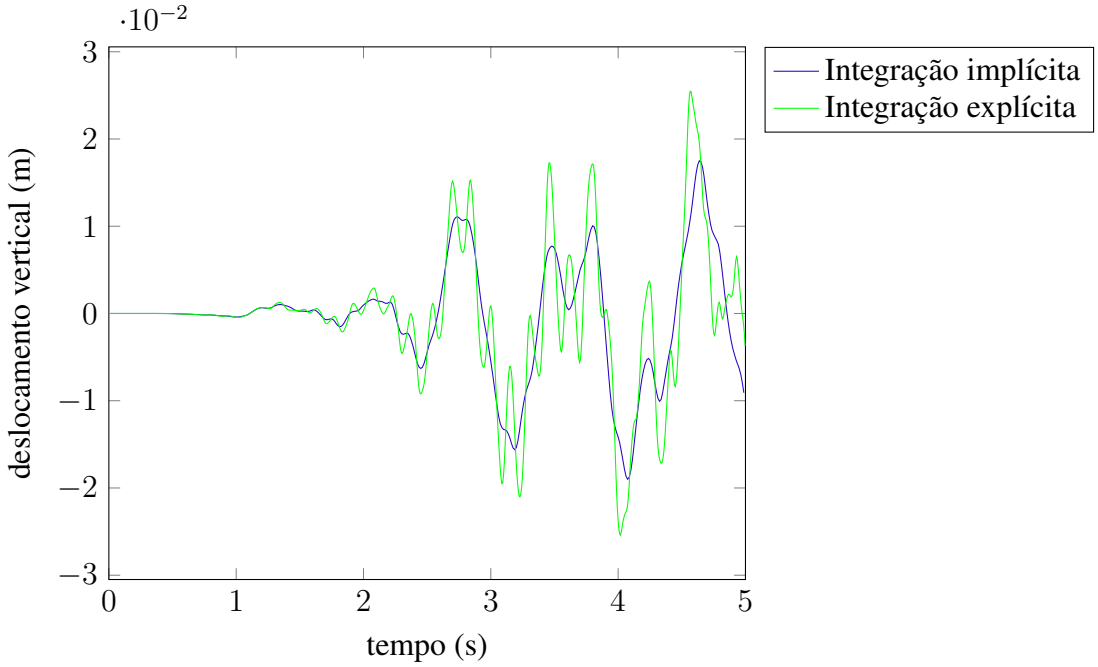


Fonte: O Autor, 2020.

Para a comparação dos resultados entre os dois métodos de integração, o deslocamento do nó central da face superior foi monitorado ao longo do tempo. Para a simulação completa da

barragem com o método explícito de integração, esta foi feita com uma malha mais grossa. Os deslocamentos podem ser vistos na Figura 18.

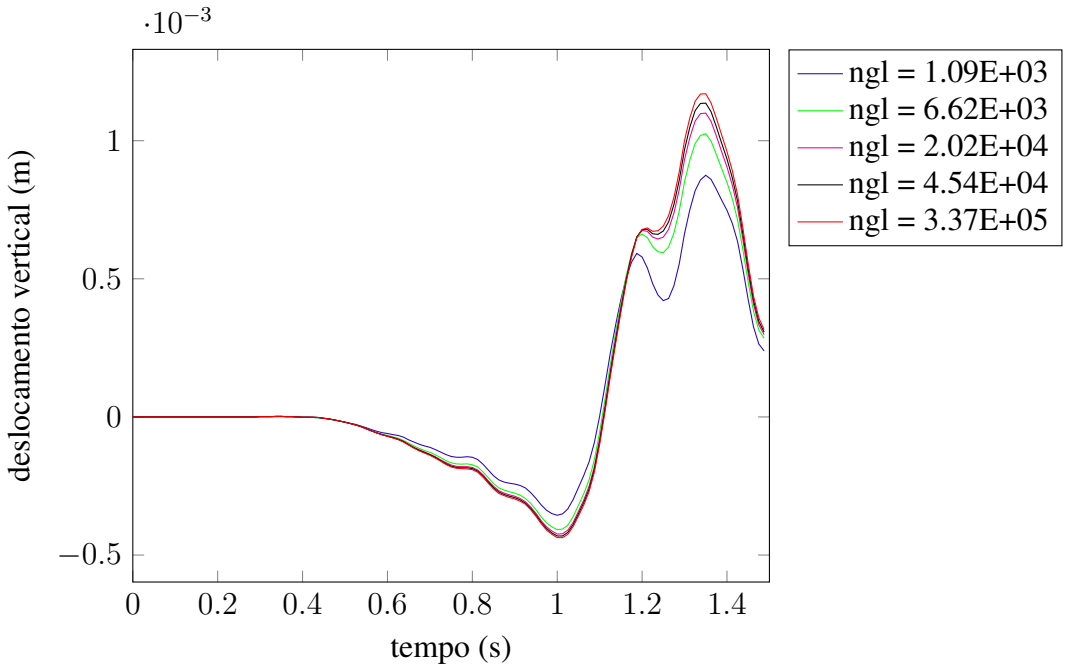
Figura 18 – Comparação dos métodos de integração



Fonte: O Autor, 2020.

Pela comparação dos deslocamentos, vemos que o método implícito possui uma transição mais suave entre os deslocamentos e que o método explícito utilizado necessita de melhoria. A Figura 19 mostra a convergência do método explícito para os primeiros 1.5 segundos da simulação do sismo.

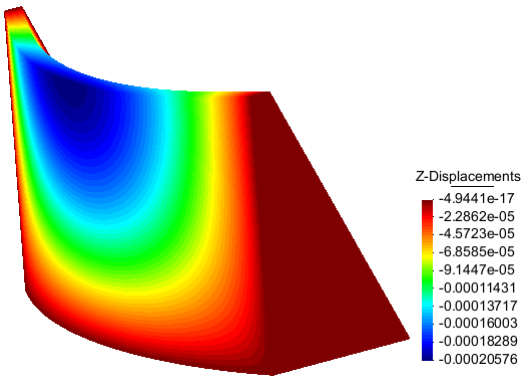
Figura 19 – Convergência do método implícito



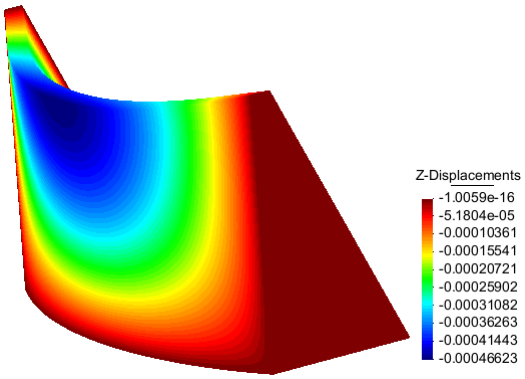
Fonte: O Autor, 2020.

Para a malha mais refinada, utilizada no teste de convergência (3.37E+05 graus de liberdade), foi ilustrado o campo de deslocamentos em 3 passos de tempo como pode ser visto na Figura 20.

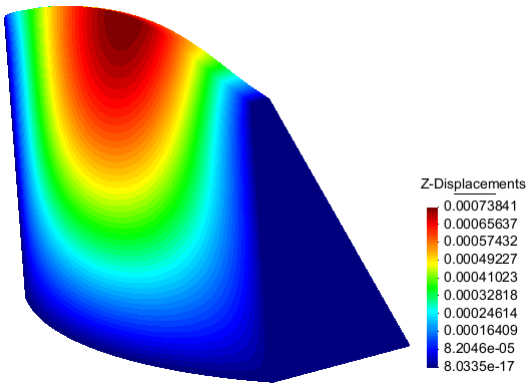
Figura 20 – Campo de deslocamentos (m) em 3 passos de tempo



(a) tempo = 0.8 s



(b) tempo = 1.0 s



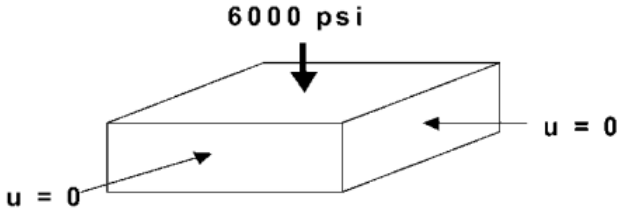
(c) tempo = 1.2 s

Fonte: O Autor, 2020.

5.2.2 Simulação hidromecânica unidirecional em reservatório

Para a avaliação dos algoritmos na simulação de reservatórios, foi escolhido o problema 1 utilizado por Dean et al. (2006). Este problema consiste em um reservatório de domínio prismático de base quadrada de lado $670.56\ m$ e altura $60.96\ m$. Possuindo uma permeabilidade horizontal de $4.9346E-14\ m^2$, vertical de $4.9346E-15\ m^2$ e uma porosidade de 20%. O fluido é monofásico, possuindo viscosidade de $1E-9\ MPa.s$, compressibilidade de $0.05381\ m^2/MN$ e peso específico igual a $1000\ kg/m^3$. A pressão inicial do fluido no topo do reservatório é de $20.68\ MPa$ com um gradiente hidráulico de $8.81E-3\ MPa/m$ até o fundo. O meio poroso possui módulo de elasticidade de $68.95\ MPa$, coeficiente de Poisson igual a 0.3 e coeficiente de Biot igual a 1. As faces verticais e a face inferior do reservatório possuem os deslocamentos travados (Figura 21). Um poço de produção é instalado no centro do reservatório com uma vazão de 15 mil barris de petróleo por dia ao longo de 500 dias. A evolução é calculada a um passo de tempo de 25 dias.

Figura 21 – Detalhamento do problema do reservatório



Fonte: DEAN et al., 2006.

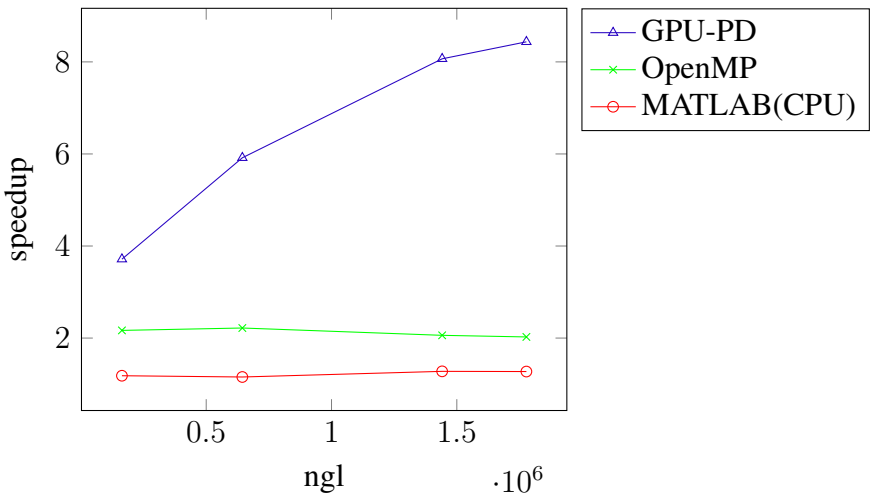
Como pode ser visto na Tabela 10, esta aplicação foi simulada a partir de quatro algoritmos com cinco malhas de quantidades de graus de liberdade variando de 160 mil a 2.7 milhões. Todas utilizando o elemento tetraédrico de quatro nós. O tempo que cada algoritmo leva para a simulação pode ser visto na Tabela 14, e os *speedups* em relação ao algoritmo C++ sequencial na CPU pode ser visto na Figura 22.

Tabela 14 – Tempo de simulação do reservatório

NGL	C++			MATLAB
	CPU(s)	OpenMP(s)	GPU(s)	CPU(s)
1.64E+05	26	12	7	22
6.44E+05	142	68	24	123
1.44E+06	484	235	60	379
1.78E+06	658	325	78	517
2.77E+06			138	1479

Fonte: O Autor, 2020.

Figura 22 – *Speedups* alcançados na simulação de reservatório

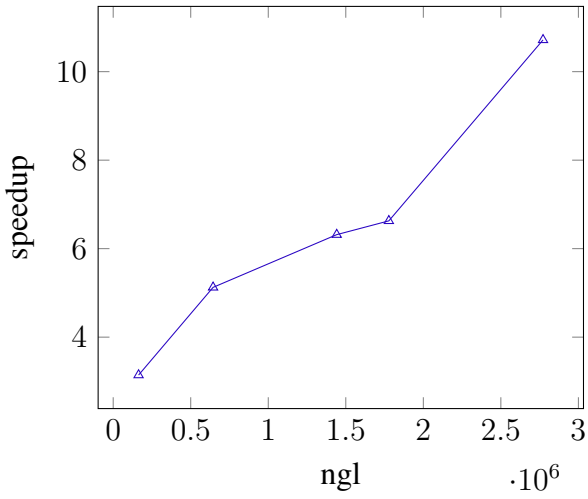


Fonte: O Autor, 2020.

O algoritmo na GPU alcançou um *speedup* acima de 8 vezes, enquanto o algoritmo OpenMP manteve um *speedup* de aproximadamente 2 vezes. O MATLAB na CPU foi um pouco

mais rápido que o sequencial em C++. Por problema na alocação na memória, não foi possível realizar a simulação com o maior número de graus de liberdade em C++ na CPU. Na Figura 23 se encontram os *speedups* de todas as malhas simuladas na GPU em relação ao algoritmo no MATLAB em CPU.

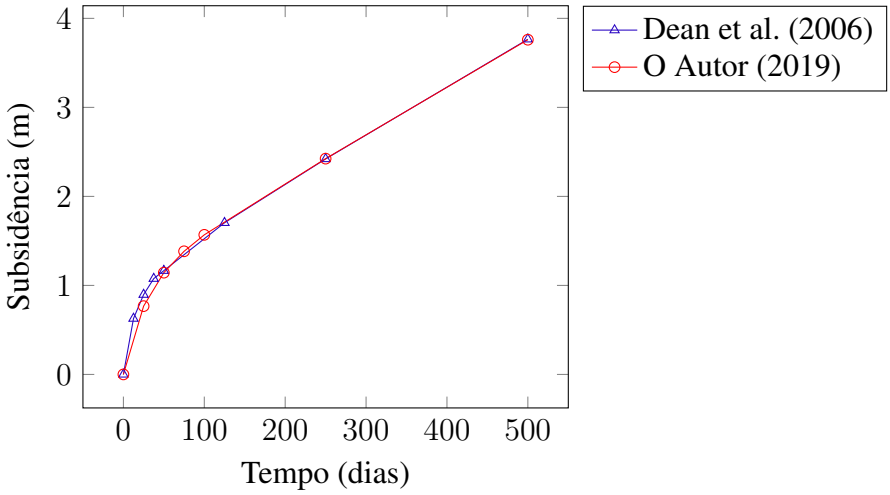
Figura 23 – *Speedup* do algoritmo GPU-PD em relacção ao MATLAB(CPU)



Fonte: O Autor, 2020.

A subsidência no nó central da face superior do reservatório ao longo do tempo é apresentada por Dean et al. (2006) para uma malha com 1210 elementos hexaédricos, e foi comparada com os resultados obtidos com uma malha de 2160 elementos tetraédricos. Esta comparação pode ser vista na Figura 24.

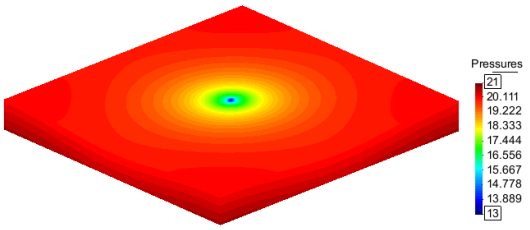
Figura 24 – Subsidência ao longo do tempo no nó central da face superior do reservatório



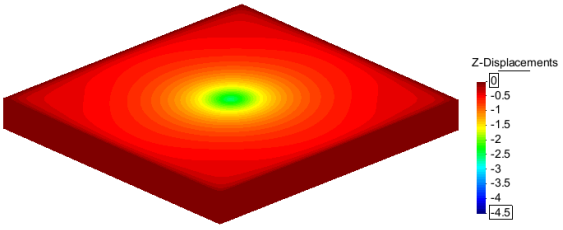
Fonte: O Autor, 2020.

Para uma malha mais refinada que a usada na comparação da Figura 24 (1.64E+05 graus de liberdade), foi ilustrado o campo de deslocamentos e pressões de 3 passos de tempo como pode ser visto na Figura 25.

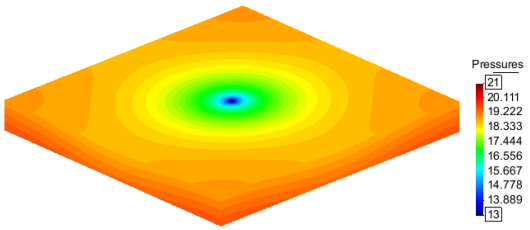
Figura 25 – Campos de deslocamentos e pressões em 3 passos de tempo



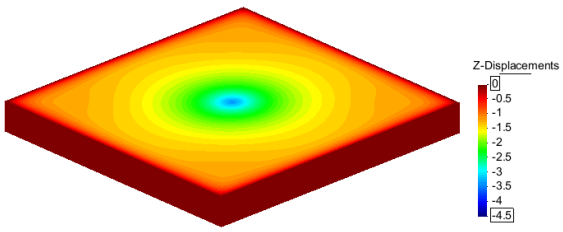
(a) Pressões (MPa) com 100 dias



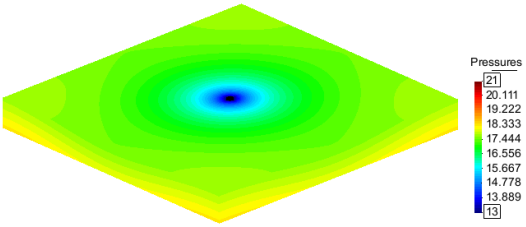
(b) Deslocamentos (m) com 100 dias



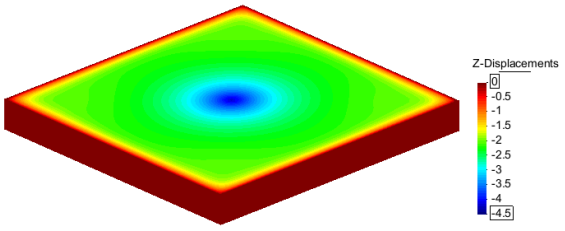
(c) Pressões (MPa) com 250 dias



(d) Deslocamentos (m) com 250 dias



(e) Pressões (MPa) com 400 dias



(f) Deslocamentos (m) com 400 dias

Fonte: O Autor, 2020.

6 CONSIDERAÇÕES FINAIS

Neste trabalho, implementações do método dos gradientes conjugados e de esquemas de integração direta implícita e explícita em GPU são propostas na linguagem C++ em conjunto com a linguagem CUDA para a aceleração de simulações mecânicas. Para a demonstração de sua eficácia forma realizadas duas simulações estáticas e duas simulações dinâmicas. Na primeira simulação estática, demonstra-se a importância de uma programação voltada a GPU, onde um pequeno rearranjo do algoritmo proporciona, quando comparado a utilização de toda a capacidade da CPU, uma pequena melhoria, passando a ser 7,1 vezes mais rápida, e a utilização de artifícios como a memória compartilhada garantem um melhor desempenho, chegando a ser 8,9 vezes mais rápida. E que, apesar de a programação ser mais simples com a utilização de bibliotecas externas como a Eigen, a utilização das bibliotecas básicas permite a otimização dos códigos para problemas específicos. Na segunda, foi demonstrado que a computação na GPU alcança maiores velocidades com o uso de pontos flutuantes de precisão simples, sendo 1,7 vezes mais rápida do que com a precisão dupla. O código em MATLAB na GPU obteve velocidades superiores ao algoritmo implementado, isso mostra o quanto o código implementado pode ser otimizado. A criação de um algoritmo próprio ainda traz o benefício de utilizar precisão simples para a aceleração do processamento. Também foi visto que, a depender da distorção da malha, a utilização de precisão simples pode trazer grandes erros para a solução. Nas simulações dinâmicas foram testadas duas formas de integração, uma explícita e outra implícita. Na primeira, a integração explícita teve uma maior aceleração quando implementada na linguagem CUDA, alcançando *speedups* de até 20 vezes, enquanto a integração implícita alcançou um *speedup* próximo de 8 vezes. Apesar disso, a integração explícita tem o empecilho de o passo de tempo ser dependente da maior frequência do sistema para garantir a estabilidade do método, o que limita muito o avanço no tempo. Na segunda aplicação, a implementação em GPU do método implícito alcançou *speedups* acima de 10 vezes em relação ao MATLAB na CPU. O algoritmo com a utilização da estrutura OpemMP também se mostrou mais vantajosa que o algoritmo no MATLAB em CPU.

6.1 Trabalhos Futuros

Para problemas de escalas ainda maiores, a utilização de uma única GPU pode não ser suficiente devido ao problema de memória. Por isso, uma importante área de trabalho futuro é a integração de múltiplas GPUs em problemas ainda maiores. Além disto, podem ser exploradas outras estratégias, como a redução de ordem da matriz na GPU. Em nossos trabalhos futuros serão implementados em GPU: o cálculo das matrizes dos elementos e a montagem da matriz de rigidez global; o método dos gradientes conjugados com precisão mista; métodos de integração explícitos e implícitos mais complexos; e métodos de redução de ordem. Testes serão realizados para comparar o desempenho com as bibliotecas de álgebra linear em CUDA já existentes e os

códigos continuarão sendo otimizados para maiores acelerações serem alcançadas.

REFERÊNCIAS

AGÊNCIA NACIONAL DO PETRÓLEO, GÁS NATURAL E BIOCOMBUSTÍVEIS. *Manual de comunicação de incidentes de exploração e produção de petróleo e gás natural*. [S.l.], 2017. 67 p. Citado na página 12.

ALBUQUERQUE, R. A. de C. *Simulação de fluxo e tensões em reservatórios aplicada a casos reais*. Tese (Doutorado) — Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, Brasil, 2015. Disponível em: <https://www.maxwell.vrac.puc-rio.br/colecao.php?strSecao=resultado&nrSeq=24660@1>. Citado na página 22.

ASSIS, D. C. A. de. *Simulação hidromecânica 3D em análogo de reservatório carbonático naturalmente fraturado*. Tese (Doutorado) — Universidade Federal de Pernambuco, Recife, Brasil, 2019. Disponível em: <https://repositorio.ufpe.br/handle/123456789/34081>. Citado na página 22.

BARROS, R. C. et al. *Apostila de dinâmica das estruturas e engenharia sísmica*. [S.l.: s.n.], 2002. Citado na página 47.

BARTEZZAGHI, A. et al. An explicit dynamics gpu structural solver for thin shell finite elements. *Computers & Structures*, v. 154, p. 29–40, 2015. Citado 2 vezes nas páginas 13 e 19.

BATHE, K.-J. *Finite Element Procedures in Structural Analisys*. [S.l.]: Pretice Hall, 1996. Citado na página 23.

BEAR, J. *Dynamics of fluids in porous media*. [S.l.]: New York: American Elsevier, 1972. Citado na página 32.

BELYTSCHCHKO, T.; LIU, W. K.; MORAN, B. *Nonlinear Finites Elements for Continua ans Structures*. [S.l.]: Wiley, 2000. Citado na página 48.

BIZZO, Y. F. *Avaliação dos efeitos das propriedades dos fluidos e das rochas na simulação geomecânica de reservatórios do campo de namorado*. Tese (Doutorado) — Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, Brasil, 2017. Disponível em: <https://www.maxwell.vrac.puc-rio.br/colecao.php?strSecao=resultado&nrSeq=30274@1>. Citado na página 22.

CHENG, J. R.; GEN, M. Accelerating genetc algorithims with gpu computing: A selective overview. *Computers & Industrial Engineering*, v. 128, p. 514–525, 2019. Citado na página 19.

CHOPRA, A. K.; WANG, J.-T. Earthquake response of arch dams to spatiallyvarying ground motion. *Wiley InterScience*, v. 2, 2009. Citado na página 20.

CIFU, S. Projeto estrutural de barragens de concreto. *Concreto & Construções*, v. 63, 2011. Citado na página 12.

CLARK, M. A. et al. Solving lattice qcd systems of equations using mixed precision solvers on gpus. *Computer Physics Communications*, v. 181, p. 1517–1528, Setembro 2010. Citado na página 18.

COLLANTES, F. C. P. *Comportamento dinâmico de uma barragem de rejeitos com considerações de ameaças símicas*. Dissertação (Mestrado) — Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2015. Citado na página 21.

CONELL, L. D. Coupled flow and geomechanical process during gas production from coal seams. *International Journal of Coal Geology*, v. 79, p. 18–28, Julho 2009. Citado na página 20.

CRAIG JR., R. R.; KURDILLA, A. J. *Fundamentals of Structural Dynamics*. 2. ed. [S.l.]: Wiley, 2006. Citado na página 47.

DEAN, R. et al. A comparison of techniques for coupling porous flow and geomechanics. *SPE Journal*, v. 11, 2006. Citado 3 vezes nas páginas 67, 68 e 69.

FAJARDO, R. I. C. *Previsão numérica do comportamento dinâmico da barragem de Breapampa no Peru*. Dissertação (Mestrado) — Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2015. Citado na página 21.

FALCãO, F. de O. L. *Efeitos geomecânicos na simulação de reservatórios de petróleo*. Dissertação (Mestrado) — Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2002. Citado na página 22.

GUTSTEIN, D. *Projeto de estrutura de barragens de gravidade de concreto por meio de métodos computacionais: visão geral e metodologia*. Tese (Doutorado) — Universidade Federal de Santa Catarina, Florianópolis, Brasil, 2011. Disponível em: <https://repositorio.ufsc.br/handle/123456789/95338>. Citado na página 21.

HELFENSTEIN, R.; KOKO, J. Parallel preconditioned conjugate gradient algorithm on gpu. *Journal of Computational and Applied Mathematics*, v. 236, p. 3584–3590, Junho 2012. Citado na página 18.

HOWARD, M. P.; Z., P. A.; NIKOUBASHMAN, A. Efficient mesoscale hydrodynamics: Multiparticle collision dynamics with massively parallel gpu acceleration. *Computer Physics Communications*, v. 230, p. 10–20, Setembro 2018. Citado na página 18.

KIM, W.; LEE, J. H. An improved explicit time integration method for linear and nonlinear structural dynamics. *Computers & Structures*, v. 206, 2018. Citado na página 23.

KIM, Y.; PARK, Y. Cpu-gpu architecture for active noise control. *Applied Acoustics*, v. 153, p. 1–13, 2019. Citado na página 19.

KWON, Y. W.; BANG, H. *The Finite Elemente Method using MATLAB*. [S.l.]: CRC Press, 1996. Citado 3 vezes nas páginas 12, 27 e 29.

LIMA, R. O. *Avaliação de técnicas de acoplamento parcial entre simuladores geomecânico e de fluxo*. Dissertação (Mestrado) — Universidade Federal de Pernambuco, Recife, 2019. Citado na página 22.

LOAYZA, F. H. *Modelagem do comportamento pós-sismo de uma barragem de rejeito*. Dissertação (Mestrado) — Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2009. Citado na página 21.

LOGAN, D. L. *A First Course in The Finite Elemente Method*. [S.l.]: Cengage Learning, 2012. Citado 2 vezes nas páginas 12 e 24.

MA, Y. et al. Optimizing sparse tensor times matrix on gpus. *Journal of Parallel and Distributed Computing*, v. 129, p. 99–109, 2019. Citado na página 18.

MARKELE, J. Finite element vibration and dynamic response analysis of engineering structures: a bibliography (1994-1998). *Shock and Vibration*, v. 7, 2000. Citado na página 12.

MEDIAWIKI. *Eigen*. 2019. Http://eigen.tuxfamily.org/index.php?title=Main_Page. Access date: 28 June 2019. Citado na página 41.

MELO, D. A. F. de. *Metodologia numérica para obtenção das propriedades equivalentes de reservatórios naturalmente fraturados utilizando elementos finitos com descontinuidades incorporadas*. Dissertação (Mestrado) — Universidade Federal de Pernambuco, Recife, 2019. Citado na página 22.

MENDES, N. B. *Um estudo de propagação de ondas e sismo na análise dinâmica acoplada a barragem em arco - reservatório - fundação*. Tese (Doutorado) — Faculdade de Tecnologia, Universidade de Brasília, Brasília, Brasil, 2018. Disponível em: <https://sucupira.capes.gov.br/sucupira/public/consultas/coleta/trabalhoConclusao/viewTrabalhoConclusao.jsf?popup=true&id_trabalho=6916938>. Citado na página 21.

MENDES, N. B.; PEDROSO, L. J. Um estudo do acoplamento barragem em arco - reservatório sob ação de um sismo. *Revista Interdisciplinar de Pesquisa em Engenharia*, 2016. Citado na página 21.

MINKOFF, S. et al. Staggered in time coupling of reservoir flow simulation and geomechanical deformation: Step 1 - one-way coupling. 02 1970. Citado na página 20.

MINKOFF, S. E. et al. Coupled fluid flow and geomechanical deformation modeling. *Journal of Petroleum Science and Engineering*, v. 38, p. 37–56, Maio 2003. Citado na página 20.

MORADI, M.; SHAMLOO, A.; DEZFULI, A. D. A sequential implicit discrete fracture model for three-dimensional coupled flow-geomechanics problems in naturally fractured porous media. *Journal of Petroleum Science and Engineering*, v. 150, p. 312–322, Fevereiro 2017. Citado na página 20.

NILSSON, K.; TORNBERG, F. On blowdown analysis with efficient and reliable direct time integration methods for wave propagation and fluid-structure-interaction response. *Computers & Structures*, v. 216, 2019. Citado na página 20.

NOH, G.; BATHE, K.-J. An explicit time integration scheme for the analysis of wave propagations. *Computers & Structures*, v. 129, 2013. Citado na página 23.

NVIDIA. *NVIDIA Tesla P100*. 2016. Https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf. Citado 2 vezes nas páginas 42 e 47.

PIKLE, N. K.; SATHE, S. R.; VYAVHARE, A. Y. Gpgpu-based parallel computing applied in the fem using the conjugate gradient algorithm: a review. *Sãdhanã*, v. 43, 2018. Citado na página 18.

RIGHETTO, G. L. *Desenvolvimento e aplicação de um esquema de acoplamento termo-hidro-mecânico-químico iterativo visando o armazenamento geológico de CO2*. Tese (Doutorado) — Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, Brasil, 2018. Disponível em: <https://www.maxwell.vrac.puc-rio.br/colecao.php?strSecao=resultado&nrSeq=33840@1>. Citado na página 22.

RODRIGUES, T. *Análise numérica de reativação de zonas de falhas geológicas devido à produção de petróleo*. Dissertação (Mestrado) — Universidade Federal de Pernambuco, Recife, 2019. Citado na página 22.

RUTQVIST, J. An overview of tough-based geomechanics models. *Computers & Geosciences*, v. 108, p. 56–63, Novembro 2017. Citado na página 12.

SANTOS, A. B. dos. *Um estudo dinâmico de uma barragem em arco gravidade considerando as juntas de contração*. Dissertação (Mestrado) — Universidade Tecnológica Federal do Paraná, Curitiba, jun. 2018. Citado na página 21.

SEABRA, G. S. *Simulações hidromecânicas parcialmente acopladas de um reservatório carbonático da bacia de campos*. Tese (Doutorado) — Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, Brasil, 2017. Disponível em: <https://www.maxwell.vrac.puc-rio.br/colecao.php?strSecao=resultado&nrSeq=29837@1>. Citado na página 22.

SHAO, G.-F. et al. Gpu-based dpso algorithm for structural optimization of pt-co bimetallic nanoparticles. *Physics Letters A*, v. 383, p. 3123–3133, Setembro 2019. Citado na página 19.

SILVA, A. C. da. *Acoplamento hidro-mecânico em reservatórios naturalmente fraturados com dupla porosidade e dupla permeabilidade utilizando a técnica do stress Split*. Dissertação (Mestrado) — Universidade Federal de Pernambuco, Recife, 2018. Citado na página 22.

TEIXEIRA, W. et al. *Decifrando a Terra*. 1. ed. [S.l.]: Oficina de Textos, 2000. Citado na página 11.

TRIANA, O. Y. D. *Development of a surrogate multiscale reservoir simulator coupled with geomechanics*. Tese (Doutorado) — Faculdade de Engenharia Mecânica e Instituto de Geociências, Campinas, Brasil, 2017. Disponível em: <http://repositorio.unicamp.br/handle/REPOSIP/331516>. Citado na página 21.

VILAÇA, S. F.; GARCIA, L. F. T. *Introdução à Teoria da Elasticidade*. [S.l.]: COPPE/UFRJ, 1998. Citado 2 vezes nas páginas 25 e 26.

WEN, W. B. et al. A comparative study of three composite implicit schemes on structural dynamic and wave propagation analysis. *Computers & Structures*, v. 190, 2017. Citado na página 23.

WEN, W. B. et al. A novel sub-step composite implicit time integration scheme for structural dynamics. *Computers & Structures*, v. 182, 2017. Citado na página 23.

XIA, G.; LIN, C.-L. An unstructured finite volume approach for structural dynamics in response to fluid motions. *Computers & Structures*, v. 86, 2008. Citado na página 20.

YANG, Y.-S.; YANG, C.-M.; HSIEH, T.-J. Gpu parallelization of an object-oriented nonlinear dynamic structural analysis platform. *Simulation Modelling Practice and Theory*, v. 40, p. 112–121, 2014. Citado na página 19.

ZHANG, H. M.; XING, Y. F. An unstructured finite volume approach for structural dynamics in response to fluid motions. *Computers & Structures*, v. 221, 2019. Citado na página 23.

ZIENKIEWICZ, O. C.; TAYLOR, R. L. *The Finite Elemente Method*. 5. ed. [S.l.]: Butterworth-Heinemann, 2000. v. 2. Citado 4 vezes nas páginas 12, 13, 19 e 51.

APÊNDICE A – CÓDIGO EM C++

Abaixo são apresentadas as funções utilizadas em C++ para a multiplicação de matriz esparsa, no formato CSR, com vetor e para operações entre vetores elemento por elemento. Estes são algoritmos necessários na utilização do método dos gradientes conjugados.

Tabela 15 – Funções em C++

Função	Processamento	Descrição	Dependências
OpVetVet	CPU	Realiza operações algébricas entre dois vetores	-
MultEspVet	CPU	Multiplicação matriz esparsa (CRS) vetor	-

Fonte: O Autor, 2020.

Função com operações entre vetores na CPU:

```
void OpVetVet(double* Va, double* Vb, long int dim, char c, ...
double lamb, double* Vc)
{
        //Operacao entre vetores elemento por elemento
        double soma = 0;
        if (c == '+')
        {
                for (long int i = 0; i < dim; i++)
                {
                        Vc[i] = Va[i] + lamb * Vb[i];
                }
        }
        else if (c == '-')
        {
                for (long int i = 0; i < dim; i++)
                {
                        Vc[i] = Va[i] - lamb * Vb[i];
                }
        }
        else if (c == '*')
        {
                for (long int i = 0; i < dim; i++)
                {
                        Vc[i] = Va[i] * (lamb * Vb[i]);
                }
        }
        else if (c == '/')
        {
                for (long int i = 0; i < dim; i++)
                {
                        Vc[i] = Va[i] / (lamb * Vb[i]);
                }
```

```
        }
}
```

Função com multiplicação matriz esparsa no formato CRS com vetor na CPU:

```
void MultEspVet(double* VVal, long int* VCol, long int* vlin, ...
double* d, long int dim, double* F)
{
        // Multiplicacao de Matriz esparsa por vetor
        for (long int i = 0; i < dim; i++)
        {
                double soma = 0;
                for (long int j = vlin[i]; j < vlin[i + 1]; j++)
                {
                        soma += VVal[j] * d[VCol[j]];
                }
                F[i] = soma;
        }
}
```

APÊNDICE B – CÓDIGO EM CUDA

Abaixo são apresentadas as funções em CUDA com as mesmas finalidades das funções apresentadas no Apêndice anterior, a diferença é que elas são processadas na GPU. O tipo $global$ da função especifica que é uma função a ser processada na GPU, mas chamada na CPU. Além deste, vale mencionar também o tipo $device$, que é processado e chamado na GPU.

Tabela 16 – Funções em C++

Função	Processamento	Descrição	Dependências
OpVetGPU	GPU	Realiza operações algébricas entre dois vetores	-
MultEspVetELL	GPU	Multiplicação matriz esparsa (ELLPACK) vetor	-
DotMemComp	CPU	Produto interno entre vetores	sDot1 e sDot2
sDot1	GPU	Realiza soma de dois vetores	-
sDot2	GPU	Soma posições de um vetor	-

Fonte: O Autor, 2020.

Temos também os contadores $bolckIdx$, que lê as posições de cada bloco na grade nas 3 dimensões; $bolckDim$, que lê o tamanho dos blocos nas 3 dimensões; e $threadIdx$, que lê as posições de cada $thread$ no bloco nas 3 dimensões.

Função com operações entre vetores na GPU:

```
    __global__ void OpVetGPU(double* Va, double* Vb, long int dim,...
char c, double *a, double *b, double* Vc)
{
        //Operacao entre vetores elemento por elemento na GPU
        //
        //Va e Vb com tamanho dim, entre eles e realizada a operacao
    //'c'. O vetor Vb e multiplicado pela constante a e
        // dividido pela constante b. O resultado e armazenado no
    //vetor Vc.


        long int i = blockIdx.x*blockDim.x + threadIdx.x;
        if (i < dim)
        {
                double lamb = a[0] / b[0];
                if (c == '+')
                {
                        Vc[i] = Va[i] + lamb * Vb[i];
                }
                else if (c == '-')
                {
                        Vc[i] = Va[i] - lamb * Vb[i];
                }
                else if (c == '*')
```

```
                {
                        Vc[i] = Va[i] * (lamb * Vb[i]);
                }
                else if (c == '/')
                {
                        Vc[i] = Va[i] / (lamb * Vb[i]);
                }
        }
}
```

Função com multiplicação de matriz esparsa no formato ELLPACK com vetor na GPU:

```
    __global__ void MultEspVetELL(double* devVal, long int* devCol, ...
double* D, long int nelnn, double* aux)
{
        // Multiplicacao de Matriz esparsa (ELLPACK) por vetor
        __shared__ double Bs[NT];
        long int i = blockIdx.x*blockDim.x + threadIdx.x;
        if (i < nelnn) {
                long int a = devCol[i], b = i % 16, c = 2;
                Bs[threadIdx.x] = D[a];
                Bs[threadIdx.x] = devVal[i] * Bs[threadIdx.x];
                __syncthreads();

                for (long int j = 0; j < 4; j++)
                {
                        if (!(threadIdx.x % c) && (i + c / 2 < nelnn)) {
                                Bs[threadIdx.x] += Bs[threadIdx.x + c / 2];
                                c *= 2;
                        }
                        __syncthreads();
                }
                if (!b) {
                        aux[i / 16] = Bs[threadIdx.x];
                }
        }
}
```

Na função que realiza a multiplicação da matriz esparsa no formato ELLPACK com vetor na GPU, cada conjunto de 16 *threads* é responsável pela multiplicação de uma linha da matriz com o vetor, pelo armazenamento dos valores resultantes da multiplicação na memória compartilhada e pela soma destes valores.

O especificador $shared$ indica que a variável será alocada na memória compartilhada da GPU, a constante NT é predefinida no código. O comando $\_\_syncthreads$ sincroniza todos os $threads$ no bloco para não haver uma acesso a memória compartilhada até todos os $threads$ a modificarem. Existe outro comando de sincronização, o comando $cudaDeviceSynchronize$ é utilizado na CPU para sincronizar todos os núcleos de uma GPU para qua na utilização de um

próximo $kernel$ a memória global acessada tenha sido completamente modificada pela $kernel$ anterior.

A função $cudaMallocManaged$ aloca memória que pode ser acessada tanto pela GPU quanto pela CPU, e a função $cudaFree$ libera a posição de memória alocada. A função $DotMemComp$ é apresentada abaixo junto com outras funções que ela necessita. Realiza faz o produto dos vetores de entrada dois a dois e armazena dos vetores de saída, de forma que o primeiro elemento de cada vetor de saída é o somatório de todos os elementos.

```
void DotMemComp(double *a, double *b, long int dim, long int nt, double *d)
{
    // Produto interno entre os vetores a e b.
        long int nb = dim / nt + 1, dim2 = dim, mult = nt;

        sDot1 << <nb, nt >> > (a, b, dim, d);
        cudaThreadSynchronize();

        while (nb > 1)
        {
                dim2 = nb;
                nb = nb / nt + 1;
                sDot2 << <nb, nt >> > (d, dim2, mult);

                mult *= nt;
        }
        cudaThreadSynchronize();
}

__global__ void sDot1(double*a, double*b, long int dim, double*d)
{
        long int it = threadIdx.x, i = blockIdx.x*blockDim.x + it, m = 2;
        __shared__ double resp[NT];
        if (i < dim) {


                resp[it] = a[i] * b[i];
                __syncthreads();

                for (int j = 0; j < NN; j++) {
                        //*
                        if ((!(it%m)) && (i < dim - m / 2)) {
                                resp[it] += resp[it + m / 2];
                        }//*/
                        __syncthreads();

                        m *= 2;
```

```
                }
                d[i] = resp[it];
        }
}


__global__ void sDot2(double*d, long int dim, long int mult)
{
        long int it = threadIdx.x, i = blockIdx.x*blockDim.x + it, m = 2;
        __shared__ double resp[NT];
        if (i < dim) {
                resp[it] = d[i*mult];
                __syncthreads();

                for (int j = 0; j < NN; j++) {
                        if ((!(it%m)) && (i < dim - m / 2)) {
                                resp[it] += resp[it + m / 2];
                        }
                        __syncthreads();
                        m *= 2;
                }
                d[i*mult] = resp[it];

        }
}
```

APÊNDICE C – CÓDIGO EM MATLAB

O código em MATLAB é bastante simples e as operações são realizadas de forma vetorizada. O tempo de montagem é contabilizado apenas na função *sparse*.

A aplicação das condições de contorno é realizada de forma vetorizada, onde as matrizes dx e dy possuem a primeira coluna sendo os nós com os deslocamentos prescritos nos eixos x e y respectivamente, e a segunda os valores destes deslocamentos.

A solução no MATLAB pelo método dos gradientes conjugados é bastante simples tanto na CPU quanto na GPU. Tudo isso pode ser visto no código abaixo.

```
% CODIGO EM MATLAB UTILIZADO PARA A RESOLUCAO DA VIGA ENGASTADA
%
%Autor = Cicero Vitor Chaves Junior
%Data = 01/03/2020

close all
clear all
clc

% Propriedades dos materiais
E=30e9;
nu=.3;
P=-1e7;
t=1;

% Criacao da malha
lx=10;ly=1;
nx=1000;ny=500;
tic
coord=zeros((nx+1)*(ny+1),2);
conec=zeros(2*nx*ny,2);

coord(:,1)=kron(ones(ny+1,1),linspace(0,lx,nx+1)');
coord(:,2)=kron(linspace(0,ly,ny+1)',ones(nx+1,1));

conecaux1=(1:(nx+1)*(ny+1)-(nx+1))';
conecaux1((nx+1):(nx+1):end)=[];
conecaux2=(1:(nx+1)*(ny+1)-(nx+1))';
conecaux2(1:(nx+1):end)=[];
conecaux3=((nx+1)+1:(nx+1)*(ny+1))';
conecaux3((nx+1):(nx+1):end)=[];
conecaux4=((nx+1)+1:(nx+1)*(ny+1))';
conecaux4(1:(nx+1):end)=[];
```

```
conec(1:2:end,1)=conecaux1;
conec(1:2:end,2)=conecaux2;
conec(1:2:end,3)=conecaux3;
conec(2:2:end,1)=conecaux2;
conec(2:2:end,2)=conecaux4;
conec(2:2:end,3)=conecaux3;

clear conecaux1 conecaux2 conecaux3 conecaux4

CMalha=toc
tic
[nn,dim]=size(coord);
[nel,nne]=size(conec);


if length(E)==1; E=ones(nel,1).*E; end
if length(nu)==1; nu=ones(nel,1).*nu; end



% Condicoes de contorno
F=zeros(nn*dim,1);
dx=[];
dy=[];
nccarga=[];
for i=1:nn
    if coord(i,1)==lx; nccarga=[nccarga,i]; end
    if coord(i,1)==0; dx=[dx;i 0]; dy=[dy;i 0]; end
end
F(nccarga*dim)=P/length(nccarga);
%% Montagem das matrizes

x=[coord(conec(:,1),1) coord(conec(:,2),1) coord(conec(:,3),1)];
y=[coord(conec(:,1),2) coord(conec(:,2),2) coord(conec(:,3),2)];

desPresc=find(coord(:,1)==0);
forPresc=find(coord(:,1)==lx);

clear coord

Ar=((x(:,1)-x(:,2)).*(y(:,1)+y(:,2))+...
    (x(:,2)-x(:,3)).*(y(:,2)+y(:,3))+...
    (x(:,3)-x(:,1)).*(y(:,3)+y(:,1))./2;

D=zeros(3,3,nel);

D(1,1,:)=E./(1-nu.^2).*Ar;
```

```
D(2,2,:)=E./(1-nu.^2).*Ar;
D(3,3,:)=E./(1-nu.^2).*(1-nu)./2.*Ar;
D(1,2,:)=E./(1-nu.^2).*nu.*Ar;
D(2,1,:)=D(1,2,:);

% B    Calculado ja multiplicado pela area;
B=zeros(6,3,nel);

B(1,1,:)=(y(:,2)-y(:,3))./(2.*Ar); B(3,1,:)=(y(:,3)-y(:,1))./(2.*Ar);
B(5,1,:)=(y(:,1)-y(:,2))./(2.*Ar); B(2,2,:)=(x(:,3)-x(:,2))./(2.*Ar);
B(4,2,:)=(x(:,1)-x(:,3))./(2.*Ar); B(6,2,:)=(x(:,2)-x(:,1))./(2.*Ar);
B(1:2:6,3,:)=B(2:2:6,2,:); B(2:2:6,3,:)=B(1:2:6,1,:);

clear Ar nu E x y

ke=MultTensor3(MultTensor3(B,D),B);

clear B D

% GL
linha=zeros(nne*dim,1,nel);
coluna=zeros(1,nne*dim,nel);

vcoorg=[conec(:,1)'.*dim-1;conec(:,1)'.*dim;...
        conec(:,2)'.*dim-1;conec(:,2)'.*dim;...
        conec(:,3)'.*dim-1;conec(:,3)'.*dim];

clear conec

linha(:,1,:)=vcoorg;
linha(:,2,:)=vcoorg;
linha(:,3,:)=vcoorg;
linha(:,4:6,:)=linha(:,1:3,:);
coluna(1,:,:)=vcoorg;
coluna(2,:,:)=vcoorg;
coluna(3,:,:)=vcoorg;
coluna(4:6,:,:)=coluna(1:3,:,:);

clear vcoorg
tic
Kg=sparse(linha(:),coluna(:),ke(:));
toc
clear linha coluna ke

MGlobal=toc

% Deslocamentos em y
```

```
ndyp=size(dy,1);
if ndyp~=0
    Kg(dy(:,1)*dim,:)=sparse([],[],[],ndyp,nn*dim);
    Kg(dy(:,1)*dim,dy(:,1)*dim)=speye(ndyp);
    F(dy(:,1)*dim)=dy(:,2);
end

% Deslocamentos em x
ndxp=size(dx,1);
if ndxp~=0
    Kg(dx(:,1)*dim-1,:)=sparse([],[],[],ndxp,nn*dim);
    Kg(dx(:,1)*dim-1,dx(:,1)*dim-1)=speye(ndxp);
    F(dx(:,1)*dim-1)=dx(:,2);
end
return
fprintf("Solucoes\n");


%% Resolucao em CPU
tic
u2=pcg(Kg,F,1e-5,1e5);
Tpcg=toc


%% Resolucao em GPU
%
% Para a resolucao em GPU a unica coisa necessaria e alocar a matriz e o
% vetor na GPU

Kg_d=gpuArray(Kg);
F_d=gpuArray(F);

tic
u3=pcg(Kg_d,F_d,1e-5,1e5);
TpcgGPU=toc
```