XL CILAMCE
IBERO-LATIN AMERICAN CONGRESS ON COMPUTATIONAL METHODS IN ENGINEERING
NOVEMBER 11-14, 2019
Praiamar Natal Hotel & Convention
Natal, RN-BRAZIL

ACCELERATION OF FEM MECHANICAL SIMULATIONS WITH A PARALLEL IMPLEMENTATION IN GPU

Cicero Vitor Chaves Junior
Paulo Marcelo Vieira Ribeiro
cicerovtjr@gmail.com
paulo.vribeiro@ufpe.br
Department of Civil Engineering, Federal University of Pernambuco-UFPE
Avenida da Arquitetura s/n, 50740-550, Pernambuco/Recife, Brasil

Abstract. Finite element simulations with the use of large scale models are becoming more frequent. 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. CPU solutions using the Eigen linear algebra library are also developed, and 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, Linear elasticity

1 Introdução

Simulações de grande escala em elementos finitos são utilizadas em muitas áreas da engenharia, e quando são feitas análises não lineares ou transientes há custos computacionais consideráveis. Em análises não lineares geralmente são aplicadas técnicas iterativas, onde são realizadas análises lineares a cada iteração de forma incremental [1]. Da mesma forma, análises transientes também costumam ser feitas como análises estáticas consecutivas a cada avanço temporal. Para materiais com não-linearidade muito expressiva e mudanças de estado bruscas no tempo, esses avanços temporais, ou incrementos, são muito pequenos. Isto leva o tempo de simulação a ser muito alto, já que a resolução estática e linear a cada iteração também demanda tempo considerável para problemas de grande escala.

Há poucos anos, houve uma mudança no desenvolvimento de estruturas que permitem a computação paralela. As unidades de processamento gráfico (GPUs) que antes eram projetadas apenas com foco no mercado de jogos e vídeos passaram a ter arquiteturas que as permitissem serem utilizadas também 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 [2]. Os dispositivos de GPU atuais são construídos com até centenas de processadores, de maneira que pode-se realizar uma grande quantidade de operações de pontos flutuantes paralelamente; um esquema é mostrado na Fig. 1. Com isso, simulações numéricas podem ser feitas de forma mais eficiente.

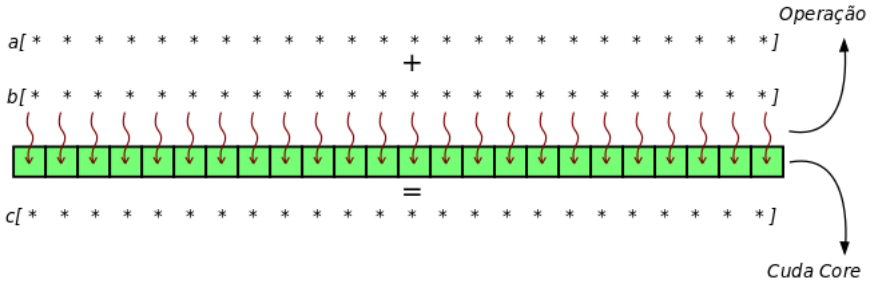


Figura 1. Distribuição das operações paralelas em uma soma de vetores na GPU

GPUs vêm sendo utilizadas não só para simulações em MEF como também para outros propósitos. Cheng e Gen [3] em 2019 deram uma visão geral de vários tipos de algoritmos genéticos implementados em GPU selecionados desde 2010. Ma et al. [4] implementaram em GPU um algoritmo que realiza a multiplicação tensor-matriz. Utilizando precisão simples, eles conseguiram acelerar suas aplicações, em média, em até 30 vezes. Kim e Park [5] utilizaram uma arquitetura mista CPU-GPU para controle de ruído, obtiveram diferentes aumentos de velocidades em cada parte do processo analisado de forma que o processo inteiro foi acelerado em 82 vezes. Yang et al. [6] 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.

Os speedups alcançados dependem da GPU utilizada, além disso também dependem de que tipo de precisão foi utilizada nas operações. Originalmente, as GPUs foram feitas para operações de pontos flutuantes de precisão simples e, ainda hoje, são mais eficientes para tal. Nas GPUs mais equilibradas, estas operações são 2 vezes mais lentas com precisão dupla em relação a precisão simples, porém, existem GPUs, mesmo modernas, onde as operações com precisão dupla chegam a ser 32 vezes mais lentas [7], alguns detalhes podem ser vistos na Tabela 1. Na grande maioria das simulações em MEF, a precisão dupla se faz necessária, logo, a escolha de uma GPU projetada com o foco em operações de precisão dupla é importante. Também é necessário pensar em que partes do código é realmente necessária a precisão

dupla, para se aproveitar a total velocidade da GPU em partes onde a precisão simples é suficiente.

Tabela 1. Algumas especificações de GPUs [7]

GPU	GK110	GM200	GP100
SMs	15	24	56
FP32 CUDA Cores	2880	3072	3584
FP64 CUDA Cores	960	96	1792
Memória(GB)	12	acima de 24	acima de 16

Na maioria das simulações de grande escala, o maior custo computacional está na resolução do sistema linear, logo seguido pela montagem das matrizes de rigidez dos elementos e global. Este trabalho tem como foco a resolução do sistema linear na GPU, com isso, os códigos desenvolvidos para tal podem ser reutilizados em qualquer simulação que for necessário. Além disso, é apresentado o que foi feito sequencialmente na CPU para a montagem do sistema a ser resolvido. Na seção 2 deste artigo é apresentada brevemente a formulação matemática; na seção 3 são encontradas as implementações feitas na CPU, desde a criação da malha até montagem da matriz global, e as implementações na GPU, o algoritmo de gradientes conjugados utilizado na resolução e como ele foi rearranjado para melhorar o desempenho; na seção 4 encontram-se as aplicações e os resultados obtidos; e por fim, na seção 5 estão as conclusões sobre o que foi apresentado.

2 Formulação Matemática

O problema mecânico resolvido neste artigo é o de elasticidade. Seu desenvolvimento é feito pelo sistema de equações de equilíbrio, obtido a partir do diagrama de corpo livre de um elemento infinitesimal, da relação tensão-deformação do material utilizado e das relações cinemáticas que devem ser satisfeitas em todo o domínio. Além destas, existem condições que devem ser satisfeitas no contorno [8].

Uma solução aproximada em elementos finitos é obtida com a aplicação do método dos resíduos ponderados com a escolha da formulação de Galerkin. Sendo V o domínio do problema, A seu contorno e procedendo da mesma forma que Kwon e Bang [9] obtemos a Eq. (1):

$$-\int_V \begin{pmatrix} \frac{\partial \omega_1}{\partial x}\sigma_x + \frac{\partial \omega_1}{\partial y}\tau_{xy} + \frac{\partial \omega_1}{\partial z}\tau_{zx} \\ \frac{\partial \omega_2}{\partial y}\sigma_y + \frac{\partial \omega_2}{\partial x}\tau_{xy} + \frac{\partial \omega_2}{\partial z}\tau_{yz} \\ \frac{\partial \omega_3}{\partial z}\sigma_z + \frac{\partial \omega_3}{\partial y}\tau_{yz} + \frac{\partial \omega_3}{\partial x}\tau_{zx} \end{pmatrix} dV + \int_V \begin{pmatrix} \omega_1 f_x \\ \omega_2 f_y \\ \omega_3 f_z \end{pmatrix} dV + \int_A \begin{pmatrix} \omega_1 \overline{q}_x \\ \omega_2 \overline{q}_y \\ \omega_3 \overline{q}_z \end{pmatrix} dA = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}. \quad (1)$$

Sendo σ_e a tensão normal na direção e, τ_{ed} a tensão de cisalhamento no plano e e na direção d, f_e a força de corpo na direção e e $\overline{q}_e$ a força externa na direção e. ω_n são as funções de ponderação, que para o método de Galerkin provêm das n funções de aproximação dos deslocamentos de cada uma das direções. Incluindo as relações cinemáticas e de tensão e deformação na Eq. (1) e colocando-a na forma matricial:

$$-\underbrace{\int_e \mathbf{B}^T\mathbf{D}\mathbf{B}\, dV}_{\mathbf{K}^e}\,\mathbf{d} + \underbrace{\int_e \mathbf{H}^T\mathbf{f} dV}_{\mathbf{f}^{c,e}} + \underbrace{\int_e \mathbf{H}^T\overline{\mathbf{q}} dA}_{\mathbf{f}^{q,e}} = \mathbf{0}. \quad (2)$$

Onde $\mathbf{H}$ é uma matriz correspondente às funções de forma do elemento escolhido, $\mathbf{B}$ é uma matriz resultante da multiplicação de uma matriz de operação de derivada sobre $\mathbf{H}$, $\mathbf{f}$ um vetor correspondente as forças de corpo, $\overline{\mathbf{q}}$ um vetor correspondente as forças externas e $\mathbf{D}$ conforme a Eq. (3):

$$\mathbf{D} = \frac{E}{(1-2\nu)(1+\nu)} \begin{bmatrix} 1-\nu & -\nu & -\nu & 0 & 0 & 0 \\ -\nu & 1-\nu & -\nu & 0 & 0 & 0 \\ -\nu & -\nu & 1-\nu & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1-2\nu}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1-2\nu}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1-2\nu}{2} \end{bmatrix}. \tag{3}$$

Com E definindo o módulo de elasticidade e ν o coeficiente de Poisson.
Ou seja, pra cada elemento temos a Eq. (4):

$$\mathbf{K}^e\mathbf{d} = \mathbf{f}^{c,e} + \mathbf{f}^{q,e}. \tag{4}$$

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

$$\mathbf{K}^e = \mathbf{B}^T\mathbf{D}\mathbf{B}\,V^e. \tag{5}$$

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 $\mathbf{B}$, de forma que o cálculo da matriz de rigidez será uma pequena modificação da Eq. (5):

$$\mathbf{K}^e = \mathbf{B}^T\mathbf{D}\mathbf{B}\,A^e t. \tag{6}$$

Onde temos a área A^e do elemento e sua espessura t. Já a matriz $\mathbf{D}$, caso esteja-se admitindo o estado plano de tensões, será como na Eq. (7):

$$\mathbf{D} = \frac{E}{1-\nu^2} \begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{bmatrix}. \tag{7}$$

E caso esteja-se admitindo o estado plano de deformações, será como na Eq. (8):

$$\mathbf{D} = \frac{E}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1-\nu & \nu & 0 \\ \nu & 1-\nu & 0 \\ 0 & 0 & \frac{1-2\nu}{2} \end{bmatrix}. \tag{8}$$

3 Aspectos Computacionais

3.1 Implementações em CPU

Como demonstrado na seção anterior, tudo que precisamos é calcular as matrizes de rigidez e o vetores de forças de cada elemento e agrupá-los na matriz e no vetor globais, como visto na Eq. (9), para depois resolver o sistema formado. Nesta seção, será apresentado como foram feitas as montagens das matrizes e dos vetores, globais e locais.

$$\mathbf{K}^g\mathbf{d}^g = \mathbf{f}^{c,g} + \mathbf{f}^{q,g}. \tag{9}$$

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.

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.

O formato esparso em que a matriz de rigidez global é armazenada é o CRS (Compressed Row Storage). Neste formato, 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.

$$\mathbf{M} = \begin{bmatrix} 2 & 0 & 4 & 0 \\ 0 & 7 & 9 & 0 \\ 3 & 18 & 0 & 5 \\ 0 & 20 & 0 & 0 \end{bmatrix} \longrightarrow \quad \begin{matrix} \mathbf{m}^1 = \begin{bmatrix} 2 & 4 & 7 & 9 & 3 & 18 & 5 & 20 \end{bmatrix} \\ \mathbf{m}^2 = \begin{bmatrix} 0 & 2 & 1 & 2 & 0 & 1 & 3 & 1 \end{bmatrix} \\ \mathbf{m}^3 = \begin{bmatrix} 0 & 2 & 4 & 7 & 8 \end{bmatrix} \end{matrix} . \tag{10}$$

No Algoritmo 2, vê-se como é feito o armazenamento da matriz global neste formato. 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.

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 10.
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, as multiplicação e divisão por escalar com vetores densos ou com matrizes densas pode ser realizados 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.

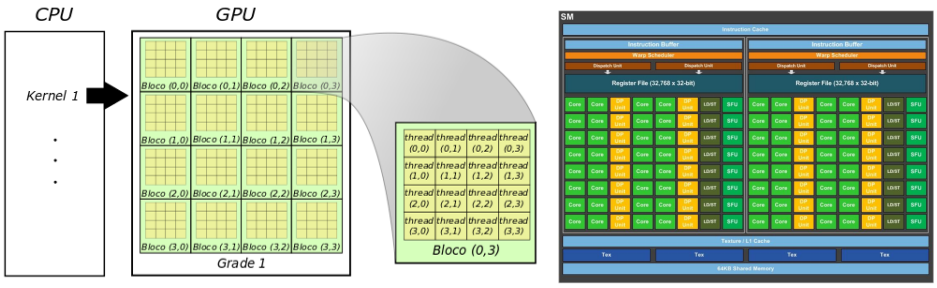
Para a manipulação de matrizes esparsas, a biblioteca Eigen permite comprimir a matriz no formato CRS. Para a montagem da matriz esparsa, esta biblioteca fornece uma outra classe chamada *Triplet* que armazena a linha, a coluna e o valor não nulo. Ao ser armazenado todos os valores das matrizes de rigidez 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 [10].

3.2 Implementações em GPU

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 Fig. 2(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 Fig. 2(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.



(a) Organização dos threads e blocos

(b) SM de um processador gráfico GP100 [7]

Figura 2. Esquemas de organização e arquitetura CUDA

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 4 e 5 demonstram o funcionamento destes kernels respectivamente.

Algoritmo 4 Multiplicação entre matriz esparsa e vetor denso na GPU

Entrada: $\mathbf{M}$ - matriz esparsa no formato CRS como demonstrado na equação 10, $\mathbf{b}$, dim - dimensão
Saída: $\mathbf{x}$
$tidx =$ 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 Operações entre vetores, elemento por elemento, exemplificado com a soma na GPU

Entrada: $\mathbf{a}$, $\mathbf{b}$, dim - dimensão
Saída: $\mathbf{c}$
$tidx =$ 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 da funções atômicas disponibilizadas pela linguagem CUDA. O somatório é feito dois a dois até que a dimensão seja reduzida a 1. O Algoritmo 6 mostra o kernel de um passo desta redução e o Algoritmo 7 mostra como a CPU o chama sucessivamente.

Algoritmo 6 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 7 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 8.

Algoritmo 8 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, ela é esvaziada com o fim do kernel. Ou seja, no inicio 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 (Fig. 2(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 lento em precisão dupla, enquanto que, o GP100 é apenas duas [7]. 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 Aplicações e Resultados

Duas aplicações 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 de gradientes conjugados 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.

Na CPU, as resoluções 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 um rearranjo do primeiro, copia algumas variáveis da memória global para a compartilhada e as utiliza a partir da segunda. Os dois últimos algoritmos tem como base o Algoritmo 8. 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.

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 2 possui os dados utilizados em cada aplicação e a Tabela 3 possui um resumo de quais algoritmos foram utilizados em cada aplicação.

Tabela 2. Dados utilizados em cada aplicação

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

Tabela 3. Algoritmos utilizados em cada aplicação

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

4.1 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 2 e na Fig. 3. Pelo fato de termos o estado plano de tensões, a matriz de elasticidade utilizada é descrita na Eq. (7). O domínio foi discretizado em oito malhas diferentes, os detalhes de seis destas podem ser vistos na Fig. 4.

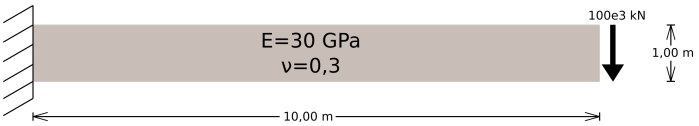
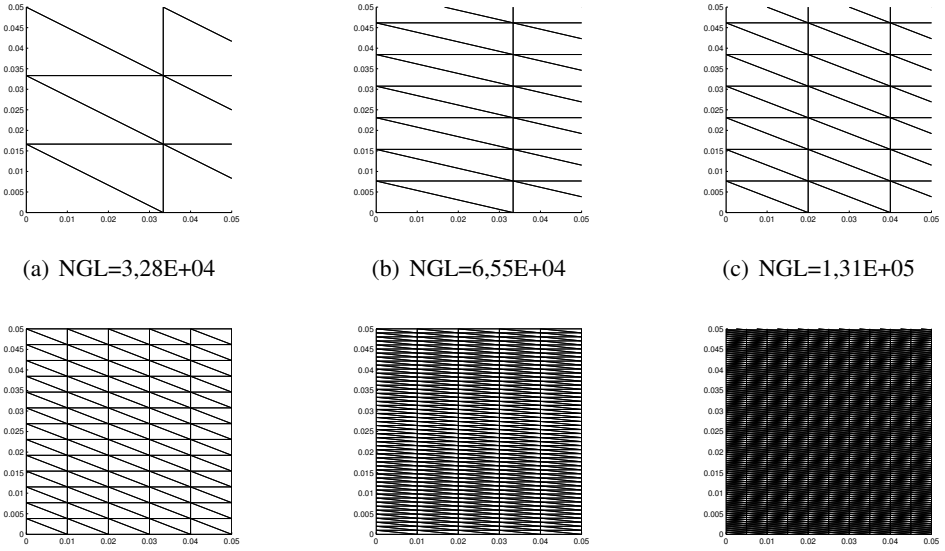


Figura 3. Detalhamento do problema da viga



(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

Figura 4. Detalhes de seis das oito malhas utilizadas

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 que podem ser vistos na Tabela 4. Na tolerância do método foi considerado um erro de 10^{-5}.

Tabela 4. Tempo de construção da matrize 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

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 5 os tempos de resolução e na Fig. 5 os speedups 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 5. 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

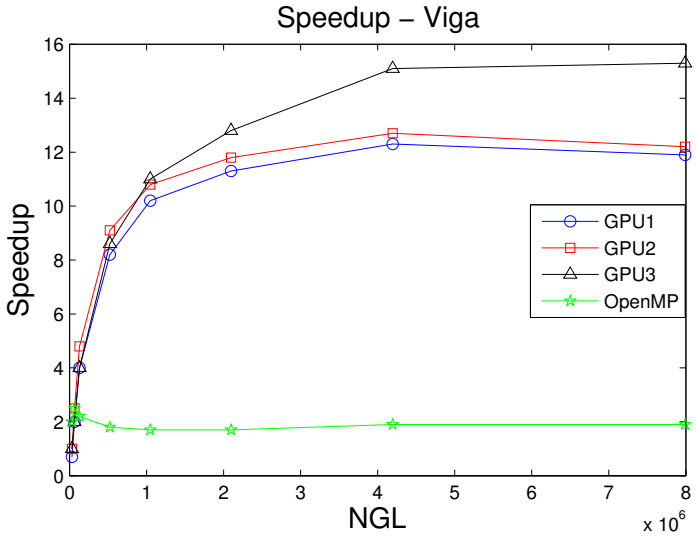


Figura 5. Speedups alcançados na resolução do sistema.

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 algoritmos 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.

4.2 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 2 e na Fig. 6. A matriz de elasticidade utilizada é descrita pela Eq. (8). As forças de corpo não foram consideradas. O domínio foi discretizado em seis malhas

diferentes, detalhes destas podem ser vistos na Fig. 7.

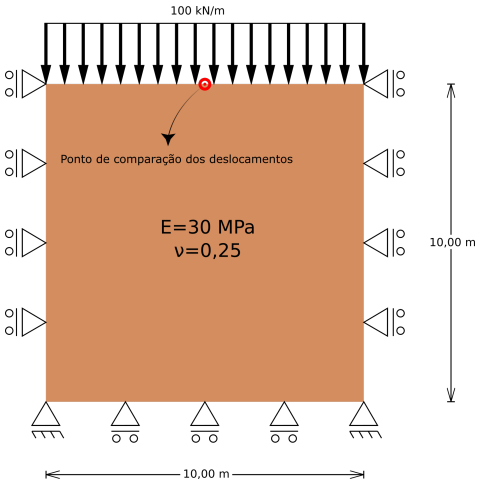
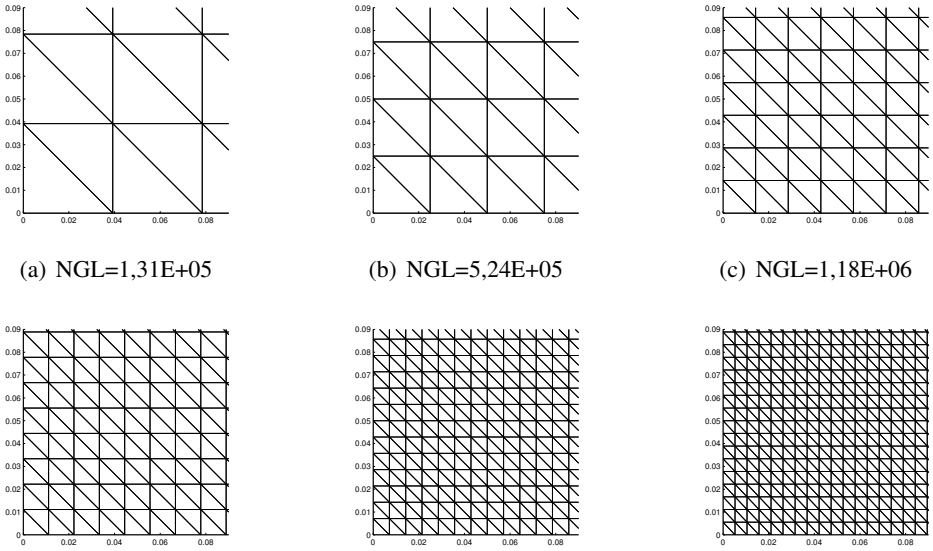


Figura 6. Detalhamento do problema do bloco de solo



(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 (f) NGL=7,79E+06

Figura 7. Detalhes das malhas utilizadas

Assim como a anterior, esta aplicação foi simulada inteiramente na CPU, para a comparação, 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 Fig. 6, 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 6 mostra o tempo gasto nesta montagem em todas as malhas:

Tabela 6. 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
7,79E+06	5	29

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 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 7 podem ser vistos os tempos para todos os algoritmos na linguagem C++ utilizados, na Tabela 8 os tempos no MATLAB com a utilização da função *pcg*, tanto em GPU quanto em CPU, e na Tabela 9 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}.

Tabela 7. 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
7,79E+06	7343	938	127	78	103	61

Tabela 8. 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
7,79E+06	7343	1156	51

Tabela 9. 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
7,79E+06	1,73E-02	1,06E-02	1,40E-02	8,31E-03	6,96E-03

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 Fig. 8, onde os speedups de todos os algoritmos estão relacionados ao tempo do algoritmo na CPU em C++ utilizando o OpenMP:

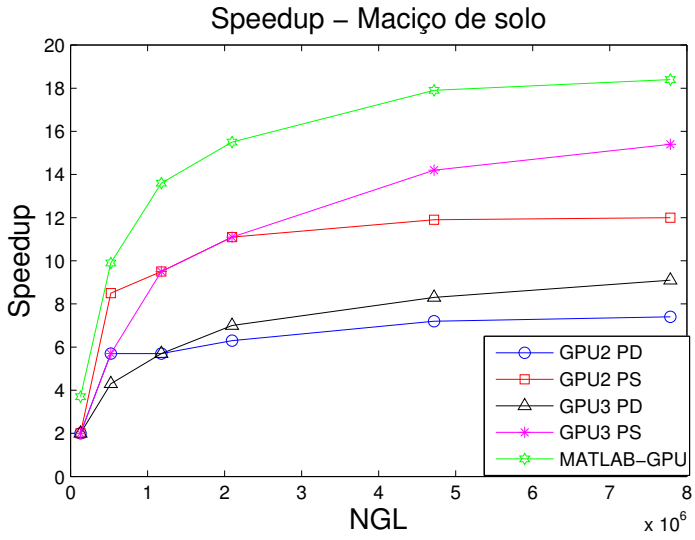


Figura 8. Speedups alcançados na resolução do sistema

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 10 podem ser vistos a quantidade de iterações, o valor dos deslocamentos verticais obtidos - para o ponto indicado na Fig. 6 - com ambas as precisões e seus erros, em relação a

solução obtida pela resolução direta realizada no MATLAB, para cada uma das malhas.

Tabela 10. Erros relativos no deslocamento vertical do ponto indicado na Fig. 6

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
7,79E+06	2,8931241E-02	3,1250003E-02	7,4200273	0,0000094

A depender da malha utilizada, os erros 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 primeiras malhas. Já na última malha utilizada, a única com elementos um pouco distorcidos, o erro foi acima de 7%. Nas aplicações com pontos flutuantes de precisão dupla os erros sempre ficaram abaixo de 1E-4%.

5 Conclusões

Neste trabalho inicial, uma implementação do método dos gradientes conjugados em GPU é proposta na linguagem C++ em conjunto com a linguagem CUDA para a aceleração de simulações mecânicas, com sua eficácia preliminar demonstrada em duas aplicações. Na primeira, 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 demostrado 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.

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, pode ser explorado outras estratégias, como a redução de ordem da matriz na GPU. Em nossos trabalhos futuros, o cálculo das matrizes dos elementos e a montagem da matriz de rigidez global será implementada também em GPU. Também será implementada em GPU, um algoritmo que utiliza ambas as precisões no método dos gradientes conjugados. A implementação de elementos mais potentes e em domínios 3D será realizada para o uso em uma maior gama de aplicações. E os códigos continuarão sendo otimizados e maiores acelerações serão alcançadas.

Agradecimentos

Os autores agradecem à Fundação de Apoio ao Desenvolvimento da Universidade Federal de Pernambuco (FADE-UFPE), a Petrobras e a CMG Reservoir Simulation Foudation pelo apoio financeiro concedido à esta pesquisa. E a NVIDIA, pela doação da GPU TITAN Xp, utilizada nas simulações presentes, através do GPU Grant Program.

Referências

[1] Zienkiewicz, O. C. & Taylor, R. L., 2000. *The Finite Elemente Method*, volume 2. Butterworth-Heinemann, 5 edition.

[2] Bartezzaghi, A., Cremonesi, M., Parolini, N., & Perego, U., 2015. An explicit dynamics gpu structural solver for thin shell finite elementes. *Computers & Structures*, vol. 154, pp. 29–40.

[3] Cheng, J. R. & Gen, M., 2019. Accelerating genetc algorithims with gpu computing: A selective overview. *Computers & Industrial Engineering*, vol. 128, pp. 514–525.

[4] Ma, Y., Li, J., Wu, X., Yan, C., Sun, J., & Vuduc, R., 2019. Optimizing sparse tensor times matrix on gpus. *Journal of Parallel and Distributed Computing*, vol. 129, pp. 99–109.

[5] Kim, Y. & Park, Y., 2019. Cpu-gpu architecture for active noise control. *Applied Acoustics*, vol. 153, pp. 1–13.

[6] Yang, Y.-S., Yang, C.-M., & Hsieh, T.-J., 2014. Gpu parallelization of an object-oriented nonlinear dynamic structural analysis platform. *Simulation Modelling Practice and Theory*, vol. 40, pp. 112–121.

[7] NVIDIA, 2016. Nvidia tesla p100. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

[8] Vilaça, S. F. & Garcia, L. F. T., 1998. *Introdução à Teoria da Elasticidade*. COPPE/UFRJ.

[9] Kwon, Y. W. & Bang, H., 1996. *The Finite Elemente Method using MATLAB*. CRC Press.

[10] MediaWiki, 2019. Eigen. http://eigen.tuxfamily.org/index.php?title=Main_Page. Access date: 28 June 2019.