UNIVERSIDADE FEDERAL DO PAMPA

Marcelo Cogo Miletto

Acelerando uma Aplicação de Simulação Computacional para o Processo de Ablação por Radiofrequência usando GPU

Alegrete
2018

Marcelo Cogo Miletto

Acelerando uma Aplicação de Simulação Computacional para o Processo de Ablação por Radiofrequência usando GPU

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal do Pampa como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Claudio Schepke

Alegrete
2018

Marcelo Cogo Miletto

Acelerando uma Aplicação de Simulação Computacional para o Processo de Ablação por Radiofrequência usando GPU

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal do Pampa como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Trabalho de Conclusão de Curso defendido e aprovado em 03 de dezembro de 2018
Banca examinadora:

Prof. Dr. Claudio Schepke
Orientador
UNIPAMPA

Prof. Dr. Dalvan Jair Griebler
PUCRS

Prof. Dr. Marcelo Resende Thielo
UNIPAMPA

RESUMO

A simulação computacional é uma importante ferramenta para a pesquisa científica atual. A viabilidade do uso da simulação em um dado cenário depende de uma representação computacional eficiente. Desta forma, neste trabalho é estudada a aplicação *Radiofrequency Ablation Finite Element Method* (RAFEM), que foi desenvolvida com o intuito de simular o procedimento clínico de Ablação por Radiofrequência (RFA). Um problema existente na implementação desta aplicação é o elevado tempo de execução envolvido em uma única simulação. Assim, o presente trabalho tem como objetivo alcançar uma redução no tempo total de de espera pela computação dos resultados numéricos da aplicação para que seu uso seja viável em um ambiente clínico. Para tanto, foi necessário analisar e estudar a estrutura do programa a fim de evidenciar as ferramentas e estratégias que irão auxiliar neste processo. Como metodologia, foi adotado um processo de desenvolvimento iterativo voltado à aplicações paralelas, a fim de coordenar esta tarefa e gerar versões paralelas do código a fim de se reduzir o tempo total de execução. Com a utilização deste processo aliado com o uso de bibliotecas de código aceleradas por GPU, foi possível gerar diferentes versões paralelas da aplicação de forma iterativa, adicionando melhorias a cada nova versão. Os resultados obtidos com a última versão gerada mostraram que foi possível reduzir o tempo de computação em até 27 vezes, mantendo os resultados numéricos gerados equivalentes aos gerados pela versão original, tornando assim o uso da aplicação RAFEM mais viável ao reduzir o tempo de espera pelos resultados.

Palavras-chave: Programação Paralela. RAFEM. *Compute Unified Device Architecture*. Método dos Elementos Finitos. Simulação Computacional. GMRES. Biblioteca MAGMA.

ABSTRACT

Computer simulation is a tool of great importance in the scientific research field nowadays. The use of this tool depends on an efficient computational representation to make its use viable. Thus, in this work we study and analyze the RAFEM application which was developed aiming to simulate the RFA clinical procedure. An existing problem in this program is the high execution time involving a single simulation. Thus, the present work aims to achieve a reduction in the total waiting time for computing the numerical results of the application so that its use is feasible in a clinical environment. As a methodology, it was adopted an iterative development process aimed at parallel applications, in order to coordinate this task and generate parallel versions of the code for reducing the total execution time. Using this process allied to the use of GPU-accelerated code libraries, it was possible to generate different parallel versions of the application in an iterative way, adding improvements to each new version. The results obtained with the last version generated showed that it was possible to reduce the computation time by up to 27 times, maintaining the numerical results generated equivalent to those produced by the original version, thus making the use of the RAFEM application more feasible by reducing waiting time for the results.

Key-words: Parallel Programming. RAFEM. Compute Unified Device Architecture. Finite Element Method. Computer Simulation. GMRES.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

RFA Ablação por Radiofrequência

API *Application Programming Interface*

APOD *Access, Parallelize, Optimize, Deploy*

COO *Coordinate*

CPU *Central Processing Unit*

CSR *Compressed Sparse Row*

CUDA *Compute Unified Device Architecture*

DRAM *Dynamic Random Access Memory*

MEF Método dos Elementos Finitos

GMRES *Método do Resíduo Mínimo Generalizado*

GMRES(m) *Método do Resíduo Mínimo Generalizado Reiniciado*

GPU *Graphics Processing Unit*

MPI *Message Passing Interface*

MUMPS *MUltifrontal Massively Parallel Solver*

NNZ *Non Zeros*

OpenCL *Open Computing Language*

OpenMP *Open Multiprocessing*

RAFEM *Radiofrequency Ablation Finite Element Method*

SM *Streamming Multiprocessor*

SP *Streamming Processor*

ULA Unidade Lógica Aritmética

SUMÁRIO

1 INTRODUÇÃO

A simulação computacional é uma ferramenta que permite representar um sistema do mundo real através do uso de modelos matemáticos. Tais modelos conseguem representar computacionalmente uma forma simplificada da realidade, o que permite estudar um determinado sistema para compreender seu comportamento através da realização de experimentos em uma simulação. Esta técnica se faz presente em praticamente todas as áreas como nas engenharias, na meteorologia e na área da saúde, sendo assim, uma ferramenta de grande potencial de aplicação.

Uma vantagem da simulação computacional é o custo reduzido para realizar um experimento em alguns casos, pois esta dispensa o uso de materiais, estruturas ou ambientes específicos, que muitas vezes são difíceis de se encontrar ou custosos para se adquirir. Embora o custo de uma simulação computacional quando comparado com um experimento real possa ser menor, ainda existem outros custos a serem considerados, uma vez que envolvem a complexidade de se implementar uma solução deste tipo e o tempo de execução decorrido para uma simulação em um ambiente computacional.

Neste contexto, o presente trabalho apresenta o programa para simulação computacional denominado RAFEM, desenvolvido no trabalho de Jiang et al. (2010). RAFEM é uma aplicação feita para simular o procedimento de RFA que é um procedimento médico minimamente invasivo utilizado para o tratamento do câncer hepático. Este procedimento envolve a utilização de um eletrodo especial conectado a um gerador de corrente alternada, onde a energia proveniente do gerador é conduzida até o centro do tumor no fígado pelo eletrodo, destruindo assim as células cancerígenas através do calor gerado no local devido ao efeito Joule. O propósito desta aplicação segundo o autor é proporcionar uma ferramenta especializada para uso clínico, que, por meio da simulação computacional, forneça uma maior compreensão e interpretação do processo de RFA, permitindo identificar qual o melhor ajuste de parâmetros envolvidos para cada caso de tratamento de um paciente específico.

1.1 Problema

A modelagem computacional do problema de distribuição de calor no fígado durante o procedimento de RFA é feita utilizando o Método dos Elementos Finitos (MEF). Este método divide o problema em elementos geométricos bem definidos que possuem relações matemáticas com os elementos vizinhos, formando uma malha. A utilização de MEF recai sobre a solução de um grande sistema de equações lineares, onde a solução deste sistema representa, no caso do RAFEM, a temperatura e a tensão em cada ponto da malha em um determinado instante de tempo de simulação.

Obter esta solução envolve um grande custo computacional. Segundo Camarda e Stadtherr (1998), a obtenção da solução de grandes sistemas de equações é uma tarefa que envolve muitos cálculos computacionais, levando mais de 80% do tempo total em algumas

simulações. Atualmente, um dos problemas existentes no RAFEM se encontra em obter esta solução, o que é feito usando o Método Frontal (IRONS, 1970), o qual não é o mais eficiente em termos de desempenho.

Alguns dados de tempo de execução do programa já foram coletados em análises de trabalhos prévios e também neste trabalho. No trabalho de Possebon (2016) um caso de teste chegou a atingir 28 horas de execução do software RAFEM para a simulação de 4 minutos e 10 segundos de um procedimento de RFA. Neste trabalho, o programa foi testado com duas malhas de tamanhos diferentes, uma maior e outra menor, para as quais o tempo de execução do programa foi de 20 horas e 27 minutos e 1 hora e 23 minutos respectivamente.

Em seu trabalho, Jiang et al. (2010) afirma que é possível reduzir o tempo para uma simulação usando um método de solução mais eficiente que o tradicional Método Frontal. Possebon (2016) também destaca como um trabalho futuro a melhoria do método de cálculo para diminuir o tempo de simulação. Alguns esforços para reduzir o tempo total de simulação do RAFEM foram feitos, como no trabalho de Kapelinski (2016), onde foi focada a paralelização da etapa de montagem do sistema de equações utilizando a *Application Programming Interface* (API) OpenMP, porém mantendo o mesmo método. A redução de tempo obtida foi de aproximadamente 4 vezes menos tempo para os casos testados. Mesmo com os bons resultados obtidos no trabalho citado anteriormente, ainda existem algumas estratégias para se obter melhor desempenho que podem ser exploradas. A utilização de outro método para a solução do sistema conforme apontado pelos outros autores é uma delas. Isto pode ser feito através do uso de bibliotecas especificamente implementadas para esta função.

Outra estratégia interessante é fazer o uso de GPU para o processamento da simulação que, embora seja possível de se fazer a partir da versão 4.0 do OpenMP lançada em 2013 (OPENMP, 2018), não foi explorada em trabalhos anteriores. Esta alternativa oferece diversas vantagens para a computação de propósito geral, pois as GPUs possuem arquiteturas altamente paralelas com diversas replicações de Unidade Lógica Aritmética (ULA), o que possibilita realizar o processamento de um grande número de operações de forma simultânea, reduzindo o tempo necessário para a execução de um programa.

1.2 Objetivos

Este trabalho tem como objetivo principal obter uma redução do tempo total de execução da aplicação RAFEM, visto que a aplicação apresenta um elevado tempo computacional. Isto já foi evidenciado através de experimentos com a aplicação em trabalhos relacionados. Este objetivo é motivado pelo fato de que se conseguirmos reduzir o tempo total da aplicação poderemos auxiliar no estudo do procedimento, possibilitando realizar mais simulações em um mesmo intervalo de tempo e também simulações utilizando uma precisão maior.

Para alcançar este objetivo serão adotadas estratégias de paralelização da aplicação, para que a mesma possa ser executada em GPU. Neste contexto podemos utilizar interfaces de programação paralela para utilizar o poder computacional de uma GPU para realizar computações de propósito geral, além de explorar bibliotecas aceleradas por GPU que implementam diferentes métodos numéricos para resolver o sistema de equações lineares.

1.3 Organização deste trabalho

O trabalho está organizado da seguinte forma: o Capítulo 2 aborda os fundamentos que baseiam este trabalho. Inicialmente é apresentado o MEF e suas aplicações, é discutida a estrutura de sistemas lineares de equações bem como suas representações e alguns métodos existentes para solucionar estes sistemas. Ainda no Capítulo 2 é apresentada uma visão geral das arquiteturas heterogêneas envolvendo *Central Processing Unit* (CPU)/GPU, e uma introdução sobre a API CUDA.

O Capítulo 3 é dedicado à análise e compreensão da estrutura do programa RAFEM, destacando as etapas do programa onde é concentrada maior parte do tempo de execução e discutindo possíveis estratégias para o aumento de desempenho.

No Capítulo 4 são estudados trabalhos diretamente relacionados com o RAFEM, e outros trabalhos que fazem o uso de GPU em aplicações que utilizam MEF.

No Capítulo 5 é discutida a metodologia, onde são apresentadas as estratégias, ferramentas e o processo de desenvolvimento adotado. O Capítulo 6 relata o desenvolvimento das versões paralelas do RAFEM. Por fim o Capítulo 7 apresenta as considerações finais e direções futuras.

2 FUNDAMENTAÇÃO TEÓRICA

O MEF é uma das técnicas mais difundidas para a solução de problemas práticos como é o caso de aplicações de simulação computacional (RAO, 2017). Sua grande utilização se dá pelo fato de o MEF ser um método bastante genérico o que o torna propício a ser aplicado em diversos campos de pesquisa (SOUZA, 2003). Outro fator que impulsionou a popularização do método foi o crescente desenvolvimento computacional - Hennessy e Patterson (2011) afirmam que a tecnologia computacional teve um incrível progresso nos últimos 65 anos, que proporcionou as ferramentas necessárias para a solução dos sistemas de equações envolvidas nos problemas de elementos finitos de forma eficiente e que são indispensáveis para este tipo de problema (HINTON; OWEN, 1980). Todos estes fatores contribuíram para que este método tenha se estabelecido tão bem atualmente.

Os problemas de elementos finitos geralmente recaem sobre o problema de solucionar um sistema de equações algébricas lineares ou não. Trata-se de um objetivo bastante estudado, para o qual foram desenvolvidas diversas técnicas, onde cada uma destas técnicas possui as suas estratégias e características, das quais algumas são interessantes de serem analisadas para este trabalho, como por exemplo, as etapas e fluxo de execução de um método, que podem ser fatores limitantes para a representação computacional paralela eficiente destes métodos.

O avanço computacional citado anteriormente trouxe modificações nas arquiteturas dos computadores ao longo do tempo, partindo de computadores com uma única CPU, hoje temos as chamadas arquiteturas *multi-core* que são máquinas com diversos núcleos de processamento que possibilitam a execução paralela de instruções. Neste contexto também se encaixam as GPUs, que foram desenvolvidas pensando em eficiência no processamento de objetos gráficos, mas que hoje com as APIs de programação existentes e pelo poder computacional fornecido devido à sua arquitetura altamente paralela, são muito utilizadas na pesquisa científica.

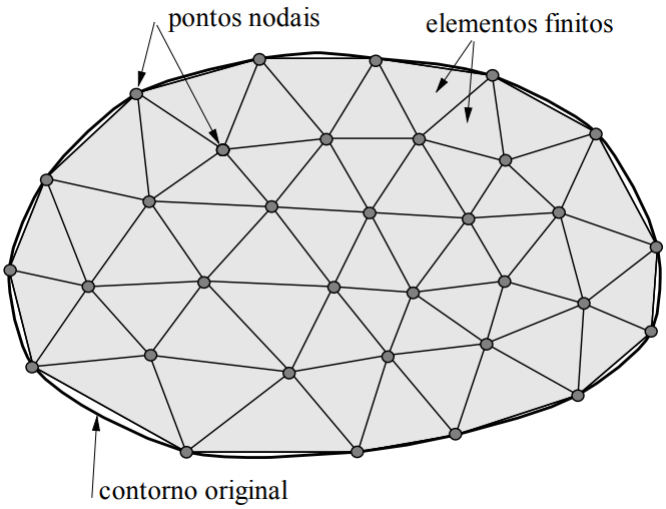
Com base neste contexto introduzido, o presente capítulo traz uma introdução sobre MEF na seção 2.1, a estrutura e representação de sistemas de equações lineares é abordada na seção 2.2 e seção 2.3 bem como os métodos para solucionar estes sistemas na seção 2.4. Uma visão geral sobre as arquiteturas de CPU e GPU é apresentada na seção 2.5, por fim a seção 2.6 é dedicada a uma introdução da interface de programação paralela CUDA.

2.1 Método dos Elementos Finitos

MEF é um método numérico muito utilizado para a modelagem e análise de problemas físicos, tais como aerodinâmica, hidrodinâmica e transferência de calor. Trata-se de um método que produz resultados aproximados para os problemas que modela, e consiste em dividir o sistema que está sendo estudado em elementos geométricos simples e bem definidos, compostos por nós que definem sua forma. A forma de relacionamento entre

um elemento e outro se dá a partir dos nós em comum que conectam os dois elementos, formando assim uma malha que fornece a representação de forma aproximada do domínio do problema. É possível observar os elementos descritos anteriormente na Figura 1, que apresenta uma malha de elementos finitos para um problema em duas dimensões utilizando elementos triangulares.

Figura 1 – Malha de elementos finitos triangulares



Fonte: Souza (2003, p. 1)

A representação deste método é aproximada justamente por envolver um número finito de elementos, Souza (2003) afirma que a precisão do método depende da quantidade de nós e elementos envolvidos, sendo assim quanto menores os elementos maior será o número de nós, e consequentemente maior será a precisão do método se comparada com a solução exata do problema.

Para a análise do problema como um todo, as equações que aproximam a solução dentro de cada elemento que formam a chamada matriz local de rigidez do elemento, são combinadas em uma única matriz para todos os elementos, gerando assim a matriz global de rigidez que é um sistema de equações lineares, onde a solução deste sistema fornecerá a solução de cada nó. Outros fatores também contribuem para a formação da matriz global, que são as condições iniciais e de contorno do domínio que está sendo estudado.

De acordo com Chapra e Canale (2008), a abordagem por elementos finitos segue um determinado padrão de passos:

- **Discretização** - Consiste na divisão do domínio em elementos finitos.
- **Equações dos elementos** - Etapa que compreende o desenvolvimento de equações para aproximar as soluções em cada elemento de forma ótima. As equações são

organizadas em um conjunto de equações algébricas, formando a matriz de rigidez do elemento, também chamada de propriedade do elemento.

- **Montagem** - Depois de modelar o comportamento individual de cada elemento, as matrizes de rigidez de cada um dos elementos são combinadas, definindo o comportamento do sistema inteiro e gerando a matriz de rigidez global. Em muitos casos esta matriz tem como características a esparsidade que significa que grande parte de seus valores são nulos e a concentração destes valores não nulos próximos de sua diagonal principal, o que permite classificar a matriz como uma matriz de banda.

- **Condições de Contorno** - Antes de resolver o sistema, é necessário incorporar as condições de contorno na matriz de rigidez global.

- **Solução** - Fase dedicada à solução do sistema de equações, é a etapa que representa uma grande porcentagem do tempo total de computação. Aqui diversos métodos e estratégias de solução podem ser utilizadas.

- **Pós processamento** - Após o término da computação da solução, os dados produzidos podem ser manipulados para serem exibidos graficamente.

2.2 Sistemas de Equações Lineares

De acordo com Leon (2000), os sistemas de equações lineares podem ser encontrados em aplicações científicas e industriais das mais diversas áreas, fazendo parte de cerca de 75% de todos os problemas matemáticos encontrados nesse contexto. Daí a importância de se buscar formas eficientes de solucionar este tipo de problema. Para alcançar este objetivo, é importante introduzir os elementos que formam um sistema de equações lineares, como eles são representados em conjunto e quais propriedades e operações permitem obter a solução de um sistema de equações.

Uma equação linear é formada por um conjunto de termos nos quais o maior expoente é um, sendo representada de acordo com a Equação 2.1.

$$a_1x_1 + a_2x_2 + ... + a_nx_n = b_1, \tag{2.1}$$

onde $x_1, x_2, ..., x_3$ são as incógnitas, $a_1, a_2, ..., a_n$ são os coeficientes das incógnitas e b_1 é o termo independente. Uma solução de uma equação linear é dada pelo conjunto de valores de incógnitas que satisfaçam a relação de igualdade com o termo independente.

Equações lineares podem ser agrupadas em um sistema de equações lineares, neste caso, a solução é dada pelo conjunto de valores que satisfaçam simultaneamente o sistema

de equações. A representação de um sistema de equações contendo m equações e n incógnitas é dada pela Equação 2.2.

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m \end{cases} \tag{2.2}$$

Durante o processo de solução de um determinado sistema de equações lineares pode-se chegar nas seguintes conclusões: (1) o sistema não tem solução, (2) o sistema tem uma única solução ou (3) o sistema tem mais do que uma solução. Com isto, os sistemas de equações lineares são classificados da seguinte forma de acordo com o seu conjunto de solução:

1. Sistemas Impossíveis, conjunto solução é vazio;
2. Sistemas Possíveis e Determinados, possui uma única solução;
3. Sistemas Possíveis e Indeterminados, possui infinitas soluções.

Sistemas de equações lineares também podem ser representados de forma matricial seguindo a Equação 2.3, onde A é uma matriz de dimensão $m \times n$ chamada matriz dos coeficientes, x é um vetor coluna de dimensão $n \times 1$ contendo as incógnitas e b é um vetor coluna com de dimensão $m \times 1$ contendo os termos independentes, a representação matricial é apresentada na Equação 2.4.

$$Ax = b \tag{2.3}$$

$$\begin{bmatrix} a_{11} & a_{12} & ... & a_{1n} \\ a_{21} & a_{22} & ... & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & ... & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} \tag{2.4}$$

A matriz também podem ser escritas na sua forma aumentada, onde os termos independentes são agrupados logo após a sua última coluna, conforme a Equação 2.5. Neste caso as incógnitas estão implícitas em cada coluna da matriz junto com os coeficientes. Esta representação matricial ajuda tanto na compreensão do processo de solução quanto na representação computacional do mesmo.

$$\left[\begin{array}{cccc|c} a_{11} & a_{12} & \dots & a_{1n} & b_1 \\ a_{21} & a_{22} & \dots & a_{2n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} & b_m \end{array}\right] \tag{2.5}$$

Antes de obter a solução de um sistema de equações lineares é importante definir a noção de equivalência entre sistemas de equações lineares. Dois sistemas são ditos equivalentes se os seus conjuntos solução são iguais (LEON, 2000), isso significa que as suas equações podem ser transformadas de forma a simplificar uma equação ao ponto de se obter uma solução óbvia que é equivalente a solução da equação original do sistema.

Para isto, são definidas três regras para a manipulação de sistemas de equações lineares mantendo a propriedade de equivalência:

1. Trocar ordem das equações;
2. Multiplicar os coeficientes e o termo independente da equação por uma constante não nula;
3. Adicionar a uma equação outra equação multiplicada por uma constante.

Sendo assim, podemos usar as três operações definidas acima para resolver o seguinte sistema de equações lineares para chegar em um sistema equivalente que seja fácil de resolver, como é o caso dos sistemas triangulares superiores (Equação 2.6) e inferiores (Equação 2.7) representados abaixo.

$$\left[\begin{array}{ccc|c} 4 & 2 & 2 & 8 \\ 0 & 5 & 1 & 6 \\ 0 & 0 & 2 & 2 \end{array}\right] \tag{2.6}$$

$$\left[\begin{array}{ccc|c} 4 & 0 & 0 & 4 \\ 2 & 5 & 0 & 7 \\ 2 & 1 & 2 & 5 \end{array}\right] \tag{2.7}$$

Podemos perceber que os sistemas de equações lineares que se encontram nesta forma são fáceis de se resolver, basta olhar para a equação que possui apenas um coeficiente diferente de zero para descobrir o valor da incógnita que ficou isolada. Com o valor de uma incógnita obtido, é possível aplicá-lo nas demais equações para resolvê-las, e assim sucessivamente. Este processo é chamado de substituição regressiva ou substituição para frente nos sistemas triangulares superiores e inferiores, respectivamente.

2.3 Representação de Matrizes Esparsas

Muitos problemas de simulação computacional como o tratado pelo RAFEM envolvem a solução de um sistema de equações que tem como uma de suas características ser bastante esparso, ou seja, a maioria dos coeficientes são nulos. Outra característica destes sistemas é que eles tem um tamanho considerável, o que leva a sua representação computacional a consumir uma boa quantidade de memória para representá-lo. Muitas vezes isto acaba se tornando um problema, visto que temos uma quantidade finita de memória para se utilizar em um problema computacional. Para amortizar este tipo de problema, podemos usar formas alternativas de se representar uma matriz esparsa, voltadas para um menor consumo de memória.

Este é o caso dos formatos de representação de matrizes esparsas *Coordinate* (COO) e *Compressed Sparse Row* (CSR), que são muito utilizados em bibliotecas que trabalham com sistemas de equações esparsos. Estes formatos são especificamente úteis em métodos iterativos para a solução de sistemas de equações, pois estes métodos trabalham apenas com os coeficientes não nulos do sistema, não sendo necessário o armazenamento dos coeficientes iguais a zero. Dois formatos muito utilizados para este tipo de representação são o COO e o CSR.

O formato COO representa uma matriz esparsa utilizando três vetores: um para armazenar os índices das linhas, um para armazenar os índices das colunas e outro para armazenar os valores da matriz. Com isto o método de armazenamento COO consegue representar uma matriz de tamanho $N \times N$ usando três vetores de tamanho igual a quantidade de coeficientes não nulos contidos nela. Para demonstrar o uso deste formato, vamos considerar a matriz apresentada pela Equação 2.8

$$A = \begin{bmatrix} 4 & 0 & 0 & 0 & 0 \\ 0 & 5 & 1 & 0 & 0 \\ 0 & 1 & 6 & 0 & 0 \\ 0 & 0 & 0 & 7 & 0 \\ 0 & 0 & 0 & 0 & 8 \end{bmatrix} \tag{2.8}$$

Para representá-la em formato COO temos de guardar a informação da linha e coluna que os valores se encontram. Por exemplo, o valor 5 se encontra na linha 1 e na coluna 1, assim basta verificar todos os coeficientes e adicionar suas informações nos vetores que representam o sistema. A representação desta matriz em formato COO é descrita pela Equação 2.9. O acesso aos elementos não nulos da matriz pode ser feito percorrendo estes três vetores. Podemos perceber que mesmo para um caso pequeno, este formato já reduz a quantidade de memória necessária para representar a matriz.

$$\begin{array}{rccccccc} Linhas = & \boxed{1} & \boxed{2} & \boxed{2} & \boxed{3} & \boxed{3} & \boxed{4} & \boxed{5} \\ Colunas = & \boxed{1} & \boxed{2} & \boxed{3} & \boxed{2} & \boxed{3} & \boxed{4} & \boxed{5} \\ Valores = & \boxed{4} & \boxed{5} & \boxed{1} & \boxed{1} & \boxed{6} & \boxed{7} & \boxed{8} \end{array} \tag{2.9}$$

O formato CSR também faz o uso destes três vetores para descrever a matriz, e pode ser organizado por linhas, assim o seu vetor correspondente as linhas da matriz terá o tamanho de $N + 1$ para uma matriz com N linhas. A diferença deste formato para o COO é que o CSR não repete as linhas, ele armazena a quantidade de elementos não nulos contidos em cada linha somados com os elementos da linha anterior, sendo que a primeira posição deste vetor possui o número 0 e a segunda posição é que armazena a quantidade de elementos da primeira linha. Para saber quantos elementos a linha i possui basta subtrair o valor da linha anterior. Olhando para a Equação 2.10 que representa a matriz

descrita anteriormente no formato CSR, podemos calcular quantos valores a primeira linha possui usando o vetor das linha para realizar uma subtração da primeira posição na segunda $(1-0)$, para a segunda linha faz-se a terceira posição menos a segunda $(3-1)$, e assim sucessivamente. Com isto, pode-se diferenciar a qual linha uma coluna e um valor pertencem.

$$\begin{array}{rccccccc} Linhas = & \boxed{0} & \boxed{1} & \boxed{3} & \boxed{5} & \boxed{6} & \boxed{7} & \\ Colunas = & \boxed{1} & \boxed{2} & \boxed{3} & \boxed{2} & \boxed{3} & \boxed{4} & \boxed{5} \\ Valores = & \boxed{4} & \boxed{5} & \boxed{1} & \boxed{1} & \boxed{6} & \boxed{7} & \boxed{8} \end{array} \tag{2.10}$$

2.4 Métodos para Solução de Sistemas de Equações Lineares

A tarefa de se obter a solução de sistemas de equações lineares provenientes dos mais diversos domínios de problemas reais, depende da eficiência e robustez de métodos numéricos para tal finalidade. Estes sistemas de equações podem ter diversas características que influenciam na eficiência dos métodos numéricos empregados para a solução, como a complexidade e o tamanho do conjunto de equações que formam o sistema. Segundo Tu, Yeoh e Liu (2018) existem duas famílias de métodos numéricos para a solução destes conjuntos de equações: os métodos diretos e os métodos iterativos.

Os **métodos diretos** são aqueles que chegam em uma solução exata, exceto nos casos em que ocorrem problemas de arredondamento. Estes métodos são baseados nas três regras de equivalência definidas na seção anterior e buscam, através de uma sequência finita da aplicação destas regras, chegar em uma representação do sistema que seja fácil de resolver, como por exemplo os sistemas de equações lineares na sua forma triangular superior ou inferior. Embora a maioria dos métodos mais familiares para solucionar sistemas de equações tenham como base as mesmas operações, a estratégia utilizada por um determinado método implica na quantidade de operações necessárias para se resolver um sistema de equações lineares e determina o fluxo de execução do método, essas características podem determinar a eficiência de um método quando representado computacionalmente.

Um método direto como a Eliminação de Gauss pode ser utilizado para resolver qualquer sistema de equações (TU; YEOH; LIU, 2018). Entretanto, este método possui um alto custo computacional devido ao número de operações que realiza. Para problemas de simulação computacional utilizando o MEF o sistema de equações resultante geralmente é grande e esparso, tornando o custo do método bastante elevado e seu uso impraticável nestas condições (JAMAL, 2017). Neste caso, algum método direto voltado para sistemas esparsos seria mais eficiente, como por exemplo o método Frontal, que não trabalha com o sistema esparso inteiro, mas com subsistemas menores e mais densos, evitando assim um grande número de operações envolvendo elementos nulos do sistema.

Desta forma, como uma alternativa para problemas maiores temos os **métodos iterativos**, estes consistem em regras para a construção de soluções aproximadas a partir de uma aproximação inicial, que tendem a convergir para a solução definitiva do problema ao longo de sucessivas aproximações, levando em consideração uma certa taxa de tolerância como critério de parada bem como um número máximo de iterações. Se o número de iterações para atingir certo nível de convergência for pequeno, um método iterativo pode ser menos custoso que um método direto, outro fator que faz com que o custo de usar este tipo de método seja menor é o de trabalhar apenas com os elementos não nulos de um sistema. O método iterativo estudado neste trabalho é o *Método do Resíduo Mínimo Generalizado* (GMRES) e é apresentado na subseção 2.4.3.

2.4.1 Método de Eliminação de Gauss com Substituição Regressiva

Este método divide o processo de solução de um sistema linear de equações em duas etapas. A primeira é chamada de eliminação regressiva e trata de uma série de eliminações de termos das equações onde os termos ditos eliminados são transformados em zero. O objetivo da primeira etapa é transformar o sistema para a forma triangular inferior, e faz isto utilizando os elementos da diagonal principal da matriz para eliminar todos os termos abaixo dele. A segunda etapa se inicia quando o sistema já se encontra na forma triangular inferior e consiste em resolver a equação com uma única variável para obter o seu resultado e substituir regressivamente nas demais equações.

Para exemplificar o processo de solução usado pelo método de eliminação de Gauss, será utilizado o sistema de equações lineares definido na Equação 2.11, contendo as equações E_1, E_2 e E_3.

$$\begin{array}{c} E_1 \\ E_2 \\ E_3 \end{array} \left[\begin{array}{ccc|c} 2 & 4 & 3 & 14 \\ 2 & 2 & 6 & 16 \\ 3 & 4 & 3 & 20 \end{array}\right] \tag{2.11}$$

A etapa de eliminação se inicia selecionando o elemento da primeira coluna da diagonal principal para eliminar os termos de mesma coluna das linhas abaixo, este elemento é chamado de pivô. Neste caso o primeiro pivô selecionado é o primeiro elemento de E_1, com o valor 2. Para realizar a eliminação dos termos das equações devemos adicionar E_1 multiplicada por uma constante de forma que o resultado dessa adição elimine os termos abaixo do pivô.

Para encontrar o valor das constantes C_1 e C_2 que fazem com que os termos a_{21} e a_{31} sejam eliminados, basta calcular:

$$a_{21} + C_1 a_{11} = 0 \rightarrow C_1 = -\frac{a_{21}}{a_{11}}$$

$$a_{31} + C_2 a_{11} = 0 \rightarrow C_2 = -\frac{a_{31}}{a_{11}}$$

onde para este passo da eliminação temos $C_1 = -1$ e $C_2 = \frac{-3}{2}$.

$$\begin{matrix} E_1 \\ E_2 \\ E_3 \end{matrix}\left[\begin{array}{ccc|c} \boxed{2} & 4 & 3 & 17 \\ 2 & 2 & 6 & 16 \\ 3 & 5 & 7 & 26 \end{array}\right] \xrightarrow{E_2+C_1E_1} \left[\begin{array}{ccc|c} \boxed{2} & 4 & 3 & 17 \\ 0 & -2 & 3 & -1 \\ 3 & 5 & 7 & 26 \end{array}\right] \xrightarrow{E_3+C_2E_1} \left[\begin{array}{ccc|c} \boxed{2} & 4 & 3 & 17 \\ 0 & -2 & 3 & -1 \\ 0 & -1 & \frac{5}{2} & \frac{1}{2} \end{array}\right]$$

Com este primeiro passo da eliminação foi possível observar que todos os termos abaixo do elemento pivô foram eliminados e as equações E_2 e E_3 foram transformadas em equações com valores diferentes porém equivalentes. Para o próximo passo o segundo elemento da equação E_2 é selecionado como pivô e o processo de eliminação é repetido, encontrando novos valores de constantes para eliminar os termos abaixo do pivô. Neste caso só é necessário calcular o valor de uma constante, pois só existe uma equação abaixo do elemento pivô. Portanto calculamos C_3 sendo:

$$a_{32} + C_3 a_{22} = 0 \rightarrow C_3 = -\frac{a_{32}}{a_{22}},$$

o que resulta em $C_3 = \frac{-1}{2}$.

Com a adição de C_3E_2 em E_3 o que deixa o sistema na forma triangular superior, damos fim à etapa de eliminação progressiva.

$$\begin{matrix} E_1 \\ E_2 \\ E_3 \end{matrix}\left[\begin{array}{ccc|c} 2 & 4 & 3 & 17 \\ 0 & \boxed{-2} & 3 & -1 \\ 0 & -1 & \frac{5}{2} & \frac{1}{2} \end{array}\right] \xrightarrow{E_3+C_3E_2} \left[\begin{array}{ccc|c} 2 & 4 & 3 & 17 \\ 0 & -2 & 3 & -1 \\ 0 & 0 & 1 & 1 \end{array}\right]$$

Depois que a Equação 2.11 se encontra no formato triangular inicia-se etapa de de substituição regressiva. Nesta etapa o primeiro passo é encontrar a solução da equação com uma única variável, para encontrar o valor de uma incógnita e aplicar este valor nas demais equações.

Olhando para a equação E_3 é possível chegar ao resultado $x_3 = 1$, o passo seguinte é substituir o valor de x_3 na equação E_2, o que resulta em $x_2 = 2$. Por fim usa-se os valores de x_2 e x_3 obtidos para resolver a equação E_1, resultando em $x_1 = 3$ e resolvendo o sistema. Este processo pode ser observado nas equações abaixo.

$$\begin{array}{lll} x_3 = \dfrac{a_3}{a_{33}} & x_3 = \dfrac{1}{1} & x_3 = 1 \\ x_2 = \dfrac{b_2 - a_{23}x_3}{a_{22}} & x_2 = \dfrac{-1 - 3 \times 1}{-2} & x_2 = 2 \\ x_1 = \dfrac{b_1 - a_{12}x_2 - a_{13}x_3}{a_{11}} & x_1 = \dfrac{17 - 4 \times 2 - 3 \times 1}{2} & x_1 = 3 \end{array}$$

Para sistemas de equações relativamente pequenos este método pode ser utilizado, mas conforme cresce o tamanho do problema, como vimos anteriormente, o seu uso se torna impraticável. Isto porque a complexidade do algoritmo para a eliminação de Gauss é $O(n^3)$.

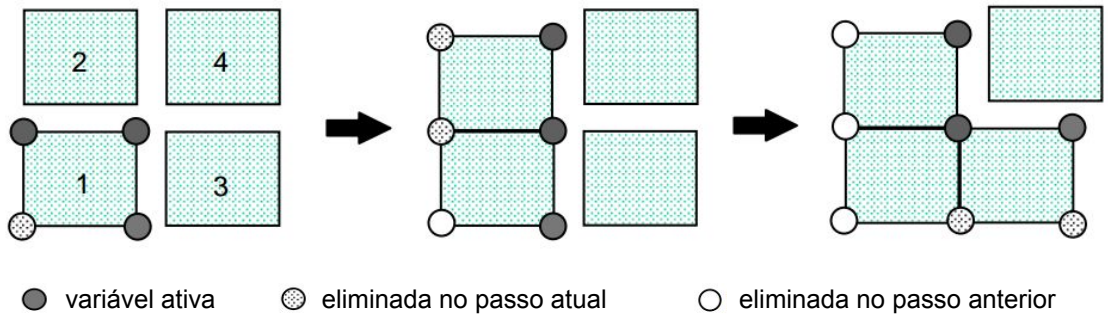
2.4.2 Método Frontal

O método de solução Frontal é uma variação do método de eliminação de Gauss (SCOTT, 2006), foi desenvolvido por Irons (1970). O método foi pensado para ser uma forma eficiente de solução que otimizasse o uso de memória em problemas de MEF, visto que na época em que foi desenvolvido existia ainda uma grande limitação na quantidade de memória disponível nos computadores.

Scott (2003) descreve em seu trabalho que a principal ideia do método Frontal é evitar a montagem completa de todos os elementos na matriz global de rigidez que geralmente é grande e esparsa, ocupando boa quantidade de memória para ser armazenada. Em contrapartida é usada uma matriz menor e mais densa chama matriz Frontal, formada também pela combinação das matrizes elementais. Porém durante o processo deste método é possível realizar a intercalação do processo de montagem e eliminação de variáveis.

O processo de eliminação se dá quando uma variável não está mais envolvida em nenhuma das matrizes elementais a serem montadas, assim linhas da matriz Frontal são escritas em disco e liberadas para a contribuição de outros elementos, desta forma a utilização de memória é reduzida. As linhas depois são recuperadas do disco para finalizar o processo de solução. O processo de eliminação pode ser observado na Figura 2.

Figura 2 – Processo de eliminação do método Frontal



Fonte: Adaptado de Kim et al. (2002)

Entretanto, estas escritas em disco são fatores que degradam o desempenho de uma aplicação pelo fato de o acesso ao disco ser muitas vezes mais lento que o acesso à memoria principal de um computador. Ao longo do tempo foram desenvolvidos variantes do método Frontal que se adequam melhor à realidade dos computadores atuais (SCOTT, 2003; DUFF; REID, 1983; AMESTOY et al., 2001). Nestas variações existe a

possibilidade de escrever ou não em disco, além da exploração do paralelismo nas operações envolvidas durante a execução do método. No entanto, o método Frontal utilizado na aplicação estudada por este trabalho realiza operações em disco e sua execução é totalmente sequencial.

2.4.3 Método do Resíduo Mínimo Generalizado (GMRES)

O GMRES (SAAD; SCHULTZ, 1986) é um método iterativo para obter a solução de sistemas de equações lineares esparsos e não simétricos, isto é, um sistema representado por uma matriz A de tamanho $N \times N$, onde $A_{i_j} \neq A_{j_i}$ para $i \neq j$, que é o caso do sistema gerado no programa RAFEM. Este método é baseado no subespaço Krylov e trata o problema de solucionar $Ax = b$ como um problema de minimização do resíduo inicial $r0$, que é calculado como $r_0 = b - Ax_0$, sendo x_0 uma primeira aproximação da solução. Assim, o resíduo $r0$ representa a diferença entre a solução prevista e a solução real do problema.

Essencialmente o GMRES realiza operações de multiplicação entre matriz e vetor e produto escalar a cada iteração. Desta forma, podemos estimar a sua complexidade como $O(IN^2)$ onde I é o número de iterações realizadas pelo método. Este número de iterações dependerá do valor de tolerância definido para o método e das propriedades de convergência do sistema de equações que está sendo processado. Uma estratégia utilizada nos métodos iterativos para reduzir o número de iterações necessárias é o uso de pré-condicionadores que operam sobre o sistema de equações para melhorar a sua taxa de convergência. Para uma matriz de tamanho N, no pior dos casos o método realiza N iterações para atingir a solução desejada, porém isto dificilmente acontece. No geral, espera-se que o método GMRES chegue numa solução aproximada dentro da tolerância especificada em uma fração relativamente pequena de N.

Uma vantagem deste método é que as operações que ele envolve são operações de álgebra linear bastante estudadas e que são possíveis de se paralelizar, inclusive existindo diversas bibliotecas que implementam estas operações de forma paralela e otimizada para GPUs, como as bibliotecas CUBLAS e MAGMA BLAS. Assim o custo de N^2 operações por iteração pode ser dividido entre as unidades de processamento de uma arquitetura de forma paralela. Com isto, o uso deste método pode se adequar no funcionamento do RAFEM, contribuindo para a melhoria de seu desempenho.

No entanto, o presente método possui um inconveniente descrito por Saad e Schultz (1986) a respeito de um conjunto de vetores de tamanho k que o método utiliza, onde k aumenta com o número de iterações, aumentando assim o custo de armazenamento destes vetores em um fator k e o número de operações de multiplicação necessárias em $\frac{1}{2}k^2N$, tornando este algoritmo impraticável devido ao custo de memória e de operações conforme k aumenta (SAAD, 2003).

Assim foram propostas variações deste método, como o *Método do Resíduo Mínimo*

Generalizado Reiniciado (GMRES(m)) que utiliza um parâmetro extra m indicando um valor de reinicialização para o algoritmo a cada m iterações, mantendo a solução obtida até então para ser o vetor x_0 de uma próxima rodada de operações até m. Com isto, o valor máximo de k é limitado por m. No entanto, este valor deve ser escolhido com cautela, pois caso seja escolhido um valor muito alto, o método GMRES(m) pode se comportar como o GMRES padrão, já se escolhermos um valor muito baixo, o método pode demorar muito para convergir ou até não convergir em alguns casos caracterizando o fenômeno de estagnação. Mas, surpreendentemente, com uma configuração de m menor pode convergir em menos iterações para outros casos (EMBREE, 2003). Diante deste problema da escolha do melhor valor de m, ainda existem outras variações do GMRES(m) que escolhem m dinamicamente ao longo da execução de uma aplicação através do uso de heurísticas e cálculos sobre as soluções obtidas (JOUBERT, 1994; BAKER; JESSUP; KOLEV, 2009).

2.5 Arquiteturas Heterogêneas *CPU/GPU*

Com o crescente desenvolvimento computacional previsto pela Lei de Moore (MOORE, 1965), foi possível executar *software* cada vez mais rápidos, tendo melhores funcionalidades e se tornando mais úteis para os usuários. Estes que cada vez mais requeriam desta tecnologia, gerando uma fase positiva para a indústria de computadores, que se responsabilizava por manter o contínuo desenvolvimento, praticamente dobrando o número de transistores por processador a aproximadamente cada dois anos ao mesmo tempo em que mantinha os custos.

O fato é que essa tendência veio diminuindo a partir do ano de 2003 devido ao consumo elevado de energia e problemas com a dissipação de calor nos processadores (KIRK; WEN-MEI, 2016). Estes problemas se tornaram fatores limitantes para o aumento da velocidade de *clock* dos processadores, o que levou os fabricantes de processadores a buscarem por novas estratégias para se obter melhor desempenho nos processadores, agora tendo um maior cuidado com a eficiência energética e a dissipação de calor. A solução para se obter melhor desempenho considerando os obstáculos mencionados foi a de começar a produzir processadores compostos por dois ou mais *cores*, que são processadores mais simples e com menor frequência de *clock*. Isto fez com que surgissem os processadores *multi-core*, que são processadores que contém múltiplas unidades de processamento, permitindo assim, a execução de instruções de forma paralela, fato de grande importância que foi nomeado por Sutter e Larus (2005) como a revolução da concorrência, para a qual a indústria de *software* deve estar preparada, sendo capaz de produzir aplicações que utilizem os recursos oferecidos pelas arquiteturas paralelas de forma eficiente.

Em contraste com os processadores *multi-core*, existem os chamados processadores *manycore*, também desenvolvidos pensando na execução paralela de instruções. A pesar de que estas duas arquiteturas tenham sido desenvolvidas com o mesmo objetivo, existem diferenças na forma com que são organizadas e as estratégias que utilizam para se obter

poder de computação. Enquanto as arquiteturas *multi-core* foram desenvolvidas para manter a velocidade de programas sequenciais enquanto faziam a transição para múltiplos processadores, os dispositivos de arquitetura *manycore* são voltados para a execução de dezenas, centenas ou até milhares de instruções simultâneas (KIRK; WEN-MEI, 2016).

Na Figura 3 podemos perceber os elementos que representam as diferenças entre as duas arquiteturas, a Figura 3(a) representa uma CPU *multi-core* e a Figura 3(b) uma GPU *manycore*. A arquitetura *multi-core* é otimizada para a performance de código sequencial, contendo uma lógica de controle sofisticada que permite a execução em paralelo de instruções de uma *thread*, somado a existência de uma grande memória *cache* para reduzir a latência de acesso aos dados e instruções necessários por uma aplicação, além de cada ULA também ser otimizada para reduzir a latência. Já as arquiteturas *manycore* possuem um *hardware* especificamente projetado para suportar um grande número de *threads*, com memórias *cache* de pequeno porte visando uma redução ao acesso à memória principal nos casos em que múltiplas *threads* acessam os mesmo dados, maximizando desta forma o espaço dedicado para as ULAs, que também são mais simplificadas.

Figura 3 – Representação das arquiteturas *multi-core* e *manycore*

(a) Arquitetura de uma CPU *multi-core*

(b) Arquitetura de uma GPU *manycore*

Fonte: (NVIDIA, 2018b)

Com isto podemos perceber que cada arquitetura é especializada em um tipo de tarefa diferente, sendo os processadores *multi-core* voltados tanto para a performance paralela quanto sequencial, dando ênfase para o desempenho individual das *threads*, porém limitado a um número menor de *cores* quando comparado com um processador *manycore*.

Estes, já são totalmente focados e otimizados para terem níveis altíssimos de paralelismo tendo seus muitos *cores* mais simples e com menos performance individual. Uma arquitetura heterogênea é aquela que combina as duas arquiteturas, aproveitando o melhor das duas estratégias para alcançar máxima eficiência computacional.

Para alcançar este melhor desempenho oferecido pelas arquiteturas paralelas, é necessário expressar o paralelismo nos programas através da programação paralela. Este paradigma de programação permite ao programador especificar as áreas de código que devem ser executadas concorrentemente entre núcleos de um dispositivo paralelo, garan-

tindo assim uma utilização mais eficiente do poder computacional proporcionado por estes dispositivos. Dada a importância do desenvolvimento de aplicações paralelas, atualmente já existem diversas APIs voltadas para este fim, a maioria delas já oferece suporte para a programação de GPU ou foram projetadas para este fim, como é o caso da API CUDA.

2.6 Compute Unified Device Architecture - CUDA

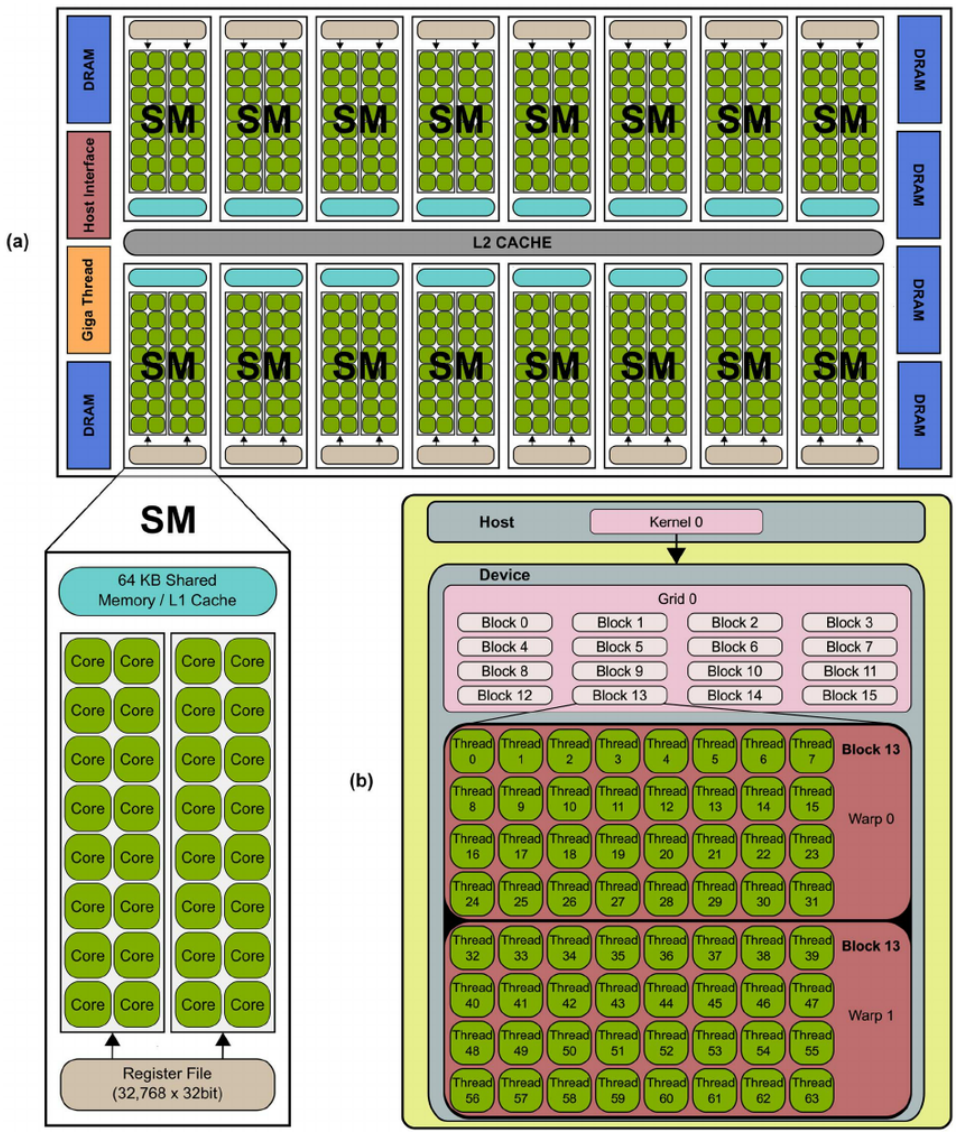
A API CUDA foi lançada pela NVIDIA no ano de 2007, como sendo um modelo de programação para as GPUs NVIDIA, permitindo a solução de vários problemas computacionalmente complexos fazendo o uso destes dispositivos (NVIDIA, 2018b). O modelo de programação CUDA é baseado em três conceitos principais, sendo eles o conceito de hierarquia de grupos de *threads*, memória compartilhada e sincronização por barreira. Estes três conceitos permitem ao programador representar um problema como um subconjunto de problemas de granularidade mais fina para que possa ser executado paralelamente entre os *cores* de uma GPU.

Para entender melhor o uso da API é necessário compreender um pouco a arquitetura das GPUs que suportam a programação em CUDA representada na Figura 4(a). Resumidamente, elas são compostas por um conjunto de *Streamming Multiprocessors* (SMs) que por sua vez são formados por vários *Streamming Processors* (SPs) onde entre eles existe o compartilhamento da lógica de controle e da *cache* de instruções. A GPU também possui uma memória global *Dynamic Random Access Memory* (DRAM) com maior taxa de leitura e gravação de dados do que as memórias convencionais utilizadas em CPU, isto pelo fato de que as GPUs foram desenvolvidas para mover grandes volumes de dados provenientes de operações de processamento gráfico. Cada SM também possui uma memória *cache* L1 compartilhada somente entre os seus SPs e ainda existe uma memória *cache* L2 que é compartilhada entre todos os SMs, tudo isto para reduzir o custo de acesso à memória.

O modelo de programação CUDA foi pensado para funcionar em conjunto com os elementos da arquitetura descrita na Figura 4. Uma função que será executada pela GPU recebe o nome de *kernel*. Na API CUDA deve-se declarar explicitamente quem poderá chamar uma função e onde ela será executada, para isso existem três identificadores de espaço de execução definidos pela API:

- ___host___, define que a função pode ser chamada e executada somente pelo host, onde *host* se refere à CPU e a sua memória;
- ___device___, a função pode ser chamada e executada somente pelo *device*, onde *device* se refere à GPU e a sua memória;
- ___global___, define um *kernel*, que permite que a função seja chamada pelo *host* e executada pelo *device*. Em versões de GPUs mais recentes é possível chamar funções

Figura 4 – Arquitetura e Hierarquia de *threads* CUDA



Fonte: Hernández et al. (2013)

denotadas com este identificador pelo *device*.

Para a execução de um *kernel* CUDA existe uma sintaxe especial, após o nome da função são passados dois parâmetros entre os símbolos de <<< e >>>. Os dois parâmetros juntos, representam a quantidade total de *threads* que vão ser criadas para a execução de um *kernel* CUDA. O primeiro parâmetro indica a quantidade de blocos de *threads* que serão criados e o segundo indica a quantidade de *threads* em cada bloco, este número é limitado a até 1024 nas GPUs atuais. Os blocos de *threads* por sua vez são organizados em uma *grid*, que agrupa todas as *threads* dedicadas à execução de um mesmo código de *kernel*.

Durante a execução de um *kernel* CUDA uma *grid* de blocos de *threads* é criada

na GPU, onde os blocos desta *grid* são distribuídos entre os SMs disponíveis. Cada bloco possui uma área de memória compartilhada entre todas as *threads* que possui e também um mecanismo de sincronização, os blocos podem ser processados em qualquer ordem. Os SMs foram projetados para permitirem a execução de centenas de *threads* em paralelo, agrupando as *threads* de um bloco em grupos de 32 *threads* denominados *warps*, onde todas as *threads* de um *warp* são executadas em paralelo. Um exemplo de organização das *threads* de um *kernel* é apresentado na Figura 4(b).

Um mecanismo de indexação na hierarquia de *threads* é utilizado para definir sobre qual parte dos dados cada *thread* será responsável. Os parâmetros de um *kernel* antes mencionados podem ser definidos usando uma estrutura que representa a quantidade de blocos e *threads* em uma, duas ou três dimensões. Assim, durante a execução de um *kernel*, todas os blocos e *threads* recebem identificadores, isso pode ser usado para acessar facilmente a parte de um vetor, matriz ou volume que se deseja processar. Dentro do código do *kernel* é possível utilizar as variáveis definidas pela API representadas na Tabela 1 para identificar atributos de uma *thread*, bloco ou *grid*.

Tabela 1 – Variáveis da hierarquia de *threads* CUDA

Elemento	**Descrição**	**Variável**
Grid	Identifica quantos blocos compõem uma *grid*	gridDim.x,y,z
Bloco	Identificador de um bloco de uma *grid*	blockIdx.x,y,z
Bloco	Identifica quantas *threads* compões um bloco	blockDim.x,y,z
Thread	Acessa o identificador da *thread*	threadIdx.x,y,z
Warp	Identifica o tamanho de um *warp*	warpSize

Isto tudo permite com que o modelo de programação CUDA seja escalável com o número de SMs de uma GPU, provendo cada vez mais desempenho para as aplicações a cada nova geração de dispositivos que contém mais SMs.

Uma das motivações para a API CUDA ter sido escolhida neste trabalho é pelo fato de que ela pode ser considerada uma API de baixo nível. Isto porque possibilita ao programador configurar aspectos como os acessos a memória global e *cache* e também permite configurar a quantidade de *threads* e a forma como elas serão distribuídas entre os dados a serem processados.

2.7 Balanço do Capítulo

Neste Capítulo foi possível conhecer a estrutura geral de programas de MEF, descrevendo as suas principais etapas e mostrando que as mais custosas são as etapas de montagem e solução do sistema de equações gerado por um programa MEF. Em seguida, são descritas propriedades dos sistemas de equações lineares, definindo formas de representá-los e a noção de equivalência entre sistemas de equações modificados por um conjunto de regras. Regras estas que são aplicadas um certo número de vezes de forma

algorítmica, para transformar um sistema de equações em um outro sistema equivalente onde a obtenção de a sua solução seja mais fácil.

Esta sequência de operações caracterizam os métodos de solução diretos, que como visto podem se tornar muito custosos por realizar operações com elementos nulos quando o sistema de equações é grande e esparso, um exemplo é o método de eliminação de Gauss. Este método é apresentado pois serve como base para o método Frontal, até então utilizado no RAFEM. O método Frontal evita algumas operações com elementos nulos ao rearranjar o problema e trabalhar com submatrizes densas, no entanto, faz o uso de escritas em disco, o que é um fator limitante para o ganho de desempenho e por isso é investigado o uso de outro método numérico.

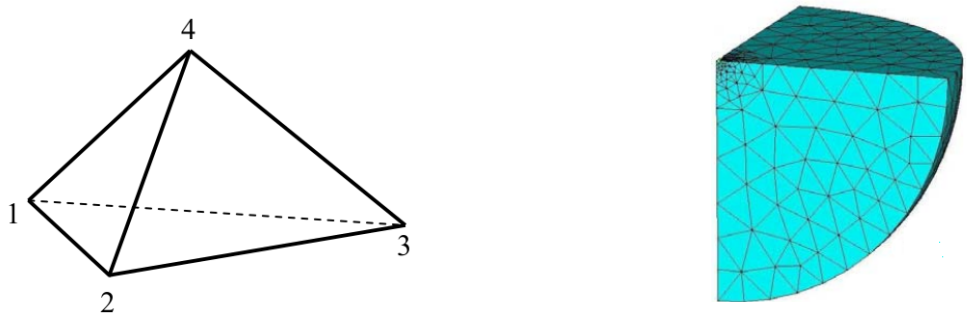
Como alternativa para este problema existem os métodos iterativos, que constroem um conjunto de soluções aproximadas usando apenas os elementos não nulos do sistema, tornando seu uso mais adequado para problemas maiores e esparsos como os que surgem de problemas de MEF. Estes métodos envolvem essencialmente produtos entre matrizes e vetores (JAMAL, 2017), o que é uma operação altamente paralelizável.

As arquiteturas *multi-core* e *manycore* apresentadas na seção 2.5 atualmente oferecem suporte para a execução de aplicações paralelas, possibilitando obter uma melhor performance através do modelo de programação paralela. Assim devemos conhecer interfaces de programação que nos permitem extrair as vantagens oferecidas por este tipo de arquitetura. No caso da programação para GPUs, a API CUDA nos permite utilizar o poder computacional de placas gráficas para a computação de propósito geral e ainda trabalhar com aspectos de mais baixo nível como o controle da memória *cache* de uma GPU.

3 ANÁLISE DO PROGRAMA RAFEM

RAFEM é uma aplicação desenvolvida em linguagem de programação `C++`, com o objetivo de simular o procedimento de RFA, que é uma forma de tratamento minimamente invasiva para o câncer hepático. Isto é feito por meio da utilização de MEF onde o domínio é dividido em elementos tridimensionais de formato tetraedral, como o ilustrado na Figura 5(a), os quais, por sua vez, são conectados aos elementos vizinhos pelos seus nós em comum, formando uma malha de elementos finitos conforme a Figura 5(b).

Figura 5 – Elemento tetraedral e malha de elementos tetraedrais



(a) Elemento tetraedral

(b) Malha de elementos tetraedrais

Fonte: Jiang et al. (2010)

A aplicação calcula a distribuição de temperatura ao longo do tempo e os valores de tensão para cada um dos nós da malha, representando um fígado durante o procedimento de RFA. Neste capítulo será apresentada a estrutura principal do programa, a forma de montagem e solução do sistema de equações lineares e é feita uma análise da execução do programa, identificando as etapas que mais influenciam no tempo de execução total.

3.1 Estrutura do programa

O programa RAFEM é composto por diversos arquivos de código fonte `C++`, porém apenas aqueles arquivos que contém as partes mais importantes para a análise de desempenho do programa serão discutidos neste trabalho. Ao todo são 20 arquivos de código fonte, dos quais foram selecionados apenas dois: `RFA.cpp` e `V_FrontalMethod.cpp`. Estes foram selecionados pelo motivo de estarem presentes no fluxo principal de execução do programa, onde se concentra a maior parte do tempo de execução de acordo com os resultados da análise na seção 3.3.

No arquivo `RFA.cpp` está localizada a função `main` do programa. Boa parte das linhas de código deste arquivo tratam os parâmetros de entrada, leitura de arquivos e inicialização de variáveis. Dentre os parâmetros de entrada necessários podemos destacar o tempo de simulação, a malha de elementos tetraedrais e o número máximo de iterações por passo do programa, que afetam diretamente o tempo de execução da simulação.

Após esta fase de pré-processamento é que se inicia a etapa mais custosa identificada no trabalho de Kapelinski (2016). Trata-se de uma etapa iterativa contendo um laço que realiza um número P de passos onde para cada passo são realizadas I iterações para a convergência de um passo, quando isto ocorre os valores de tensão e temperatura para o instante de tempo atual são armazenados e o instante de tempo é incrementado. Dentro de cada iteração de um passo é chamada a função `VFrontalMethod` contida no arquivo `V_FrontalMethod.cpp`. Nesta função, todos os elementos da malha de elementos finitos são combinados e é resolvido o sistema de equações lineares resultante. Essa estrutura do programa descrita é representada pela Figura 6, onde o retângulo destacado representa a função `VFrontalMethod`, que é a etapa mais custosa do processo de simulação.

Figura 6 – Representação reduzida da estrutura do programa RAFEM

3.2 Montagem e solução do sistema de equações lineares

A função `VFrontalMethod` recebe uma série de parâmetros representando os elementos da malha e outras propriedades, alguns são passados por referência para ter seus

valores utilizados posteriormente. É nesta função que está implementado o método Frontal (IRONS, 1970), ela percorre cada elemento da malha montando a sua matriz elemental seguido da montagem da contribuição do elemento para a matriz Frontal. A matriz Frontal é então verificada a cada iteração do processo para identificar se alguma linha já pode ser eliminada. Caso isto seja verdadeiro, a linha da matriz é escrita em um arquivo e a matriz Frontal que está em memória tem as suas entradas referentes a variável eliminada zeradas. Ao ter uma linha eliminada, poupa-se memória porém gera-se um sobrecusto de escrita em disco.

Para a solução final do sistema os arquivos escritos são utilizados para leitura dos valores sendo lidos no sentido reverso, do fim para o início, ou seja é feito o processo de substituição regressiva.

3.3 Análise de execução do programa

Para a quantificação do percentual de tempo de execução da parte mais custosa do programa foi utilizada a ferramenta `gprof` (GRAHAM; KESSLER; MCKUSICK, 1982). Trata-se de uma ferramenta que permite fazer a análise de execução de um programa, bem como das características internas de sua execução como o número de chamadas de uma função, o tempo de cada chamada e o tempo total envolvido em cada função que faz parte do programa.

Na execução do programa foram utilizados dois casos de teste representados por malhas de elementos finitos de tamanhos diferentes, os dois casos de teste são descritos na Tabela 2. Esta tabela apresenta o número de elementos e nós de cada malha, bem como características da matriz que representa o sistema de equações lineares por elas gerado. A coluna Matriz representa o número de posições que a matriz possui e *Non Zeros* (NNZ) é a quantidade de campos com valores diferentes de zero, ao lado está o custo de memória necessário para armazenar cada uma das matrizes.

Tabela 2 – Malhas de elementos finitos utilizadas

Malha	Elementos	Nós	Matriz	NNZ	Memória
Caso 1	18.363	3.548	50.353.216	189.462 (0,0037%)	384,1 MB
Caso 2	44.811	8.364	279.825.984	450.451 (0,0016%)	2,08 GB

Utilizando estes casos de teste, foi possível realizar execuções experimentais e observando os resultados, foi possível identificar que cerca de 99,5% do tempo total de execução do programa RAFEM se encontra na função `VFrontalMethod`, representada pelo retângulo destacado na Figura 6, sendo este tempo total de execução 1 hora e 23 minutos para o Caso 1 e 20 horas e 27 minutos para o Caso 2. Nesta função é que são realizadas as operações de montagem e solução do sistema de equações lineares. Diante disto, estas duas etapas devem ser focadas para se obter maiores ganhos de desempenho.

Podemos destacar algumas estratégias para reduzir o tempo de execução do programa tendo como foco as duas operações mencionadas. Para a montagem do sistema de equações lineares é possível explorar o paralelismo entre os elementos da malha, uma vez que cada elemento possui uma contribuição para a formação do sistema de equações lineares e calcular esta contribuição não é uma tarefa trivial. Também é importante considerar manipular a matriz utilizando formatos de representação para matrizes esparsas como COO e CSR, visto que o custo de memória para representá-la no formato 2D é bastante elevado e que a grande maioria de seus componentes são nulos.

Já para a etapa de solução do sistema de equações lineares existem diversas opções de métodos e técnicas possíveis de se aplicar. Atualmente existem diversas bibliotecas especializadas em resolver este tipo de problema, que implementam vários métodos diferentes dentre métodos de solução diretos e iterativos. Tais bibliotecas são focadas em desempenho, portanto exploram o paralelismo nos métodos que implementam tanto em CPU quanto em GPU. Assim uma boa opção para obter um melhor desempenho seria trocar o método Frontal que atualmente é utilizado no RAFEM e que apresenta algumas limitações como as escritas em disco que realiza, por algum outro método que resolve o mesmo problema mas de forma mais eficiente, isto é, métodos que exploram aspectos como paralelismo, uso eficiente de memória e o uso de GPUs.

3.4 Balanço do Capítulo

Nesse capítulo foi apresentada a aplicação RAFEM e como ela trata o problema de simulação do procedimento de RFA. A aplicação também foi descrita em termos de estrutura de código, mostrando como é o seu fluxo de execução e as partes críticas deste fluxo que demandam maior tempo computacional, sendo estas a montagem do sistema de equações lineares e sua solução. Uma análise foi feita utilizando a ferramenta `gprof` e usando dois casos de teste diferentes para avaliar o tempo de execução da aplicação e realizar o *profiling*, com isto foi possível ver que as partes mais custosas demandavam cerca de 99,5% do tempo de execução.

Com estas etapas conhecidas e seus respectivos custos computacionais mensurados, devemos focar os esforços para reduzir o tempo de execução da aplicação nestas etapas. Para alcançar este objetivo de redução do tempo computacional foram listadas algumas estratégias que podem ser benéficas para a aceleração da aplicação, como otimizações em termos de uso de memória para e o paralelismo para a etapa de montagem e o uso de bibliotecas aceleradas por GPU que implementam métodos numéricos eficientes para a solução.

4 TRABALHOS RELACIONADOS

Neste capítulo são evidenciados os trabalhos que de alguma forma se relacionam com o que está sendo proposto nesta monografia, eles foram divididos em duas categorias: (1) trabalhos realizados diretamente ligados ao RAFEM e (2) trabalhos que envolvam a utilização de GPU para a aceleração de programas que utilizam MEF.

4.1 Melhoria de desempenho no RAFEM

O trabalho de Kapelinski (2016) é voltado para a redução do tempo total de processamento do programa RAFEM, para tanto foi utilizada a API OpenMP. No trabalho é realizada uma análise do programa, discutindo sua estrutura e as possíveis áreas do código que podem ser paralelizadas, o que resulta na identificação da etapa mais custosa como a função de montagem e solução do sistema de equações. A partir disto, realizou-se uma análise mais profunda dos passos envolvidos na montagem do sistema onde se chegou a um laço que continha três laços internos que foram o foco dos esforços para se paralelizar o código. Por fim, foram utilizadas diretivas OpenMP para paralelizar a parte da montagem do sistema, como resultado foi gerada uma versão paralela do RAFEM reduzindo em até 4 vezes o tempo total de execução.

No trabalho de Miletto e Schepke (2017), o objetivo foi obter uma redução de tempo para a obtenção da solução do sistema de equações lineares, onde é explorado um outro método para a solução. Foi utilizado o método Multi Frontal (DUFF; REID, 1983) implementado pela biblioteca *MUltifrontal Massively Parallel Solver* (MUMPS) (AMESTOY et al., 2001), que é uma variação do método Frontal que permite explorar o paralelismo das operações organizando as tarefas em uma árvore de subsistemas lineares, além de utilizar outras bibliotecas para a execução de operações algébricas de forma eficiente. Para isto foi necessário adaptar e gerar uma outra versão do código do RAFEM que gerasse a matriz de rigidez global completa de acordo com as especificações da biblioteca MUMPS. Embora o trabalho não tenha explorado todos os recursos que a biblioteca oferece como o uso de processos *Message Passing Interface* (MPI) em um ambiente distribuído, com esta abordagem foi possível reduzir em até 14 vezes o tempo total de simulação em um ambiente de memória compartilhada.

Mesmo se obtendo uma redução do tempo total, ainda existem outras alternativas que podem reduzir também o tempo execução e que não foram abordadas nos trabalhos apresentados até agora. Entre as alternativas podemos citar o uso de GPU tanto para a etapa de montagem quanto para a de solução do sistema de equações lineares. A partir disto é possível explorar diferentes métodos e bibliotecas que permitam paralelizar as etapas de montagem e solução do sistema fazendo o uso de GPU. Por este motivo, a próxima seção apresenta os trabalhos que fazem o uso de GPUs em programas que utilizam MEF.

4.2 Uso de GPU no Método dos Elementos Finitos

O trabalho de Georgescu, Chow e Okuda (2013) trata-se de uma pesquisa para reunir os trabalhos que se encontram na literatura para a aceleração das etapas que consomem a maior parte do tempo em programas de MEF através do uso de GPUs. Destacam-se nos trabalhos encontrados os resultados obtidos e eventuais pacotes de *software* ou bibliotecas que foram utilizados, provendo uma visão geral do estado da arte nesta área. No artigo é discutida a melhoria de desempenho de programas de MEF usados para a análise de estruturas de uma forma geral, partindo da estruturação e simplificação da malha de elementos finitos na etapa de pré-processamento até a etapa de montagem e solução das equações dos elementos.

Segundo os autores, a etapa de montagem da matriz global de rigidez é dividida em dois passos. A primeira é a montagem das matrizes elementais, que é uma tarefa inerentemente paralela, uma vez que não existe dependência entre os elementos para este procedimento. Deste modo, é possível montar todas as matrizes elementais de rigidez em paralelo. Para o segundo passo, o mapeamento das matrizes elementais para a matriz global, que é descrito como uma etapa de intenso uso de dados, é preciso tomar cuidado com as condições de corrida que podem surgir devido a contribuição de diferentes elementos na mesma posição da matriz global. Os autores também destacam o caso de aplicações de MEF dinâmicas, onde a montagem da matriz deve ser refeita a cada passo de tempo da simulação.

Para a etapa de solução dos grandes e esparsos sistemas de equações provenientes de problemas de MEF, os autores destacam que existem duas alternativas: os métodos de solução diretos e os métodos de solução iterativos. A partir desta pesquisa foi possível encontrar alguns trabalhos relacionados dentre os que são destacados na sequência desta seção.

No trabalho de Filipovic, Peterlik e Fousek (2009) é realizado um estudo a respeito da aceleração da etapa montagem das equações de um sistema em um problema de *Saint Venant-Kirchhoff* utilizando uma GPU, que é um modelo usado para representar materiais hiperelásticos. No trabalho, os autores afirmam que existem duas tarefas custosas no MEF, que são a montagem do sistema de equações e a resolução do mesmo. Para obter esta aceleração é utilizada a API CUDA. Durante o trabalho é realizada uma análise de aspectos que prejudicam o desempenho de uma aplicação acelerada por GPU, como a carga de trabalho fornecida em relação ao sobrecusto das transferências de dados entre o *host* e o *device* e as estratégias para manipulação de memória dentro do dispositivo.

Em relação à carga de trabalho, é destacada que geralmente são utilizadas duas abordagens para se obter uma boa aceleração utilizando GPUs considerando a divisão de tarefas baseadas pela sua granularidade:

- Granularidade Grossa - São mapeadas para blocos de *threads* onde cada bloco é

subdividido em operações de granularidade fina para serem resolvidos por *threads*;

- Granularidade Fina - São resolvidas por *threads* individuais, sem papel significativo de blocos de *threads*.

Porém para o problema de montagem do sistema estudado existem várias operações de granularidade média, ou seja, a montagem de uma matriz de um elemento é muito complexa para apenas uma *thread*, ocupando muita memória, ao mesmo tempo que esta tarefa não pode ser paralelizada por um bloco de *threads* devido a subutilização da capacidade de um bloco.

Assim o processo de montagem foi adaptado para gerar carga de trabalho suficiente para obter ganho de desempenho tendo uma boa utilização de GPU, usando a montagem das matrizes de múltiplos elementos simultâneos e estratégias de memória compartilhada entre as *threads* de um bloco. Como resultados do trabalho, os autores apresentam duas versões para a etapa de montagem, uma utilizando memória compartilhada e outra não. Os resultados para a versão que utiliza memória compartilhada superam em até 15 vezes a versão serial utilizando uma CPU.

Dziekonski et al. (2012) apresenta uma técnica eficiente para a montagem das matrizes envolvidas em um problema de eletromagnetismo utilizando GPU. Os autores dividem as simulações computacionais de MEF em quatro passos:

- Discretização do domínio;
- Integração numérica, corresponde a montagem das matrizes locais de cada elemento;
- Montagem da matriz, construção da matriz global a partir das matrizes locais;
- Solução do sistema de equações lineares.

Destes passos, a computação da integração numérica e montagem da matriz é estudada neste trabalho. É apresentada uma discussão sobre trabalhos relacionados envolvendo a etapa de integração numérica, onde é apontando o problema de limite de recursos por *thread*. Para a etapa de montagem da matriz também são apresentados trabalhos relacionados onde as condições de corrida na montagem e o uso eficiente de memória global e memória compartilhada são evidenciadas pelo autor.

Duas malhas de elementos finitos são utilizadas no trabalho, uma com 13.613 e outra com 28.806 elementos tetraedrais. No processo de integração numérica é criado um *kernel* para um laço externo que percorre todos os elementos, possibilitando o processamento dos elementos finitos em paralelo. Também foi criado um *kernel* para um laço interno que realizava operações com matrizes. Inicialmente foi utilizada a biblioteca CUBLAS, porém os autores afirmam que a biblioteca não apresenta bom desempenho para matrizes de ordem menor que 100, que é o caso das matrizes dos elementos envolvidas no laço interno. Para contornar o problema foi implementada um *kernel* para as

operações com matrizes dedicados a tamanhos específicos de matrizes utilizando memória compartilhada.

Na etapa de montagem da matriz global as matrizes geradas na etapa de integração numérica devem ser agrupadas. Isto também é feito na GPU, onde não existe o custo de transferência de memória porque as matrizes elementais montadas na etapa anterior já se encontram no dispositivo. Nesta etapa é levada em consideração o custo de armazenamento em memória demandado pela matriz global, onde para reduzi-lo são utilizados os formatos COO e CSR, que omitem os valores nulos na representação das matrizes esparsas. Primeiramente a matriz global é escrita no formato COO, já que é provado que a montagem direta em formato CSR usando GPU é ineficiente (DZIEKONSKI et al., 2012). Para tanto são criados dois *kernels*: um para reunir informações dos valores não nulos e o outro para montar definitivamente a matriz global em formato COO.

Por fim é realizada uma etapa de conversão do formato COO para o CSR que segundo os autores é 30% mais eficiente em termos de armazenamento de memória. A conversão é realizada utilizando operações atômicas sobre a leitura da matriz em formato COO. A vantagem de ter a matriz neste formato é que ela já se encontra no formato requerido por bibliotecas que resolvem sistemas lineares esparsos como a biblioteca CUSPARSE. Entretanto, o formato COO também é utilizado por algumas bibliotecas.

Durante a execução, a etapa de integração numérica tomou maior parte do tempo (87%), enquanto a montagem da matriz global em formato COO levou 4%, restando 9% do tempo para a conversão para CSR. Por fim são discutidas as diferentes versões de código geradas ao longo do trabalho e são apresentados os resultados de *speedup* para a geração da matriz global. A versão mais otimizada tem um *speedup* de 81 vezes em relação a versão sequencial do código para a etapa de integração numérica.

Este estudo foi continuado por Dziekonski, Lamecki e Mrozowski (2016) expandindo o uso de GPU para a etapa de solução do sistema de equações lineares. Para isto foi feito o uso do método do gradiente conjugado juntamente com uma série de pré-condicionadores e o uso das bibliotecas CUBLAS e MKL PARDISO.

A partir de um conjunto de testes os autores chegaram a conclusão que para a fase de solução de sistemas com um número de elementos menor que 2 milhões, é mais vantajoso usar um método direto em CPU como o oferecido pela biblioteca MKL PARDISO do que um método iterativo em GPU. Em contrapartida, para sistemas com mais de 2 milhões de entradas, a versão em GPU se sai melhor. Como resultado o tempo total necessário para a análise MEF realizada pelo programa estudado foi reduzida em 4.7 vezes.

4.3 Análise dos trabalhos relacionados

Nesta seção é feita uma análise dos trabalhos relacionados descritos na seção anterior com o objetivo de identificar a contribuição de cada um para o trabalho que está

sendo proposto. Para isto foi organizada a Tabela 3 considerando os itens de acordo com as etapas de montagem e solução presentes nos trabalhos relacionados de MEF e a aplicação RAFEM. Assim, foram definidas as seguintes comparações:

- Montagem CPU, se o trabalho explora o paralelismo durante a etapa de montagem utilizando CPU;
- Solução CPU, se o trabalho explora o paralelismo durante a etapa de solução utilizando CPU;
- Montagem GPU, se o trabalho explora o paralelismo durante a etapa de montagem utilizando GPU;
- Solução GPU, se o trabalho explora o paralelismo durante a etapa de solução utilizando GPU;
- RAFEM, se o trabalho é diretamente relacionado com a aplicação RAFEM.

Tabela 3 – Comparação de trabalhos relacionados

Trabalho	Montagem CPU	Solução CPU	Montagem GPU	Solução GPU	RAFEM
Kapelinski (2016)	X	–	–	–	X
Miletto e Schepke (2017)	–	X	–	–	X
Filipovic, Peterlik e Fousek (2009)	–	X	X	–	–
Dziekonski et al. (2012)	–	–	X	–	–
Dziekonski, Lamecki e Mrozowski (2016)	–	X	X	X	–
Este Trabalho	–	–	X	X	X

A solução proposta neste trabalho será aplicada no RAFEM, onde pretende-se aplicar o paralelismo em GPU tanto na etapa de montagem como na etapa de solução. Esta abordagem ainda não foi aplicada ao RAFEM, mas já foi avaliada em outros trabalhos relacionados.

5 METODOLOGIA

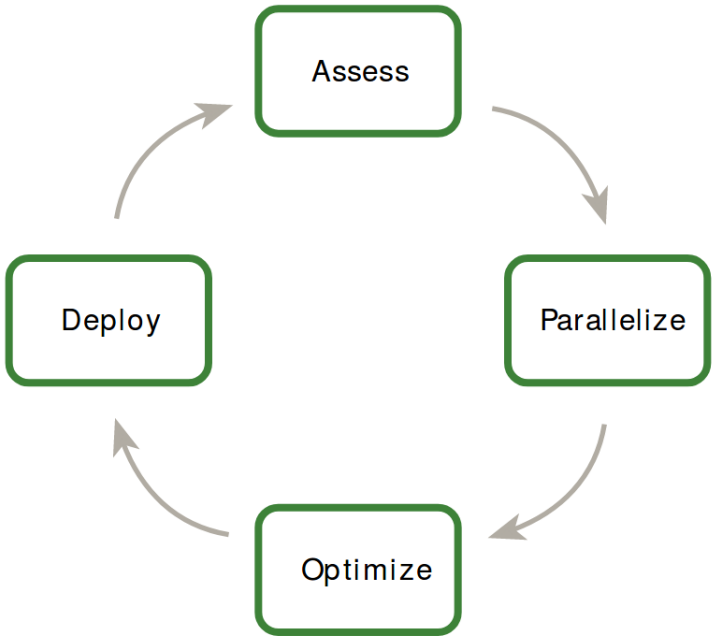
Este capítulo descreve como foi o processo para alcançar um melhor desempenho para o programa RAFEM através do uso de GPU. Aqui serão estabelecidas as estratégias para alcançar o objetivo final, juntamente com a escolha e especificação de uma série de ferramentas que irão auxiliar no processo. Por fim são descritas as versões de código geradas para o programa juntamente com uma discussão dos resultados obtidos em cada versão.

5.1 Estratégias de Desenvolvimento

Como o código do programa RAFEM não foi desenvolvido inicialmente pensando em ser executado em sistemas paralelos, as estruturas de dados utilizadas, a movimentação de dados e o fluxo do programa em si não são os mais adequados para a computação paralela. Isso afeta principalmente o uso de GPUs, onde existe a necessidade de transferência de dados do *host* para o *device*, o que deve ser feito de forma eficiente.

Uma boa prática é adotar algum tipo de processo de desenvolvimento que possa guiar a tarefa de paralelização do código existente, para torná-la menos difícil, porém não menos desafiadora. Levando este ponto em consideração, o método *Access, Parallelize, Optimize, Deploy* (APOD) (NVIDIA, 2018a) ou avaliar, paralelizar, otimizar e implantar, descrito na Figura 7, será utilizado durante o desenvolvimento deste trabalho.

Figura 7 – Etapas do ciclo APOD



Fonte: NVIDIA (2018a)

O processo de desenvolvimento é executado de forma cíclica, onde no decorrer de cada ciclo os quatro passos do processo são abordados. A ideia do processo é identificar inicialmente uma parte de código que pode ser acelerada por GPU, desenvolver uma versão de código paralelo inicial aplicando otimizações para gerar alguma melhoria no código, antes de implantar uma nova versão do código e retornar ao início do processo novamente enquanto melhorias ainda forem possíveis. A seguir os passos do processo APOD são discutidos em mais detalhes.

1. **Avaliar** - esta etapa consiste em avaliar e analisar a execução da aplicação em um cenário real, para que se possa coletar os dados necessários para a identificação das partes do programa que mais contribuem para o tempo de execução total. Também pode ser realizada uma análise de forma a estimar o valor de *speedup* que pode ser obtido de acordo com as leis de Amdahl e Gustafson (GUSTAFSON, 1988). A avaliação de uma aplicação é apoiada pelo uso de ferramentas chamadas *profilers*. Estas ferramentas permitem fazer a instrumentação do código que está sendo avaliado para obter dados de consumo de memória, tempo de execução além de outras métricas relevantes para a análise da execução de um programa. As ferramentas que foram utilizadas neste trabalho são descritas na seção 5.2.

2. **Paralelizar** - depois de identificar os pontos do programa onde se concentra a maior parte do tempo de execução, é realizado o estudo para a paralelização do código. Nesta etapa podem ser utilizadas bibliotecas aceleradas por GPU, diretivas para o pré-processador indicando áreas de código paralelizáveis ou por meio da programação de GPU através de uma API para programação paralela como OpenCL e CUDA. Entretanto, para alguns casos é necessário realizar refatorações no código sequencial para que seja possível criar uma primeira versão de código paralelo.

3. **Otimizar** - depois de realizada a etapa de paralelização, pode ser necessário realizar etapas de otimização para melhorar o desempenho obtido. As otimizações podem ser implementadas aos poucos e de forma incremental devido a característica iterativa do processo. Assim, o programador não precisa se preocupar com todas a possibilidades de otimização de uma única vez. Dentre as otimizações possíveis podemos destacar a gestão eficiente de memória considerando estratégias como a sobreposição das tarefas de transferência e execução, o uso de memória compartilhada na GPU e estratégias para otimizar o acesso à memória global do dispositivo. Um ponto muito importante nesta etapa é realizar a verificação dos resultados que estão sendo produzidos por estas novas versões geradas, para que estes estejam consistentes com os resultados de versões anteriores.

4. **Implantar** - a implantação de uma versão de código gerado durante o processo é uma etapa bastante importante. Ela pode feita mesmo que apenas otimizações

parciais tenham sido elaboradas até o momento de implantação, pois mesmo que tenha se obtido um aumento de desempenho parcial, ele ainda é válido (NVIDIA, 2018a). Com essa abordagem de implementações parciais também é possível reduzir os riscos que grandes mudanças podem trazer. Após uma versão ter sido implantada o processo entra em um novo ciclo, passando novamente por todas as etapas.

O processo APOD foi escolhido por permitir que, durante este trabalho, melhorias no código possam ser incrementalmente adicionadas de uma forma padronizada que reduza os riscos. As diferentes versões de implantações geradas ao longo do processo serão mantidas em um repositório de controle de versionamento para que estejam acessíveis a qualquer momento, para facilitar a tarefa de comparação entre versões ou para auxiliar em casos que seja necessário reverter modificações.

Outra estratégia que é adotada neste trabalho é explorar o uso de bibliotecas aceleradas por GPU. O uso de bibliotecas pode facilitar a tarefa de paralelização e otimização de código, o que é feito com relativamente pouco esforço quando comparado com o trabalho envolvido na implementação específica de uma solução para um problema já bastante estudado como o de resolver sistemas de equações lineares.

Para o trabalho proposto são consideradas as seguintes bibliotecas de acordo com as suas possíveis contribuições para o trabalho:

- **CUSOLVER** - é uma biblioteca baseada nas bibliotecas CUSPARSE e CUBLAS que reúne diversos métodos acelerados por GPU para a solução de sistemas de equações lineares tanto densos quanto esparsos. A biblioteca oferece alguns métodos iterativos e diretos para obter a solução dos sistemas de equações, além de oferecer rotinas de reordenação e pré-condicionamento destes sistemas. Esta biblioteca pode ser explorada para a etapa de solução do sistema de equações lineares gerados no programa RAFEM.

- **MAGMA** - A biblioteca MAGMA (*Matrix Algebra for GPU and Multicore Architectures*) (TOMOV; DONGARRA; BABOULIN, 2010) tem como objetivo fornecer uma biblioteca de álgebra linear com algoritmos que cheguem em uma solução precisa no menor tempo possível. Para atingir este objetivo a biblioteca explora as arquiteturas heterogêneas, oferecendo algoritmos paralelizados tanto em CPU quanto em GPU. Oferece métodos e rotinas semelhantes aos que a biblioteca CUSOLVER disponibiliza. Também faz o uso das bibliotecas CUSPARSE e CUBLAS e ainda possui a biblioteca MAGMA BLAS que serve como uma biblioteca complementar a CUBLAS. Uma vantagem da biblioteca MAGMA em relação a biblioteca CUSOLVER, é que ela fornece uma variedade maior de métodos implementados para GPU como por exemplo o método GMRES(m) escolhido neste trabalho só é implementado pela biblioteca MAGMA.

- **Thrust** - esta biblioteca é baseada na biblioteca *Standard Template Library* (STL) da linguagem `C++`. Focada em fornecer mais flexibilidade e produtividade para a programação em GPU, a biblioteca oferece um conjunto de algoritmos paralelos além de simplificar a gestão de memória e as transferências entre *host* e *device* no ambiente de programação CUDA.

5.2 Ferramentas

Durante o desenvolvimento de uma aplicação paralela existem diversas ferramentas que auxiliam no melhor andamento desta tarefa. Principalmente quando se está utilizando um processo de desenvolvimento para obter o melhor desempenho possível em um código paralelo. Visto que tornar um código sequencial em um código paralelo não é uma tarefa trivial (RAEDER, 2014), o uso de determinadas ferramentas pode fazer com que esta tarefa realizada feita com êxito de forma mais eficiente.

A seguir são destacadas algumas ferramentas escolhidas para serem utilizadas ao longo do desenvolvimento do trabalho e do ciclo do processo APOD. O conjunto de ferramentas foi escolhido de acordo com algumas funcionalidades oferecidas como o *profiling* de aplicações, a depuração de código tanto em CPU quanto em GPU, a gerência de memória, controle de versionamento e verificação da corretude dos resultados numéricos obtidos.

- **Kanban** - este método serve para delimitar e organizar as tarefas a serem feitas durante o trabalho. Principalmente nas tarefas de desenvolvimento do código, onde é relatado que o uso dessa técnica possibilita obter uma melhor qualidade de *software* ao melhorar a coordenação do processo de desenvolvimento (AHMAD; MARKKULA; OIVO, 2013). Para isto, existem diversas ferramentas disponíveis *online* que possibilitam realizar a aplicação do método Kanban.

- **gprof e nvprof** - estas duas ferramentas são importantes para a fase de avaliação da aplicação em um cenário de teste real. Com elas é possível coletar os dados da aplicação durante a sua execução, como o número de chamadas de função, o tempo de execução, dados sobre a execução de *kernels* e informações sobre transferências de memória entre *host* e *device*.

- **cuda-gdb** - provê a funcionalidade de depuração de código para o tratamento e identificação de erros de programação que podem ocorrer. Ela permite depurar o código tanto em CPU quanto em GPU, possibilitando ainda a visualização das variáveis identificadoras da hierarquia de *threads* CUDA.

- **cuda-memcheck** - esta ferramenta permite encontrar erros relacionados a memória da GPU como acessos fora do limite de uma variável nos diferentes níveis da hierarquia de memória e erros de alocação e desalocação de memória, dentre outras funcionalidades.

- **nvcc** - é o compilador para os programas escritos utilizando CUDA. Ele trabalha em conjunto com os compiladores de linguagem C ou C++ presentes no ambiente de execução. O nvcc separa o código do *host* e do *device* sendo o código do *device* compilado pelo nvcc e o do *host* pelos outros compiladores.

- ***Script* para verificação de resultados** - mais importante que acelerar o código é manter a consistência dos resultados numéricos que o código do programa original gera. Para realizar esta verificação será implementado um *script* para a verificação e comparação dos resultados dos valores finais de temperatura e tensão gerados pelo programa em cada passo de simulação. O *script* servirá para encontrar possíveis discrepâncias entre duas soluções obtidas para um mesmo caso de teste.

- **Repositório de controle de versionamento** - para manter um controle das alterações de código realizadas ao longo do trabalho será adotada uma ferramenta para realizar o controle de versionamento. A cada etapa de implantação do processo APOD será criada uma *tag*, identificando como uma versão estável do programa. Ao fim do processo, será possível ter acesso à todas as versões estáveis para fins de comparação de resultados.

5.3 Aplicação do processo APOD

Para iniciar a aplicação do processo APOD foram definidos os casos de teste. Foram empregadas duas malhas de tamanhos diferentes definidas na Tabela 2, elas serão referenciadas ao longo do trabalho de acordo com a nomenclatura definida na tabela: Caso 1 para a malha menor e Caso 2 para a maior. As malhas são formadas por elementos tetraedrais, criadas a partir de modelos de fígados reais e foram geradas utilizando o *software* ANSYS *release* 12.1.

O ambiente computacional utilizado neste trabalho para os testes de execução é melhor descrito nas tabelas 4 e 5. Ele é formado por dois processadores Intel Xeon CPU E5-2650, e uma GPU Nvidia Quadro M5000.

Tabela 4 – Ambiente de execução: CPU

Características	**Xeon E5-2650 (×2)**
Frequência	2.00 GHz
Núcleos	8 (×2)
Threads	16 (×2)
Cache L1	32 KB
Cache L2	256 KB
Cache L3	20 MB
Memória RAM	128 GB

Tabela 5 – Ambiente de execução: GPU

Características	**Quadro M5000**
Frequência	1.04 GHz
CUDA *cores*	2048
Cache L1	64 KB
Cache L2	2 MB
Memória Global	8 GB

Para iniciar o processo, foi realizada uma avaliação inicial do código original da aplicação usando a ferramenta `gprof`. Com isto foi identificado que a maior parte do tempo de execução envolvido em uma simulação se encontra na etapa de montagem e solução do sistema de equações lineares. Além do fato destas etapas envolverem diversas operações computacionais, elas se encontram em uma parte iterativa do programa, sendo chamadas diversas vezes durante a execução do programa RAFEM. De acordo com os resultados da ferramenta, são 278 chamadas para o Caso de teste 1 e 974 para o Caso 2.

Para uma melhor amostragem do tempo de execução do programa foram realizadas algumas execuções a fim de obter a média dos tempos para cada um dos casos de teste. Foram realizadas 30 execuções para o Caso de teste 1 e 15 para o Caso 2, a média dos resultados das execuções obtidas foram de 1 hora e 23 minutos para o Caso 1 e de 20 horas e 27 minutos para o Caso 2.

5.4 Balanço do Capítulo

Nesse capítulo foram apresentados os aspectos metodológicos para o desenvolvimento do trabalho. Foram descritas as ferramentas, bibliotecas de código e estratégia de otimização e paralelização a ser adotada adotada. Apresentamos também o ambiente de testes e um *profiling* da aplicação.

Em relação aos tempos de execução sequenciais obtidos para os dois casos de teste, pode-se perceber que eles são expressivos. Para este trabalho pretende-se reduzir o tempo de execução para valores viáveis para o tratamento em nível clínico de maneira imediata.

6 EXPERIMENTAÇÃO

As seções seguintes descrevem as versões paralelas do RAFEM geradas ao longo da execução do processo APOD. No fim de cada seção serão apresentados os resultados obtidos no presente trabalho. Serão evidenciados tanto os resultados representando a *performance* do programa quanto a qualidade dos seus resultados numéricos quando comparados com os resultados gerados pela versão original do programa.

6.1 Versão 1: Montagem da matriz dos elementos e solução

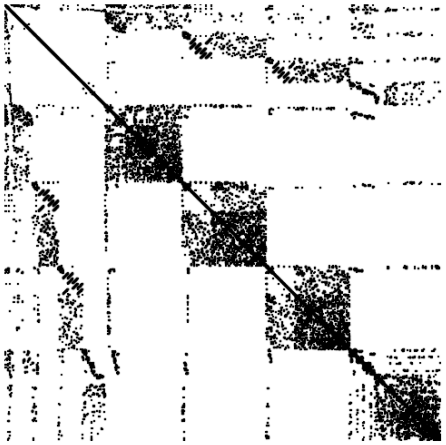
De acordo com o estudo de trabalhos relacionados realizado, a análise do código do programa e as informações coletadas na etapa de avaliação, sabe-se que é possível paralelizar as duas etapas mencionadas. Assim, os esforços para a paralelização do programa RAFEM serão focados primeiramente na etapa de montagem do sistema de equações lineares, que se encontra no arquivo `VFrontalMethod`.

Antes de gerar código paralelo, é necessário fazer uma refatoração no código. Como o RAFEM utiliza o método Frontal para a solução do problema, a matriz global não chega a ser montada completamente, o que é um requisito para poder utilizar as rotinas de solução oferecidas pela biblioteca MAGMA. Sendo assim, faz-se necessário alterar o código para que este gere a matriz completa contendo todas as contribuições das matrizes elementais. Ainda dentro deste processo de refatoração, é possível identificar e eliminar variáveis não utilizadas pelo programa, de forma a otimizar o código e facilitar futuras depurações. Para facilitar a tarefa de remoção das variáveis não utilizadas, o programa foi compilado utilizando a *flag* de compilação `-Wall` que ativa os avisos para alguns casos como variáveis não utilizadas.

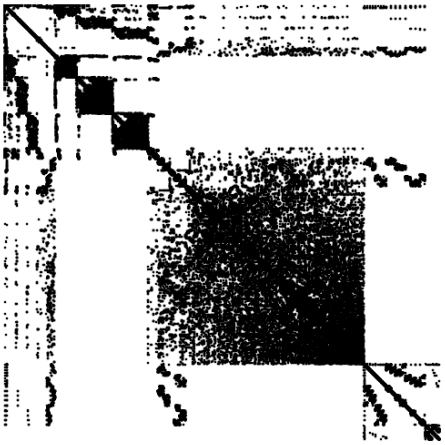
O primeiro passo em direção a paralelização do código foi a refatoração da estrutura da função de montagem. A variável que representava a matriz Frontal e as demais variáveis relacionadas com ela foram substituídas por uma representação da matriz global completa. Os descritores de arquivos utilizados pelo método Frontal também foram eliminados desta etapa do código. Na sequência foi realizada a substituição na adição das matrizes dos elementos para a matriz global utilizando uma estrutura de mapeamento por meio de vetores já definida no código original. Através dela é possível calcular a posição da adição de cada nó de um elemento na matriz global. Com isto foram geradas as matrizes globais para os dois casos de teste. Pode-se perceber que as matrizes geradas são esparsas conforme ilustrado pela Figura 8, onde os pontos em preto representam valores não nulos em uma dada posição da matriz. Para o Caso de teste 1 da Tabela 2 representado na Figura 8(a) é gerada uma matriz quadrada de dimensão 7.096 onde apenas 188.799 de suas entradas não são nulas. Para a malha maior representada pelo Caso de teste 2, temos 446.730 valores não nulos em uma matriz de dimensão 16.728 representada na Figura 8(b).

Após as substituições realizadas no código sequencial, iniciou-se a implementação de um *kernel* CUDA para realizar esta etapa. Durante este processo foram identificadas

Figura 8 – Matrizes globais dos dois casos de teste



(a) Caso 1 - Malha Menor



(b) Caso 2 - Malha Maior

algumas variáveis que são usadas como somatórios dentro do laço que percorre todos os elementos da malha e que são posteriormente utilizadas. Este tipo de problema pode ser facilmente resolvido com a troca da variável por um vetor, onde cada *thread* escreve sua contribuição em uma posição distinta e, no fim do processo, é aplicada uma operação de redução. Assim foram utilizadas as rotinas `thrust::reduce()`, disponíveis na biblioteca *thrust* para realizar este tipo de operação.

A função deste *kernel* é dividir o trabalho de montagem das matrizes dos elementos entre *threads* de forma que cada *thread* seja responsável pela montagem da matriz de um elemento. Assim um número E de elementos pode ser processado de forma simultânea. Este número E dependerá da quantidade de recursos disponíveis para cada bloco de *threads*. Ou seja, pode não ser possível que um bloco contenha o seu número máximo de *threads* (1024). Isto porque as *threads* de um bloco dividem os recursos do SM em que serão executadas.

Para organizar a execução do *kernel*, foi utilizada uma estrutura de dados denominada `element_descriptor` que representa um elemento e a sua contribuição. Esta estrutura possui um espaço para a matriz elemental além de outras variáveis como identificadores de posição do elemento e seus nós. Isto serve para facilitar o controle e possibilitar a localização da posição da matriz global em que deverá ser realizada a contribuição de um elemento a partir da própria estrutura de dados. Assim o *kernel* retorna um vetor contendo estas estruturas para uma próxima etapa que realizará a montagem da matriz global. A estrutura usada para agrupar as informações dos elementos é apresentada na Figura 9.

Ao fim da execução deste *kernel*, um vetor da estrutura `element_descriptor` se encontra disponível em GPU para que a montagem da matriz global seja realizada. Para isto a estrutura contém a matriz `Kelem`, do tipo `kelem_t`, contendo um valor do tipo `double`, a linha e coluna da matriz global onde este valor deve ser adicionado, os vetores

Figura 9 – Estrutura de dados utilizada para representar um elemento e sua relação com a matriz global

```
1  // struct for Kelem matrix
2  typedef struct {
3      double val;
4      int global_row;
5      int global_col;
6  } kelem_t;
7
8  // struct to organize data for global matrix assembly
9  typedef struct {
10     int elem_id;              // element id
11     int node_number;          // number of nodes
12     int node_ids[4];          // node ids
13     kelem_t Kelem[8][8];      // Kelem -> A
14     double Belem[8];          // Belem -> B
15
16     int index_holder[4];              //  dR index control
17     double dRvtdLamda_values[4];      //  dRvtdLamda -> A[i][n]
18     double dRpowerdVT_values[4];      //  dRpowerdVT -> A[n][i]
19 } element_descriptor;
```

`dRvtdLamda_values` e `dRpowerdVT_values` também contribuem para a montagem da matriz global e são adicionados ao longo da última coluna e da última linha. Já o vetor `Belem` contribui para a montagem do vetor de termos independentes b. Mesmo dispondo de todas estes dados em GPU, nesta primeira versão do código paralelo, a montagem ainda não é feita totalmente em GPU, isso será trabalhado em uma futura iteração do processo.

O próximo passo depois da matriz global ter sido montada é obter a solução do sistema de equações que ela representa. Para isto, é utilizada da biblioteca MAGMA e do método numérico GMRES(m), que como vimos anteriormente necessita da configuração de um parâmetro indicando quando o método é reiniciado, também deve-se informar um valor de tolerância entre a diferença da solução prevista e a solução exata do sistema que é utilizado como critério de parada, utilizou-se o valor 10^{-10} para garantir uma boa precisão. Para escolher um parâmetro que proporcionasse um bom resultado alguns valores foram explorados para os dois casos de teste, primeiramente o método foi executado com um m do tamanho da matriz, para que funcionasse como o GMRES padrão e para que pudéssemos comparar os próximos valores escolhidos de m. Para o Caso 1, obter a solução uma única vez com esta configuração levou cerca de 9 minutos e para o Caso 2 o processo de solução foi interrompido após 1 hora de execução pois já estava ocupando 7GB de memória da GPU. Isto torna essa configuração impraticável, pois a solução do sistema é realizada iterativamente ao longo da execução do programa e com este tempo para uma solução, demoraria mais do que a versão original. Assim foram testados alguns valores até chegar nas configurações $m = 80$ para o Caso 1 e $m = 150$ para o Caso 2 que apresentaram bons resultados.

Também é importante configurar um pré-condicionador para o sistema de equações. Os pré-condicionadores tem o papel de melhorar a taxa de conversão dos métodos iterativos, alterando algumas características do sistema de equações. Dentre as opções disponíveis pela biblioteca MAGMA o pré-condicionador ILU (*Incomplete LU*) foi escolhido. Após esta configuração, para obter a solução do sistema montado, ele foi descrito usando uma estrutura de dados definida pela biblioteca MAGMA para a representação de matrizes esparsas no formato CSR, a própria biblioteca oferece uma rotina para a conversão de uma matriz para o formato, no entanto ela só foi implementada para executar em CPU.

Nesta primeira etapa, não houve nenhuma medida envolvendo otimizações, deixando este trabalho para iterações futuras do processo. Para fins de comparação com as demais versões geradas, esta primeira versão será denominada **Versão 1**, e as versões posteriores a ela serão numeradas em sequência. Os resultados numéricos gerados foram testados através de comparações por meio do *script* desenvolvido para esta tarefa. Para a obtenção de resultados comparativos do desempenho da aplicação, as execuções das versões paralelas em GPU terão a sua *performance* comparada com execuções da versão original do código sequencial executado em CPU no ambiente computacional descrito pelas Tabelas 4 e 5.

Assim, os resultados para verificar o desempenho da **Versão 1** foram obtidos calculando o tempo médio de execução levado pela aplicação nos casos de teste definidos anteriormente. Para o Caso 1, foram realizadas 30 execuções e a média do tempo foi de 8 minutos e 31 segundos. Para o Caso 2, foram realizadas 15 execuções e o tempo obtido foi de 150 minutos e 32 segundos. Com a implementação desta primeira versão de código paralelo, foi possível atingir uma redução de aproximadamente 9 vezes no tempo total da aplicação.

Para a avaliação da qualidade dos resultados numéricos obtidos, as soluções que o programa RAFEM gera foram comparadas. Esta solução é escrita em um arquivo contendo a descrição da malha, os passos de simulação ao longo do tempo e para cada passo, a temperatura e tensão de cada nó no momento atual. Assim podemos comparar os valores obtidos e verificar se a versão implementada está gerando os mesmos resultados ou resultados próximos aos do código original. Foram comparados os valores calculados para o tempo de simulação de 5 minutos, onde a maior diferença encontrada para os valores de temperatura foi no nó 3527 e a maior diferença de tensão foi no nó 3516 para o Caso 1, ambas as diferenças são muito próximas de zero. Estes valores são apresentados na Tabela 6 onde os nós com as maiores diferenças estão destacados em negrito, bem como a diferença entre os dois valores obtidos.

Estes valores são um bom indício de que a solução fornecida pela versão paralela desenvolvida está correta, uma vez que estamos trabalhando com precisão dupla em GPU e pequenas diferenças são esperadas. Isto porque as operações em ponto flutuante passam

Tabela 6 – Comparação de resultados da Versão 1 com a Original - Caso 1

Nó	Original		Versão 1		Diferença	
	V(Volt)	T(°C)	V(Volt)	T(°C)	V(Volt)	T(°C)
1	1e-07	38.2414	1e-07	38.2414	0.0	0.0
8	12.6745	28.1273	12.6745	28.1273	9.9e-08	0.0
415	2e-07	38.2012	2e-07	38.2012	0.0	0.0
622	25.3580	39.6453	25.3580	39.6453	0.0	1e-07
774	14.5047	20.0045	14.5047	20.0045	0.0	0.0
1510	8.6082	30.9617	8.6082	30.9617	0.0	0.0
2084	5.1542	28.8030	5.1542	28.8030	0.0	0.0
3516	**14.5775**	37.0764	**14.5775**	37.0764	**1e-07**	0.0
3527	10.5233	**20.3770**	10.5233	**20.3770**	0.0	**1e-07**
3548	14.5775	20.3770	14.5775	20.3770	0.0	0.0

por passos de arredondamento e, se um mesmo conjunto de operações for realizado em uma ordem diferente, algo provável em algoritmos paralelos, pode ocorrer deste mesmo conjunto de operações gerar resultados ligeiramente diferentes. Assim, estas discrepâncias não evidenciam automaticamente que o resultado produzido pela GPU está incorreto ou que há um problema na GPU (WHITEHEAD; FIT-FLOREA, 2011). Algo que pode ajudar neste caso é arredondar os valores calculados em alta precisão para uma precisão menor antes de compará-los. Assim, as comparações futuras realizadas neste trabalho irão arredondar os resultados até a quarta casa decimal e usar os valores arredondados para calcular a diferença. Essa métrica foi aplicada na comparação dos resultados da Versão 1 para o Caso 2, apresentados na Tabela 7.

Tabela 7 – Comparação de resultados da Versão 1 com a Original - Caso 2

Nó	Original		Versão 1	
	V(Volt)	T(°C)	V(Volt)	T(°C)
1	30.1317	24.6521	30.1317	24.6521
259	26.0980	20.3643	26.0980	20.3643
1005	25.6495	22.8014	25.6495	22.8014
2011	28.6921	23.5852	28.6921	23.5852
3524	28.2609	20.1560	28.2609	20.1560
4981	29.0731	20.0438	29.0731	20.0438
5613	29.5669	20.5590	29.5669	20.5590
6813	30.1317	20.5760	30.1317	20.5760
7560	30.1317	20.5812	30.1317	20.5812
8364	30.1317	20.4286	30.1317	20.4286

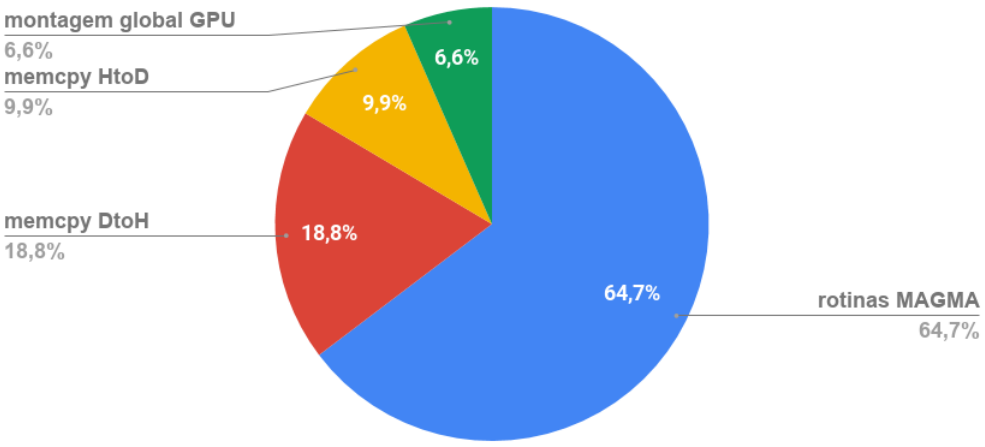
Os resultados obtidos para o segundo caso de teste seguindo o novo critério de comparação de quatro casas decimais demonstram que não houveram diferenças entre os resultados calculados pela versão original dos calculados pela Versão 1 para o Caso 2. Desta forma, foi possível validar a versão do RAFEM desenvolvida neste trabalho, já que a versão original do programa foi validada no trabalho de Jiang et al. (2010)

comparando com a solução obtida no *software* ANSYS. Assim foi finalizado o primeiro ciclo de desenvolvimento do processo APOD resultando na Versão 1, que reduziu 9,7 vezes o tempo de execução para o Caso 1 e 8,1 para o Caso 2, mantendo os resultados consistentes.

6.2 Versão 2: Montagem da matriz global no formato CSR em GPU

Para gerar uma nova versão iniciou-se um novo ciclo de desenvolvimento, assim devemos realizar novamente a etapa de análise da aplicação. Porém, agora o código a ser analisado é a Versão 1 implementada anteriormente, que já é uma aplicação paralela e portanto deve ser analisada com outra ferramenta além do `gprof` utilizado na etapa passada. Assim foi utilizado o *profiler* `nvprof`, pois com ele é possível extrair informações dos *kernels* que estão executando na GPU, bem como detalhes das chamadas da API CUDA como por exemplo transferências de memória entre CPU e GPU. A partir dos dados coletados pelo *profiler* foi gerado o gráfico da Figura 10, onde podemos perceber que existe uma grande porção de tempo de execução em GPU gasto apenas em comunicação entre CPU e GPU através do comando `memcpy`.

Figura 10 – *Profiling* da Versão 1 usando a ferramenta nvprof



Isso acontece porque na Versão 1 a montagem da matriz não é realizada totalmente em GPU, ela é copiada para a CPU, montada sequencialmente, convertida para o formato CSR usando a função da biblioteca MAGMA e só então transferida de volta para a GPU onde se obtém a solução do sistema de equações montado. Essas cópias podem ser evitadas com a implementação de *kernels* para realizar a montagem do sistema de equações diretamente na GPU e este foi o foco desta segunda iteração do processo APOD.

O processo de montagem da matriz no formato CSR foi dividido em algumas etapas para as quais foram desenvolvidos *kernels* CUDA. A lista das etapas em sequência

que devem ser realizadas para se obter a representação da matriz em formato CSR é apresentada a seguir:

1. Montagem da matriz dos elementos.
2. Montagem da matriz global:
    a) Contribuição das matrizes dos elementos;
    b) Contribuição `dRvtdLamda` e `dRpowerdVT`;
    c) Condições de contorno.
3. Contagem dos termos NNZ.
4. Conversão da matriz para o formato CSR.

A partir destas etapas definidas, foram implementados *kernels* para realizar cada uma delas, a primeira etapa já tinha sido implementada na Versão 1, então a primeira tarefa a se fazer após foi escrever as contribuições dos elementos para a matriz global que se encontram na matriz `Kelem` do vetor de estrutura `element_descriptor`. Isso foi realizado num primeiro momento montando a matriz no formato 2D, sendo cada uma das *threads* criadas responsável pela montagem de uma linha da matriz, em futuras otimizações deseja-se evitar a montagem da matriz no formato 2D e usar diretamente o formato COO para economizar memória em GPU. Neste *kernel* ao mesmo passo em que os elementos eram montados na matriz global, as contribuições de cada elemento para os vetores `dRvtdLamda` e `dRpowerdVT` eram adicionados, também sendo uma *thread* responsável por uma posição distinta destes vetores. Essa abordagem de destinar uma linha a cada *thread* evita as condições de corrida que podem acontecer pela contribuição de elementos vizinhos em uma mesma posição.

Após a execução deste primeiro *kernel* os vetores `dRvtdLamda` e `dRpowerdVT` foram montados na matriz global usando um segundo *kernel* onde cada *thread* acessa um índice desses vetores e monta na respectiva posição. Por fim as condições de contorno foram adicionadas, aqui foi usado um vetor com tamanho igual ao número de nós da malha e que armazen as condições que devem ser verificadas para controlar onde as condições de contorno devem incorporadas na matriz global. Para isso, os elementos deste vetor também foram divididas em um para cada *thread*.

Terminadas estas operações, a matriz encontra-se totalmente montada em GPU, no entanto ainda não está no formato requerido pela biblioteca MAGMA. Realizou-se uma etapa de conversão para obter a representação da matriz no formato CSR. Para isto, os termos NNZ da matriz foram contados para cada linha num outro *kernel*, permitindo criar o vetor de linhas da matriz já no formato CSR. Com estas informações prontas, novamente um *kernel* divide as linhas da matriz entre as *threads* que identificam a coluna

e então adicionam o valor respectivo a ela, convertendo a matriz para o formato CSR que então foi passado para a biblioteca MAGMA. Após todas estas configurações, antes de investir tempo em otimizações os resultados foram novamente testados, comparando-os com os resultados gerados pela versão original. A verificação dos resultados nesta segunda versão foi realizada da mesma forma que na Versão 1 e também não identificou erros nos resultados calculados.

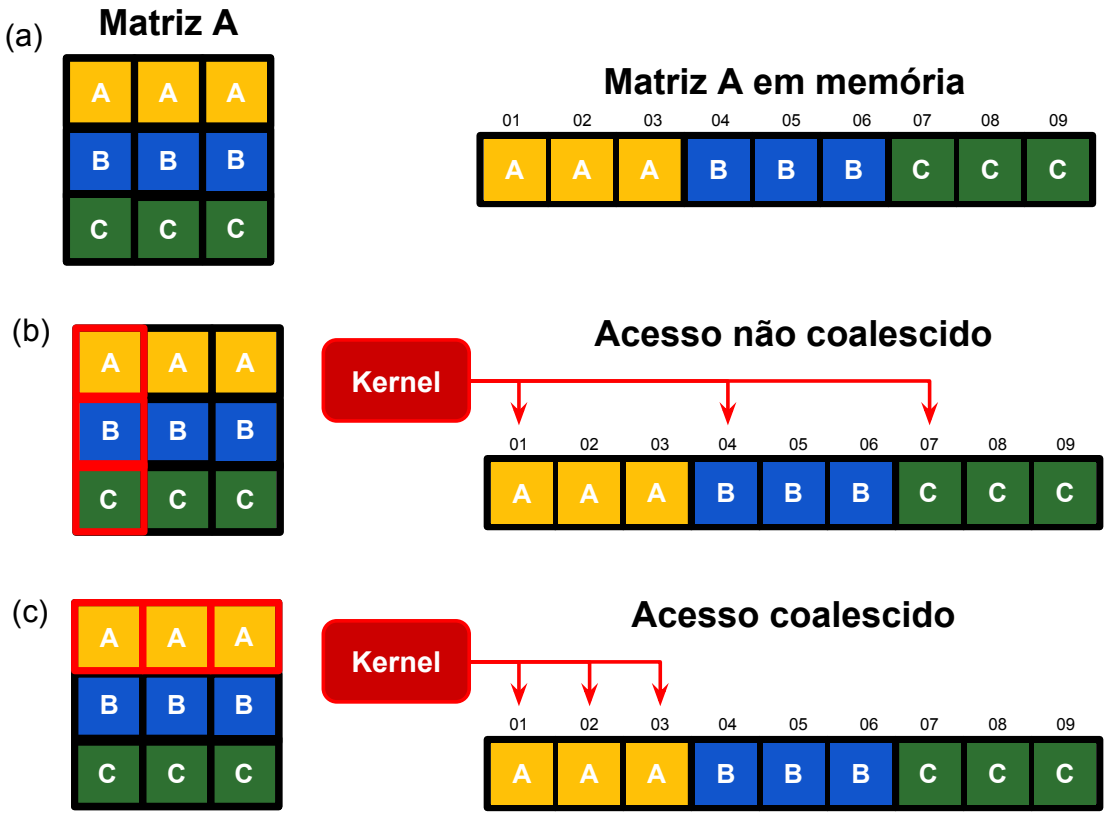
Entrando na etapa de otimização, ao analisar os *kernels* existentes, percebeu-se que o *kernel* para a montagem das matrizes dos elementos não necessitava passar por uma etapa de otimização, pois o seu tempo de execução era pequeno comparado aos demais, o tempo médio para a montagem das matrizes dos elementos no Caso 1 foi de 2,5ms e de 3,9ms para o Caso 2. Estes valores representam apenas uma pequena parcela da etapa de montagem, que em sua totalidade levava 92ms para o Caso 1 e 733ms para o Caso 2. Deste total, a maioria do tempo gasto foi distribuído entre os *kernels* de montagem da matriz, contagem dos termos NNZ e de conversão para o formato CSR, sendo a soma dos tempos 85ms no Caso 1 e 615ms no Caso 2. Com estas informações a otimização será focada nestes três *kernels*.

Um dos problemas identificados dentre estes *kernels*, foi na operação de contagem dos elementos NNZ. O que acontece neste *kernel* é que cada *thread* que está executando é responsável por uma linha inteira da matriz, que atualmente é montada no formato 2D, sendo assim, em um *warp* temos 32 *threads* acessando $32 \times N$ posições de memória diferentes para uma matriz de tamanho N. O problema aqui não é só o fato da quantidade de acessos necessárias, mas também o padrão destes acessos. O ideal é que os acessos à memória por um *warp* sejam próximos, a fim de minimizar o número de transações de memória necessárias, porém não é o que acontece neste *kernel*. O acesso de cada *thread* é referente a uma linha da matriz, onde entre duas linhas existe uma distância em memória, caracterizando assim um acesso não coalescido. Para resolvê-lo, neste caso, temos duas opções: realizar o acesso organizado por colunas ou fazer os blocos de *threads* serem responsáveis por uma linha e ordenar o acesso de suas *threads* ao longo da linha. O problema de acessos não coalescidos é melhor descrito na Figura 11, a representação de uma matriz em memória e apresentada na Figura 11(a), um exemplo de um bloco de *kernel* com acessos não coalescidos é descrito na Figura 11(b) e um bloco de um *kernel* com acesso coalescido a memória é ilustrado na Figura 11(c).

A solução adotada foi deixar cada bloco responsável por uma linha e usar a operação *atomicAdd* da API CUDA entre as *threads* deste bloco para contar os elementos não nulos de uma linha, salvando este valor em um vetor `thrust` para depois fazer uma redução e obter o valor total de NNZ. Com isto foi possível reduzir o tempo para este *kernel* nos Casos 1 e 2 de 11,2ms e 22,4ms para 2,1ms e 12,4ms.

Após a otimização do *kernel* para a contagem de termos NNZ, notou-se que o *kernel* para a conversão da matriz para o formato CSR também possuía o padrão de

Figura 11 – Descrição dos padrões de acesso à memória



acesso a memória não coalescido. Porém não podemos paralelizar a conversão em uma linha, pois os elementos devem seguir a ordem em que se encontram na matriz e caso isso seja realizado em paralelo entre posições de uma mesma linha, não temos como saber qual a ordem dos dados que farão parte dos vetores de colunas e valores do formato CSR.

Por fim, percebemos que a montagem da matriz também apresentava o mesmo padrão de acesso. Desta forma, aplicamos a mesma estratégia usada na contagem dos termos NNZ aqui: um bloco é responsável por uma linha da matriz e as *threads* deste bloco percorrem paralelamente a linha. No entanto, neste *kernel* temos que nos preocupar com as condições de corridas geradas por elementos que contribuem para uma mesma coluna, isto é um problema pois a GPU Quadro M5000 que está sendo utilizada, não suporta a operação `atomicAdd` para valores de precisão dupla, apenas para GPUs da NVIDIA com *compute capability* maior que 6.0 (NVIDIA, 2018c), a GPU usada possui 5.2. Porém, é possível implementar a nossa própria operação atômica, mas isto pode degradar a performance e não terá o mesmo desempenho que a função `atomicAdd` suportada pelas GPUs mais recentes (NVIDIA, 2018c). Usando a versão implementada da função aplicada no momento da adição dos valores na matriz global pode-se perceber um aumento no tempo de execução para o *kernel* de montagem, logo, voltou-se a utilizar a versão anterior

dele, finalizando a etapa de otimizações.

Com o fim do processo das otimizações, os resultados obtidos foram novamente testados e nenhuma discrepância foi encontrada. Após os testes, foram realizadas 30 execuções para o Caso 1 e 15 para o Caso 2 para obter os tempos médios de execução da Versão 2 do programa RAFEM, que foram de 6 minutos e 42 segundos e 116 minutos e 36 segundos, melhorando o desempenho em 12,4 vezes para o Caso 1 e 10 vezes para o Caso 2 quando comparados com a versão original.

6.3 Versão 3: Migração do laço principal para GPU

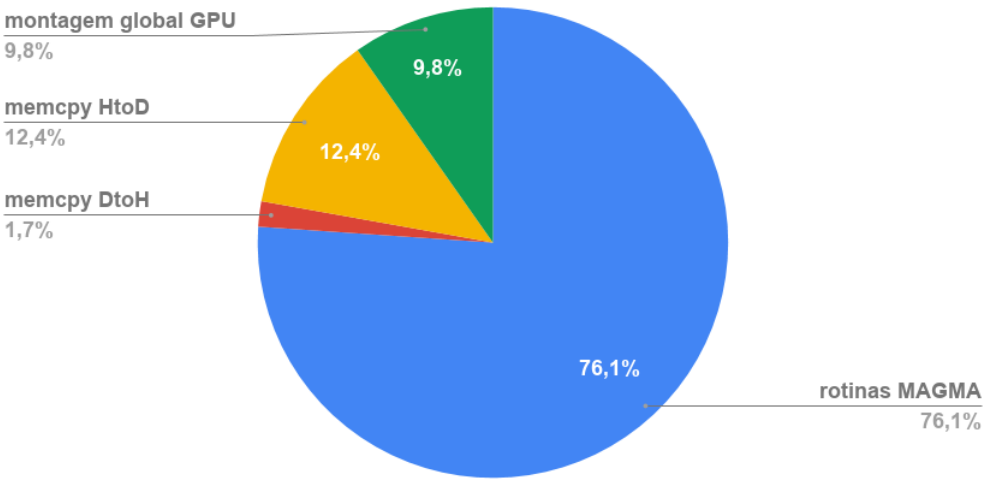
Como pudemos ver na Figura 6 a parte mais custosa da aplicação encontra-se dentro de dois laços, a chamada dos *kernels* desenvolvidos se encontram dentro deste laço e, no fim da solução do sistema de equações lineares existe a cópia do vetor de solução para CPU onde é realizada uma análise de convergência e são ajustados os passos de tempo da simulação, dentre outras atualizações de dados. O objetivo deste ciclo é passar estas etapas todas para a GPU, minimizando ainda mais as transferências que são realizadas entre CPU e GPU.

No decorrer deste ciclo também foram organizadas as declarações das variáveis utilizadas em GPU nos *kernels* desenvolvidos e pela biblioteca MAGMA, para que sejam declaradas uma única vez, pois até então estavam sendo declaradas e liberadas a cada chamada da função que fica dentro dos laços. Isso impacta no gráfico que podemos ver na Figura 12, onde embora a porção de tempo de transferências no sentido GPU para CPU (memcpy HtoD) tenham diminuído consideravelmente quando comparado com os resultados apresentados na Figura 10, ainda existe uma boa parcela de tempo envolvendo transferências no sentido contrário. Logo iniciou-se um estudo acerca das variáveis envolvidas neste laço principal e quais delas poderiam ser declaradas em GPU, para evitar cópias desnecessárias.

Com isto, alguns vetores e matrizes antes declarados e utilizados em CPU foram passados como parâmetros para a função que coordena a execução em GPU para que fossem copiados uma única vez e depois apenas atualizados em GPU. O laço que controla as iterações também foi movido para dentro desta função, para que pudéssemos coordenar as operações de atualização das variáveis em GPU.

Nesta etapa fez-se o uso da biblioteca `thrust` para declarar os vetores em GPU e aproveitar as funções que a biblioteca oferece, visto que as operações de atualização dos vetores que eram realizadas sequencialmente envolviam operações do tipo somar as posições de dois vetores ou aplicar uma fórmula sobre cada posição de um vetor. Tais operações podem ser resolvidas facilmente em GPU combinando a rotina `thrust::transform` e o `namespace thrust::placeholders` ou pelo uso do conceito de `Functors` em `C++`. Estas estratégias juntas permitem a criação de operações de transformação personalizadas aplicando-as paralelamente nos vetores em GPU. Para servir de exemplo, destacamos uma

Figura 12 – *Profiling* da Versão 2 usando a ferramenta nvprof



linha do código original dentro do laço principal onde esta estratégia pôde ser aplicada. As diferentes formas de realizar as operações são apresentadas na Figura 13.

Figura 13 – Exemplo do uso da rotina *transform*

```
1  // Original
2  for (i = 1; i < TotalDOFplus1; i++)
3      dVTdt2[i] = (Vstep[nstep][i] - Vstep[nstep-1][i]) / last_dt;
4  cudamemcpy(dVTdt2);
5
6  // 1. thrust transform placeholders (_1, _2)
7  using namespace thrust::placeholders;
8  thrust::transform(Vstep[nstep], Vstep[nstep-1], dVTdt2,(_1-_2)/last_dt);
9
10 // 2. Functor
11 struct my_operation {
12     double last_dt;
13
14     my_operation(double l) : last_dt(l) {}
15
16     double operator() (double a, double b) {
17         return (a - b) / last_dt;
18     }
19 };
20
21 // 2. thrust transform Functor
22 transform(Vstep[nstep], Vstep[nstep-1], dVTdt2, my_operation(last_dt));
```

Neste ciclo que envolveu o desenvolvimento da Versão 3 da aplicação, os passos aqui realizados podem ser considerados tanto da etapa de paralelização quando de otimização. Antes de finalizar o ciclo de desenvolvimento que deu origem a Versão 3, os resultados numéricos gerados pela aplicação foram comparados assim como para as demais versões geradas e mantiveram-se consistentes. Para definir o tempo médio de execução da Versão

3, foram realizadas mais 30 execuções para o Caso 1 e 15 para o Caso 2, resultando no tempo médio de execução de 4 minutos e 33 segundos para o Caso 1 e 77 minutos e 49 segundos para o Caso 2. Com isto obteve-se uma redução no tempo total de execução de até 18,2 vezes no Caso 1 e 15,8 vezes no Caso 2. Os resultados gerados pelas três versões são recapitulados e discutidos na próxima seção.

6.4 Análise dos Resultados

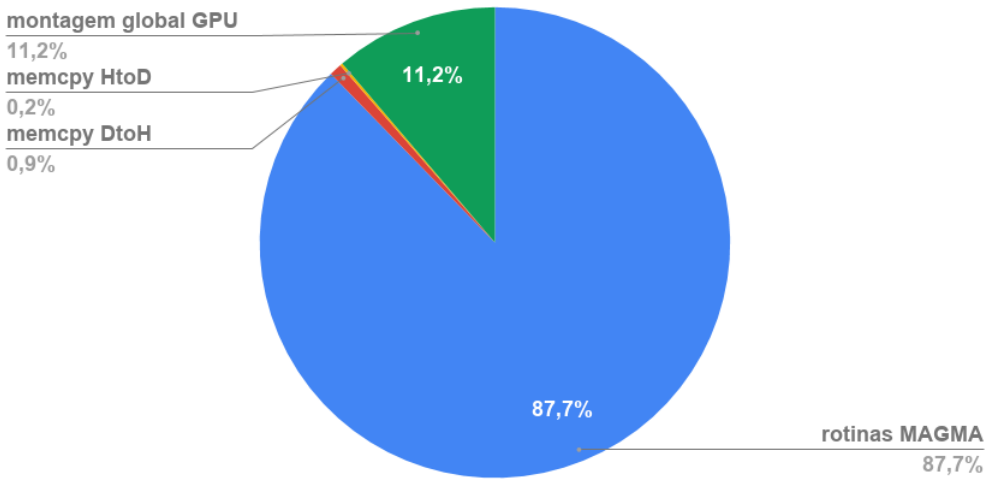
Após os três ciclos de desenvolvimento completos, foi possível observar uma boa redução no tempo total de execução do programa RAFEM, gerando 3 versões diferentes de código paralelo. Um grande problema nas primeiras duas versões eram as transferências de memória e o custo que elas representavam no tempo de execução em GPU. Para resolver isto o problema foi abordado aos poucos. Na Versão 1 tinha-se o problema de montar parte da matriz global em GPU e o restante era feito em CPU, criando assim uma transferência a cada iteração da aplicação. Este primeiro problema foi solucionado na Versão 2 do código através do desenvolvimento de *kernels* para montar a matriz global totalmente em GPU, com isto as transferências no sentido da GPU para CPU foram reduzidas a uma fração do custo que existia inicialmente, porém ainda existia um custo alto de transferência no sentido contrário.

As transferência da CPU para GPU foram tratadas na Versão 3, onde foi mencionado que o motivo deste custo existir era o fato de que as transferências estavam sendo realizadas dentro do laço principal do programa RAFEM, copiando dados para a GPU a cada iteração. Também existiam operações de atualização em variáveis que estavam localizadas na memória da CPU, o que demandava mais operações de transferência. A solução apresentada para este problema na Versão 3 foi mover o laço principal da aplicação para dentro da função criada para realizar a execução dos *kernels* desenvolvidos. Isso permitiu realizar as transferências antes do inicio do laço, enviando o que era necessário inicialmente para a GPU e posteriormente trabalhar com os dados que precisavam ser atualizados durante a execução do laço dentro da própria GPU, reduzindo ainda mais o tempo de execução gasto com transferências.

O resultado destas modificações podem ser observados na Figura 14, que apresenta os resultados do *profiling* da Versão 3 representando a divisão do tempo total de execução em GPU. Aqui podemos observar que os esforços para a redução do tempo gasto com transferências de memória foi bem sucedido já que agora a porcentagem do tempo gasto com a comunicação entre CPU e GPU somada é de 1,1%, restando assim, duas porções maiores que envolvem a montagem da matriz global e a solução do sistema de equações que ela representa.

Este cenário é semelhante ao apresentado pelo *profiling* da versão original do código, que gastava 99,5% do tempo total de execução com as rotinas de montagem e solução do sistema de equações lineares. Diante deste cenário, uma boa opção para obtermos uma

Figura 14 – *Profiling* da Versão 3 usando a ferramenta nvprof



maior redução do tempo de execução é investigar e experimentar novos parâmetros para a biblioteca MAGMA, já que suas rotinas envolvem quase 90% do tempo de execução em GPU. Essa análise é realizada na próxima seção juntamente com alguns experimentos.

Os resultados numéricos obtidos ao longo de todas as três versões de código paralelo geradas foram considerados idênticos aos gerados pelo original após compararmos eles com o *script* que foi desenvolvido para processar o arquivo de saída que a aplicação gera. Vale lembrar que para realizar as comparações adotou-se um arredondamento de 4 casas decimais para os valores de tensão e temperatura. Quanto aos resultados de tempo de execução da aplicação, podemos perceber nas Figura 15 e Figura 16 que o avanço no processo APOD foi trazendo novas contribuições para a redução do tempo de execução. A Figura 15 representa a média do tempo total de 30 execuções das diferentes versões para o Caso 1 e a Figura 16 representa a média do tempo total de 15 execuções das as diferentes versões para o Caso 2.

Outra medida que podemos analisar a partir das médias dos tempos de execução coletados é o *speedup* entre as versões geradas e o código original. A noção de *speedup* foi proposta inicialmente por Amdahl (1967) e focava em tarefas paralelas, no entanto esta medida pode ser utilizada mais genericamente para comparar a *performance* entre duas soluções como um todo que resolvem o mesmo problema, neste caso o programa RAFEM e suas versões desenvolvidas. Para calcular este valor de *speedup* podemos usar a fórmula:

$$Speedup = \frac{Tempo_{anterior}}{Tempo_{novo}} \tag{6.1}$$

Aplicando esta fórmula foi possível gerar o gráfico apresentado na Figura 17, onde podemos observar a evolução da aplicação em termos de *speedup*. Podemos notar também que os valores alcançados para o *speedup* são diferentes para uma mesma versão usando

Figura 15 – Média dos tempos de execução para o Caso 1

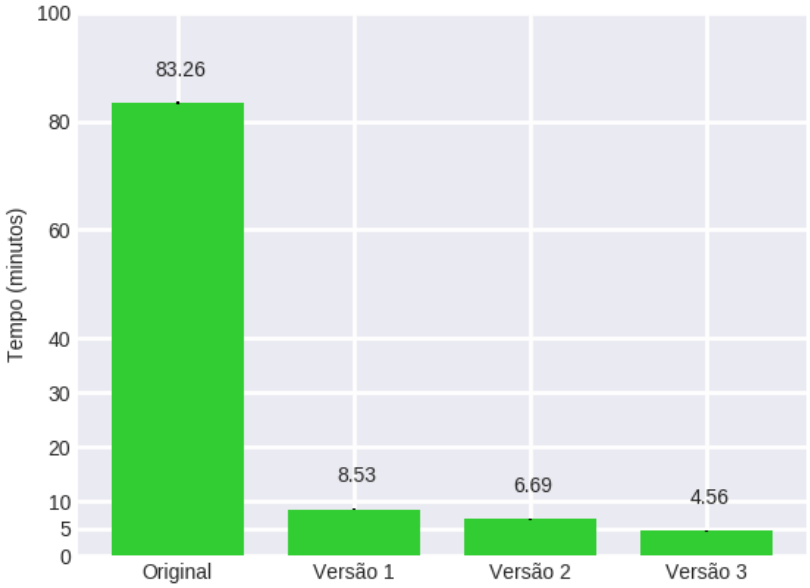
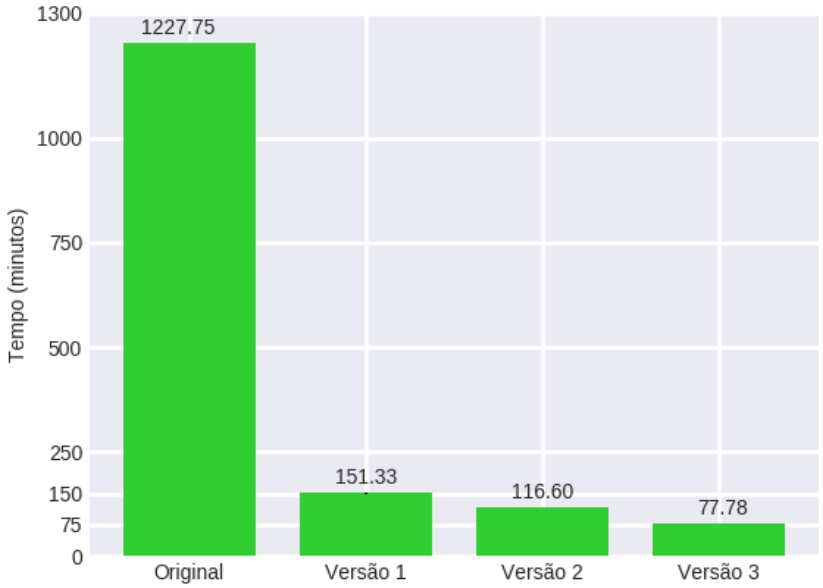
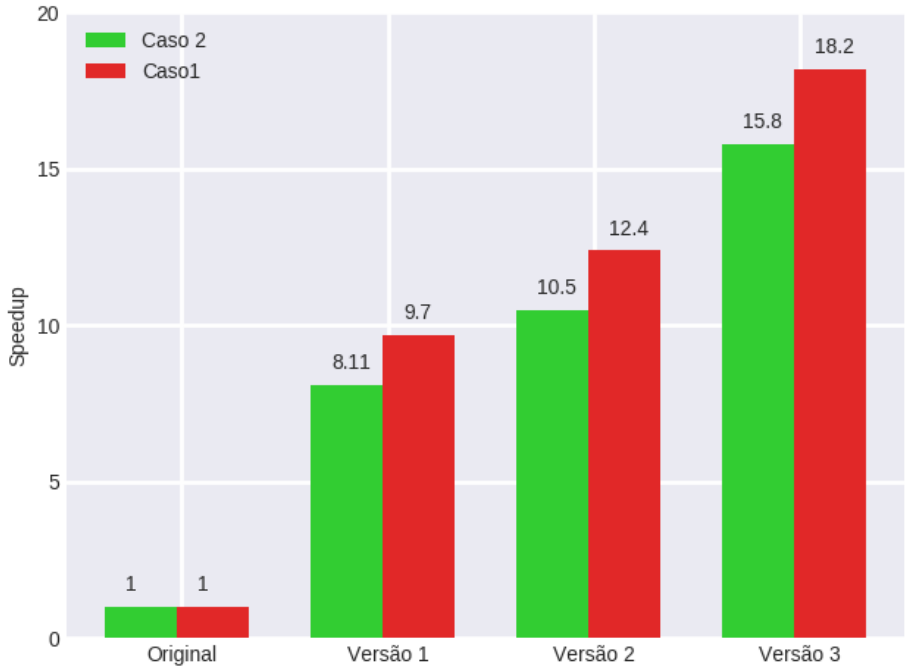


Figura 16 – Média dos tempos de execução para o Caso 2



casos de teste diferentes. Isso se deve ao fato de que temos cargas de trabalho distintas entre os dois casos, principalmente quando se trata da etapa da solução, que envolve um certo número de iterações que depende dos parâmetros de configuração do método utilizado. Esse fato, juntamente com o resultado do *profiling* da Versão 3, motivaram a análise mais profunda dos parâmetros utilizados no método numérico caracterizando um novo ciclo do processo APOD.

Figura 17 – Valores de *speedup* para os casos de teste 1 e 2



6.5 Versão 4: Otimizações Numéricas

Após a execução de três ciclos do processo APOD, podemos perceber no gráfico representado pela Figura 14 que a maioria do tempo de execução em GPU (87,7%) é dedicada as rotinas da biblioteca MAGMA que fornecem a solução do sistema de equações lineares. Como mencionado anteriormente, o custo de um método numérico iterativo como o GMRES(m) depende muito do número de iterações que ele realiza para obter a solução com a precisão desejada. Sendo assim, uma forma de reduzir o tempo das rotinas da biblioteca MAGMA é diminuir o número de iterações executadas para obter a solução. Para tanto, podemos diminuir a precisão utilizada como critério de parada, representada por um parâmetro de tolerância além de ajustar o parâmetro m do método GMRES(m).

Assim, esta seção trata de explorar e avaliar o impacto de diferentes configurações de parâmetros para o método de solução utilizado, considerando o seu desempenho e os resultados finais gerados pela aplicação. A primeira coisa a se fazer é verificar a convergência do método em termos do seu resíduo e o número de iterações. Inicialmente usou-se um valor de tolerância de 10^{-10} e o parâmetro m foi configurado como 80 para o Caso 1 e 150 para o Caso 2, estes valores foram escolhidos com base em experimentos iniciais não tão abrangentes quanto os apresentados nesta seção.

A fim de observar o comportamento do método GMRES(m) nos diferentes casos de teste, testamos valores para o parâmetro m num intervalo de 20 a 140 em passos de tamanho 20 para o Caso 1 e num intervalo de 50 a 300 em passos de tamanho 50 para

o Caso 2. Com os resultados deste experimento podemos ver qual número de iterações necessárias para satisfazer a condição de tolerância com diferentes configurações de m escolhidas. Estes valores são apresentados na Figura 18 para o Caso 1 e na Figura 19 para o Caso 2.

Figura 18 – Valores do resíduo para o método GMRES(m) no Caso 1

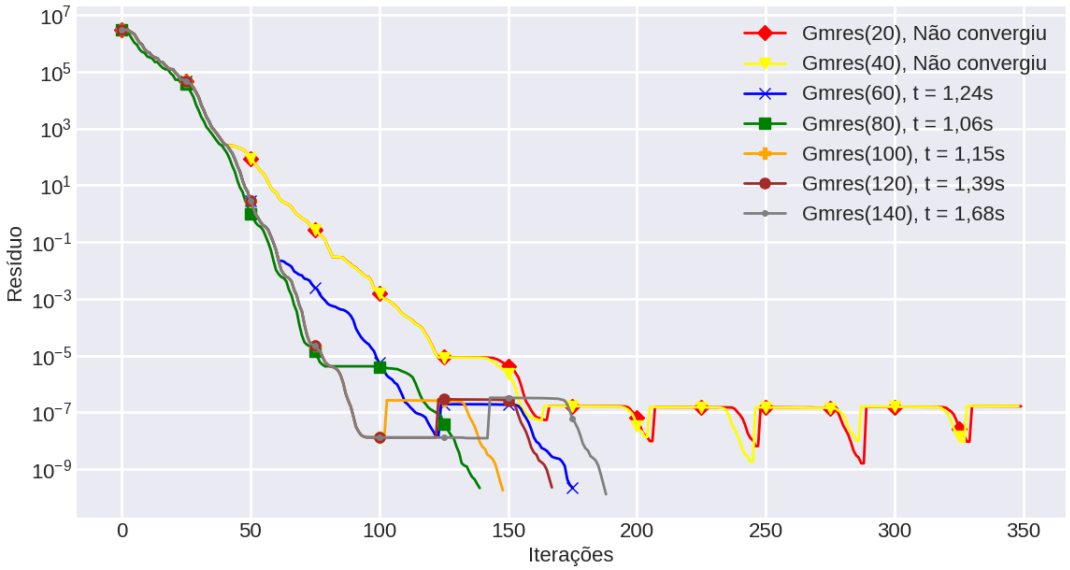
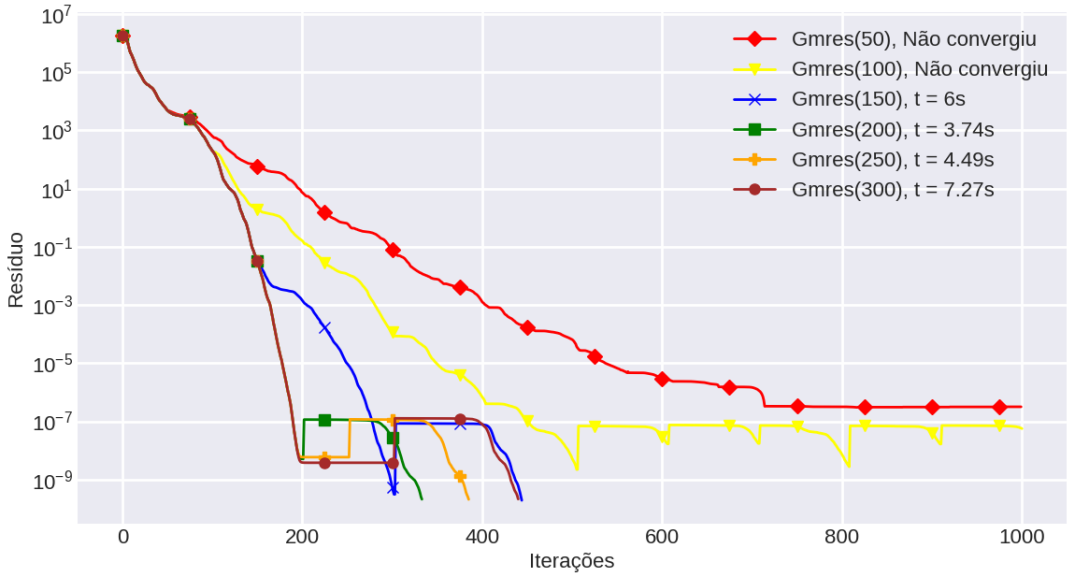


Figura 19 – Valores do resíduo para o método GMRES(m) no Caso 2



Isso permitiu observarmos o impacto do parâmetro m para o número de iterações do método e por consequência no tempo de execução da solução. Pode-se perceber também que o valor de 80 para m no Caso 1 foi uma boa escolha, pois é a configuração que

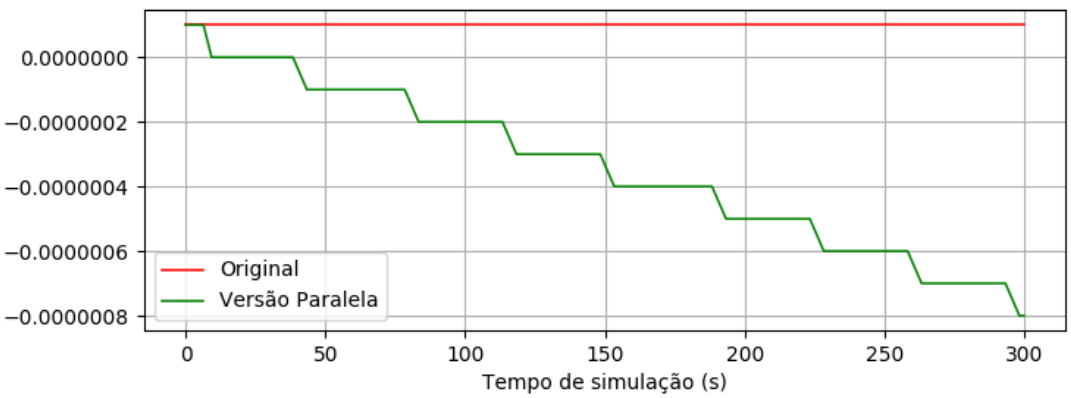
apresenta o menor número de iterações para a convergência e o menor tempo de execução. Já para o Caso 2, onde o valor escolhido para m foi de 150, não tivemos a melhor configuração, pois os valores 200 e 250 chegam na precisão configurada em um menor número de iterações e em menor tempo.

A elaboração destes gráficos também permitiu enxergar uma oportunidade de reduzir o tempo total de execução. Como podemos perceber nas Figuras 18 e 19 o resíduo apresenta uma grande redução nas primeiras 100 iterações para o Caso 1 e nas primeiras 200 para o Caso 2, em ambos os casos para algumas configurações de m o valor do resíduo é menor que 10^{-5} nestes intervalos de iterações. Sabendo que os resultados numéricos gerados com a tolerância inicial de 10^{-10} foram equivalentes aos da versão original do programa, podemos fazer alguns questionamentos como: Qual é a precisão mínima para que a aplicação continue gerando resultados utilizáveis? O uso de uma mesma precisão funciona para qualquer caso de teste? Se ao reduzirmos a tolerância e os resultados continuarem os mesmos, podemos realizar os cálculos utilizando precisão simples?

Partindo destes questionamentos, testou-se o impacto da redução da tolerância pela metade ficando o seu valor em 10^{-5}, que como vimos é um valor para o qual o método GMRES(m) chega em poucas iterações usando os mesmos valores de $m = 80$ para o Caso 1 e $m = 150$ para o Caso 2.

Para o Caso 1 com a configuração descrita, não houveram discrepâncias dentro dos critérios de comparação estabelecidos no Capítulo 6. Existem diferenças apenas se olharmos em uma escala menor que 4 casas decimais. Observando os resultados deste caso minuciosamente como mostra a Figura 20, podemos perceber um declínio constante da tensão calculada em um dos nós onde isto ocorreu. Isso pode ser um fator problema para outros casos de teste que envolvam um tempo de simulação maior, já que a diferença calculada parece aumentar com o tempo, assim trabalhos futuros podem investigar se isso ocorre em outros cenários de teste e identificar se isto pode ser de fato um problema.

Figura 20 – Tensão ao longo do tempo de simulação no nó 1



Com esta abordagem foi possível reduzir o tempo da atual Versão 3 de 4 minutos

e 33 segundos para 3 minutos e 39 segundos sem alterar outras linhas de código a não ser as que configuram os parâmetros do método GMRES(m), representando um *speedup* de 22,78 vezes, porém algo mais pode ser testado no Caso 1. No gráfico da Figura 18 podemos observar que os valores de m de 100, 120 e 140 também atingem o valor de tolerância de 10^{-5} em poucas iterações. Assim podemos testar outro valor para m para um destes valores e verificar se ele influencia no tempo final da aplicação. O novo m escolhido para o Caso 1 foi 100, 30 execuções foram realizadas para se obter a média do tempo de execução que foi de 3 minutos e 1 segundo, aumentando o valor de *speedup* para 27,53 e mantendo os mesmos resultados numéricos. Isso aconteceu porque esta configuração do parâmetro m reduziu o número médio de iterações necessárias para obter as soluções ao longo da simulação e por consequência o tempo para obtê-las. Reduziu-se de 118 iterações em média levando cerca de 0,64 segundos para solucionar o sistema dada a tolerância de 10^{-5} com $m = 80$ para 88 iterações levando 0,52 segundos com $m = 100$, assim, evidenciando as possíveis vantagens que podemos ter ao usar um método de ajuste dinâmico para o GMRES(m).

Para os experimentos envolvendo o Caso 2, o parâmetro m foi configurado com o mesmo valor anterior de 150 só que agora com a tolerância de 10^{-5}. Os resultados numéricos mantiveram-se consistentes e foram realizadas 15 execuções do programa para obter o tempo médio de execução de 52 minutos e 42 segundos, atingindo um *speedup* de 23 vezes, representando um ótimo ganho de *performance* quando comparado com o *speedup* de 15,7 vezes da Versão 3.

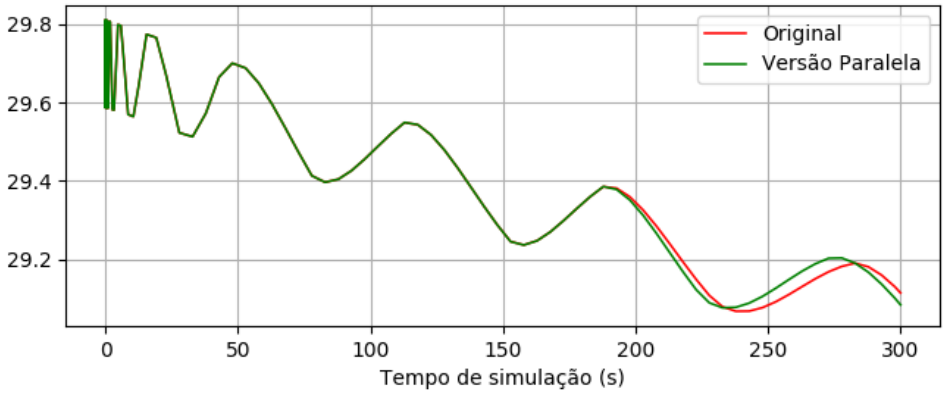
Ainda para o Caso 2, é possível usar um outro valor para m visto que os valores 200, 250 e 300 chegam em poucas iterações para o valor do resíduo de 10^{-5}. Assim, usamos o valor de $m = 250$ e a tolerância 10^{-5} para verificar o impacto nos resultados obtidos, onde para o tempo de execução foi possível observar uma redução para 45 minutos e 51 segundos, representando um *speedup* de 26,7 vezes. Novamente a explicação para esta redução no tempo total de execução entre os dois valores de m testados é a redução do número de iterações e do tempo para obter-se uma solução. Estes valores foram de 271 e 2,7 segundos para $m = 150$ e 186 e 1,97 segundos para $m = 250$.

Os resultados testados apontaram que para a temperatura a maior diferença encontrada foi de 0,0001 no nó 1954, porém diferenças para a tensão calculada ao longo do tempo foram mais nítidas como a Figura 21 mostra, chegando ao valor de 0,02 para o mesmo nó. Com isto reforça-se a importância de se ter uma boa configuração do conjunto de parâmetros envolvidos no método, não só em termos de tempo de execução mas também de precisão numérica.

6.6 Balanço do Capítulo

Neste capítulo foram descritas as iterações resultantes da aplicação do processo APOD, mostrando aspectos do desenvolvimento de cada versão e o impacto causado pelas

Figura 21 – Tensão ao longo do tempo de simulação no nó 1954



modificações propostas. Ao longo deste capítulo pôde-se perceber a evolução da solução proposta em termos de redução do tempo total de execução, que foi diminuindo a cada versão gerada.

Apresentou-se também uma seção para a análise dos resultados onde foram reunidos os valores de *speedup* obtidos em cada versão, e analisou-se o *profiling* da última versão gerada até então. Isto motivou a criação de uma quarta versão, onde realizou-se uma investigação envolvendo os parâmetros usados no método GMRES(m) e verificando qual o seu impacto para o tempo total de execução e para os resultados numéricos. Com isto percebeu-se a importância de configurar o método com um bom conjunto de parâmetros e destacou-se como uma possível solução para isto o uso de algum método de ajuste dinâmico para o GMRES(m).

7 CONSIDERAÇÕES FINAIS

O presente trabalho investigou a aplicação de simulação computacional RAFEM considerando estratégias para reduzir o seu tempo total de execução. Essas estratégias envolveram a adoção de um método de desenvolvimento de *software* focado na otimização de aplicações através do uso de GPU, tratando-se de um método de desenvolvimento iterativo, onde neste trabalho foram completados quatro ciclos visando alcançar uma melhoria no desempenho da aplicação preocupando-se também com os resultados numéricos gerados.

Durante o desenvolvimento seguindo as etapas do ciclo definido pelo método APOD, foram implementados *kernels* usando a API CUDA para realizar as tarefas paralelas do código executando-as em GPU, realizando-se otimizações para as transferências de memória entre a CPU e a GPU visando reduzi-las e adotou-se uma implementação paralela do método de solução iterativo GMRES(m) disponibilizada pela biblioteca MAGMA. Também foram realizados experimentos usando o método GMRES(m) onde percebeu-se o impacto que o uso de um determinado conjunto de parâmetros possui afetando o tempo de execução e a solução numérica obtida. Com isto, para os casos de teste apresentados obteve-se uma redução do tempo de execução de até 27 vezes para o Caso 1 e de 23 vezes para o Caso 2, isto sem alterar a qualidade dos resultados numéricos gerados pela aplicação RAFEM.

Por fim, a análise da aplicação e o processo de desenvolvimento expuseram diversas possíveis soluções diferentes das que foram adotadas. Assim, identificamos algumas oportunidades para trabalhos futuros. Estas podem envolver uma análise numérica com mais casos de teste, a comparação com diferentes métodos de solução e diferentes bibliotecas, a execução das versões em diferentes ambientes computacionais e uma verificação de como a aplicação se comportaria utilizando apenas variáveis de precisão simples para montar e resolver o sistema de equações, visto que alguns testes onde se reduziu o valor de tolerância pela metade ainda geraram os mesmos resultados. Caso isto seja possível, podemos obter um melhor desempenho, pois as operações computacionais envolvendo precisão simples são menos custosas do que as realizadas em precisão dupla.

Dentre diferentes métodos a serem utilizados, destacamos que seria muito interessante para o trabalho a adoção de uma implementação do método GMRES(m) dinâmico, visto que os seus parâmetros influenciam muito no seu custo computacional e assim no tempo total de execução da aplicação e que estes parâmetros se mostraram diferentes para os casos que foram testados.

REFERÊNCIAS

AHMAD, M. O.; MARKKULA, J.; OIVO, M. Kanban in software development: A systematic literature review. In: IEEE. **Software Engineering and Advanced Applications (SEAA), 2013 39th EUROMICRO Conference on**. [S.l.], 2013. p. 9–16. Citado na página 52.

AMDAHL, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In: ACM. **Proceedings of the April 18-20, 1967, spring joint computer conference**. [S.l.], 1967. p. 483–485. Citado na página 67.

AMESTOY, P. R. et al. A fully asynchronous multifrontal solver using distributed dynamic scheduling. **SIAM Journal on Matrix Analysis and Applications**, SIAM, v. 23, n. 1, p. 15–41, 2001. Citado 2 vezes nas páginas 30 e 43.

BAKER, A. H.; JESSUP, E. R.; KOLEV, T. V. A simple strategy for varying the restart parameter in GMRES (m). **Journal of computational and applied mathematics**, Elsevier, v. 230, n. 2, p. 751–761, 2009. Citado na página 32.

CAMARDA, K. V.; STADTHERR, M. A. Frontal solvers for process engineering: local row ordering strategies. **Computers & chemical engineering**, Elsevier, v. 22, n. 3, p. 333–341, 1998. Citado na página 17.

CHAPRA, S. C.; CANALE, R. P. **Métodos numéricos para engenharia**. Massachusetts, EUA: McGraw-Hill, 2008. Citado na página 22.

DUFF, I. S.; REID, J. K. The multifrontal solution of indefinite sparse symmetric linear. **ACM Transactions on Mathematical Software (TOMS)**, ACM, v. 9, n. 3, p. 302–325, 1983. Citado 2 vezes nas páginas 30 e 43.

DZIEKONSKI, A.; LAMECKI, A.; MROZOWSKI, M. GPU-accelerated finite element method. In: **2016 IEEE MTT-S International Conference on Numerical Electromagnetic and Multiphysics Modeling and Optimization (NEMO)**. Beijing, China: IEEE, 2016. p. 1–2. Citado 2 vezes nas páginas 46 e 47.

DZIEKONSKI, A. et al. Finite element matrix generation on a GPU. **Progress In Electromagnetics Research**, EMW Publishing, v. 128, p. 249–265, 2012. Citado 3 vezes nas páginas 45, 46 e 47.

EMBREE, M. The tortoise and the hare restart GMRES. **SIAM review**, SIAM, v. 45, n. 2, p. 259–266, 2003. Citado na página 32.

FILIPOVIC, J.; PETERLIK, I.; FOUSEK, J. GPU Acceleration of Equations Assembly in Finite Elements Method - Preliminary Results. In: **SAAHPC: Symposium on Application Accelerators in HPC**. Urbana, Illinois: Conference Publishing Services (CPS), IEEE, 2009. Citado 2 vezes nas páginas 44 e 47.

GEORGESCU, S.; CHOW, P.; OKUDA, H. GPU acceleration for FEM-based structural analysis. **Archives of Computational Methods in Engineering**, Springer, v. 20, n. 2, p. 111–121, 2013. Citado na página 44.

GRAHAM, S. L.; KESSLER, P. B.; MCKUSICK, M. K. Gprof: A Call Graph Execution Profiler. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 17, n. 6, p. 120–126, jun. 1982. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/872726.806987>. Citado na página 41.

GUSTAFSON, J. L. Reevaluating Amdahl's law. **Communications of the ACM**, ACM, v. 31, n. 5, p. 532–533, 1988. Citado na página 50.

HENNESSY, J. L.; PATTERSON, D. A. **Computer architecture: a quantitative approach**. San Francisco, CA, USA: Elsevier, 2011. Citado na página 21.

HERNÁNDEZ, M. et al. Accelerating fibre orientation estimation from diffusion weighted magnetic resonance imaging using GPUs. **PloS one**, Public Library of Science, v. 8, n. 4, p. e61892, 2013. Citado na página 35.

HINTON, L.; OWEN, D. **Finite Element Programming**. University of California, Berkeley, USA: Elsevier, 1980. Citado na página 21.

IRONS, B. M. A frontal solution program for finite element analysis. **International Journal for Numerical Methods in Engineering**, Wiley Online Library, v. 2, n. 1, p. 5–32, 1970. Citado 3 vezes nas páginas 18, 30 e 41.

JAMAL, A. **A parallel iterative solver for large sparse linear systems enhanced with randomization and GPU accelerator, and its resilience to soft errors**. Tese (Doutorado) — Université Paris-Saclay, 2017. Citado 2 vezes nas páginas 27 e 37.

JIANG, Y. et al. Formulation of 3D finite elements for hepatic radiofrequency ablation. **International Journal of Modelling, Identification and Control**, Inderscience Publishers, v. 9, n. 3, p. 225–235, 2010. Citado 4 vezes nas páginas 17, 18, 39 e 59.

JOUBERT, W. On the convergence behavior of the restarted GMRES algorithm for solving nonsymmetric linear systems. **Numerical linear algebra with applications**, Wiley Online Library, v. 1, n. 5, p. 427–447, 1994. Citado na página 32.

KAPELINSKI, K. **Aplicação de Processamento Paralelo no Programa RAFEM utilizando OpenMP**. Dissertação (Mestrado) — Universidade Federal do Pampa (UNIPAMPA), Alegrete, RS, Brazil, June 2016. Disponível em: <http://dspace.unipampa.edu.br:8080/jspui/handle/riu/1613>. Citado 4 vezes nas páginas 18, 40, 43 e 47.

KIM, S. J. et al. Direct numerical simulation of composite structures. **Journal of composite materials**, Sage Publications Sage CA: Thousand Oaks, CA, v. 36, n. 24, p. 2765–2785, 2002. Citado na página 30.

KIRK, D. B.; WEN-MEI, W. H. **Programming massively parallel processors: a hands-on approach**. [S.l.]: Morgan kaufmann, 2016. Citado 2 vezes nas páginas 32 e 33.

LEON, S. J. **Álgebra Linear com Aplicações**. [S.l.]: Grupo Gen-LTC, 2000. Citado 2 vezes nas páginas 23 e 25.

MILETTO, M. C.; SCHEPKE, C. Uso do Método Multi Frontal para Acelerar uma Aplicação de Ablação por Radiofrequência. **Anais do WSCAD-WIC 2017**, 2017. Citado 2 vezes nas páginas 43 e 47.

MOORE, G. E. **Cramming more components onto integrated circuits, Electronics Magazine**. 1965. Citado na página 32.

NVIDIA. **CUDA C Best Practices Guide**. 2018. <https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html>. Accessado em: 2018-06-09. Citado 2 vezes nas páginas 49 e 51.

NVIDIA. **CUDA C Programming Guide**. 2018. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>. Accessado em: 2018-05-28. Citado 2 vezes nas páginas 33 e 34.

NVIDIA. **CUDA C Programming Guide, Atomic Operations.** 2018. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomicadd>. Accessado em: 2018-05-28. Citado na página 63.

OPENMP. **Home - OpenMP**. 2018. <http://www.openmp.org/>. Accessado em: 2018-05-08. Citado na página 18.

POSSEBON, R. B. **Investigação de Relações entre Temperatura e Impedância Elétrica e Permeabilidade de Biomaterial (Fígado) Usadas para Ablação de Tumor por Radiofrequência**. Dissertação (Mestrado) — Mestrado em ENGENHARIA - UNIPAMPA, 2016. Citado na página 18.

RAEDER, M. **Um processo de geração automática de código paralelo para arquiteturas híbridas com afinidade de memória**. Tese (Doutorado) — Programa de Pós-Graduação em Ciência da Computação, 2014. Faculdade de Informática. Disponível em: <http://tede2.pucrs.br/tede2/handle/tede/7390>. Citado na página 52.

RAO, S. S. **The finite element method in engineering**. [S.l.]: Butterworth-heinemann, 2017. Citado na página 21.

SAAD, Y. **Iterative Methods for Sparse Linear Systems**. [S.l.]: siam, 2003. v. 82. Citado na página 31.

SAAD, Y.; SCHULTZ, M. H. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. **SIAM Journal on scientific and statistical computing**, SIAM, v. 7, n. 3, p. 856–869, 1986. Citado na página 31.

SCOTT, J. A. Parallel frontal solvers for large sparse linear systems. **ACM Transactions on Mathematical Software (TOMS)**, ACM, v. 29, n. 4, p. 395–417, 2003. Citado na página 30.

SCOTT, J. A. A frontal solver for the 21st century. **International Journal for Numerical Methods in Biomedical Engineering**, Wiley Online Library, v. 22, n. 10, p. 1015–1029, 2006. Citado na página 30.

SOUZA, R. M. de. **O método dos elementos finitos aplicado ao problema de condução de calor**. 2003. Citado 2 vezes nas páginas 21 e 22.

SUTTER, H.; LARUS, J. Software and the Concurrency Revolution. **Queue**, ACM, New York, NY, USA, v. 3, n. 7, p. 54–62, set. 2005. ISSN 1542-7730. Disponível em: <http://doi.acm.org/10.1145/1095408.1095421>. Citado na página 32.

TOMOV, S.; DONGARRA, J.; BABOULIN, M. Towards dense linear algebra for hybrid GPU accelerated manycore systems. **Parallel Computing**, v. 36, n. 5-6, p. 232–240, jun. 2010. ISSN 0167-8191. Citado na página 51.

TU, J.; YEOH, G. H.; LIU, C. **Computational fluid dynamics: a practical approach**. United Kingdom: Butterworth-Heinemann, 2018. Citado na página 27.

WHITEHEAD, N.; FIT-FLOREA, A. Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs. **rn (A+ B)**, v. 21, n. 1, p. 18749–19424, 2011. Citado na página 59.

ÍNDICE