Universidade Federal de Minas Gerais
Escola de Engenharia
Programa de Pós-Graduação em Engenharia Elétrica

Solução Ponta a Ponta em GPU do Método Sem Malha Local Petrov-Galerkin (MLPG)

Lucas Pantuza Amorim

Belo Horizonte, Dezembro de 2019

Lucas Pantuza Amorim

Solução Ponta a Ponta em GPU do Método Sem Malha Local Petrov-Galerkin (MLPG)

Tese submetida à Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Minas Gerais, como requisito para obtenção do título de Doutor em Engenharia Elétrica.

Orientador: Prof. Renato Cardoso Mesquita

Belo Horizonte, Dezembro de 2019

Amorim, Lucas Pantuza.
A524s Solução ponta a ponta em GPU do método sem malha local Petrov-Galerkin (MLPG) [recurso eletrônico] / Lucas Pantuza Amorim. - 2019.
1 recurso online (xvii,108 f. : il., color.) : pdf.

Orientador: Renato Cardoso Mesquita.

Tese (doutorado) - Universidade Federal de Minas Gerais, Escola de Engenharia.

Inclui bibliografia.
Exigências do sistema: Adobe Acrobat Reader.

1. Engenharia elétrica - Teses. 2. Eletromagnetismo - Teses. I. Mesquita, Renato Cardoso. II. Universidade Federal de Minas Gerais. Escola de Engenharia. III. Título.

CDU: 621.3(043)

Ficha catalográfica: Biblioteca Profº Mário Werneck, Escola de Engenharia da UFMG

"Solução Ponta a Ponta em Gpu do Método sem Malha Local Petrov-galerkin (mlpg)"

Lucas Pantuza Amorim

Tese de Doutorado submetida à Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em Engenharia Elétrica da Escola de Engenharia da Universidade Federal de Minas Gerais, como requisito para obtenção do grau de Doutor em Engenharia Elétrica.

Aprovada em 04 de dezembro de 2019.

Por:

Prof. Dr. Renato Cardoso Mesquita
DEE (UFMG) - Orientador

Prof. Dr. Ricardo Luiz da Silva Adriano
DEE (UFMG)

Prof. Dr. Renato Antônio Celso Ferreira
DCC (UFMG)

Prof. Dr. Marcio Matias Afonso
DEE (CEFET-MG)

Prof. Dr. Eduardo Henrique da Rocha Coppoli
DEE (CEFET-MG)

Prof. Dr. Alexandre Ramos Fonseca
Instituto de Ciência e Tecnologia (UFVJM)

Agradecimentos

Agradeço ao meu orientador, Professor Renato Mesquita, pela paciência infinita e por se tornar minha referência profissional ao longo destes anos de convívio. À minha família, por abrir mão da minha presença. E, enfim, aos colegas do Laboratório de Otimização e Projeto Assistido por Computador — LOPAC (UFMG) e do Centro Federal de Educação Tecnológica de Minas Gerais — CEFET-MG, que se prontificaram toda vez que precisei de ajuda.

Resumo

Este trabalho apresenta um procedimento massivamente paralelo ponta a ponta para a solução de problemas de valor de contorno em unidades de processamento gráfico (*graphics processing units* — GPU). A proposta é uma estratégia integrada que envolve não apenas o cálculo das contribuições nodais e a montagem da matriz de rigidez usando o método *Meshless Local Petrov Galerkin* (MLPG), mas também a solução iterativa do sistema de equações algébricas com o uso dos métodos da família dos Gradientes Conjugados (CG).

A contribuição mais importante é o MLPG ponta a ponta em GPU implementado e tirando proveito máximo da arquitetura SIMT (*Single Instruction Multiple Thread*) do dispositivo. Desde os cálculos iniciais até a entrega da solução para o problema tratado, as etapas integradas são computadas exclusivamente na GPU para evitar custos relacionados à movimentação e conversão de dados. A solução proposta usa a natureza substancialmente paralela do MLPG mantendo cada nó da nuvem em uma *thread* no dispositivo, permanecendo desta forma até a finalização da computação da última etapa do processo com subsequente entrega dos valores de interesse nos respectivos nós da nuvem. Dessa maneira, são necessárias estruturas auxiliares mínimas e poucos pontos de sincronização.

Variações da solução são testadas para avaliar o impacto efetivo na GPU, suportando decisões mais precisas. Entre o que é testado estão o posicionamento inicial dos nós no domínio, as técnicas de funções de forma do MLPG e quatro *solvers* diferentes da família dos Gradientes Conjugados, sendo eles: (i) Gradientes BiConjugados — BICG, (ii) Gradientes Conjugados Quadrático — CGS, (iii) Gradientes BiConjugados Estabilizado — BICG-Stab e (iv) Gradientes BiConjugados Estabilizado Aprimorado — I-BICG-Stab.

Para avaliar a viabilidade da solução e as métricas de desempenho, é utilizado o problema do capacitor de duas placas paralelas. Apesar da simplicidade do problema, os algoritmos são os mesmos usados para problemas complexos, e foi observado um *speedup* de sete vezes na solução completa. Este valor pode ser ainda mais expressivo com o crescimento do número de nós da nuvem durante a discretização de domínio.

Palavras chaves: GPU, método sem malha, MLPG, solucionador.

Abstract

This work presents an end-to-end massively parallelized procedure for the solution of boundary value problems on Graphics Processing Units (GPU). The proposal is an integrated strategy that not only entails the calculation of nodal contributions, and the stiffness matrix assembly using the Meshless Local Petrov Galerkin Method (MLPG) but also the iterative solution of the system of algebraic equations in combination with methods from the Conjugate Gradient (CG) family.

The most important contribution is a complete end-to-end MLPG implementation in GPU taking full advantage of the device's Single Instruction Multiple Thread (SIMT) architecture. From initial calculations to solution delivery for the problem addressed, the integrated steps are computed exclusively in the GPU to avoid costs related to data movement and conversion. The proposed solution takes advantage of the parallel nature of the MLPG by holding each cloud node in a thread on the device, until the final computation of the last process step is completed with subsequent delivery of the values of interest to the respective cloud nodes. Thus minimal auxiliary structures and few synchronization points are required.

Different solution variations are tested to assess the effective impact on the GPU, supporting more accurate decisions. Among others, the tests include variations on the initial node position in the domain, the MLPG form function techniques and four different solvers of the Conjugate Gradient family, namely: (i) BiConjugate Gradients — BICG, (ii) Quadratic Conjugate Gradients — CGS, (iii) BiConjugate Gradients Stabilized — BICG-Stab and (iv) BiConjugate Gradients Enhanced — I-BICG-Stab.

To evaluate the solution viability and performance metrics, the two parallel plate capacitor problem is used. Despite the simplicity of the problem, the application of the proposed algorithms to more complex problems is straightforward, and a sevenfold speedup is observed in the end-to-end solution. This number can be even more significant with cloud node growth during domain discretization.

Keywords: GPU, meshless, MLPG, solver.

Lista de Figuras

1.1 Comparação entre formas de discretização de um domínio 2
1.2 Problemas com elementos estruturantes de uma malha triangular 3

2.1 Dados de tendências dos últimos 42 anos a respeito do microprocessador . . . 12
2.2 Arquitetura da CPU *versus* GPU 15
2.3 Evolução ao longo dos anos do desempenho teórico em precisão dupla 17
2.4 Evolução ao longo dos anos da quantidade de operações de ponto flutuante . . 18
2.5 Evolução ao longo dos anos do limite de largura de banda de memória 19
2.6 A hierarquia entre as estruturas CUDA e o modelo de acesso à memória . . . 19
2.7 Arquitetura heterogênea de um programa em CUDA 20
2.8 Execução aninhada de *grid* de *threads* em GPU 29

3.1 Representação do domínio e fronteiras do capacitor de duas placas 34
3.2 Representação de uma parte do domínio usada pelo MLPG 36
3.3 Determinação do domínio de suporte do MLS e RPIMp 39
3.4 Representação do domínio e região de vizinhança 40
3.5 Otimização do algoritmo Knn usando subdivisão de espaço 42
3.6 Montagem da matriz de rigidez através da avaliação da forma fraca local . . . 46

4.1 Distribuição do custo computacional das operações de álgebra linear nos *solvers* 55
4.2 Avaliação de impacto do uso de pré-condicionador 60
4.3 Comparativo dos *solvers* (vetores auxiliares e pontos de sincronização) 61
4.4 Representações gráficas do problema de minimização da função quadrática . . 63
4.5 Evolução da precisão no Método dos Gradientes 65

5.1 Comparação entre perturbações nas coordenadas dos nós da mesma nuvem . . 81
5.2 Evolução do número da condição ao inserir perturbação 82
5.3 Evolução do número de iterações dos *solvers* necessárias para convergência . . 82
5.4 Evolução do tempo da montagem da matriz na GPU 83

5.5 Evolução do número de condição variando-se o domínio de suporte 84
5.6 Evolução do número de condição variando-se o número de nós da nuvem . . . 85
5.7 Evolução do tempo da iteração dos *solvers* na GPU 86
5.8 Evolução do número de iterações para a convergência dos *solvers* 86
5.9 Evolução do tempo total para convergência dos *solvers* 87
5.10 Comparação de tempo do algoritmo knn na CPU *versus* GPU 88
5.11 Comparação de tempo da montagem da matriz na CPU *versus* GPU 89
5.12 Comparação de tempo dos *solvers* na CPU *versus* GPU 90
5.13 Comparação percentual de tempo total gasto na CPU *versus* GPU 91
5.14 Comparação de tempo total gasto na CPU *versus* GPU 92
5.15 Evolução percentual de tempo total gasto na GPU 93
5.16 Evolução de tempo total gasto na GPU 93

Lista de Tabelas

2.1 Glossário de termos relativos à arquitetura GPU 21

3.1 Métodos *Meshless Local Petrov-Galerkin* (MLPG) 45

Lista de Algoritmos

1 Método dos Gradientes 65

2 Método dos Gradientes Conjugados — CG 67

3 Método dos Gradientes BiConjugados Clássico — BICG 69

4 Método dos Gradientes Conjugados Quadrático — CGS 75

5 Método dos Gradientes BiConjugados Estabilizado — BICG-Stab 76

6 Método dos Gradientes BiConjugado Estabilizado Aprimorado — I-BICG-Stab . 77

Lista de Códigos Fonte

2.1 Exemplo de paralelelismo dinâmico 31
4.1 Método dos Gradientes BiConjugados clássico — BICG 70
4.2 Função que aloca memória para um vetor 71
4.3 Função que inicializa um vetor com zeros 72
4.4 Função que copia os dados de um vetor para outro 72
4.5 Função que libera a memória alocada para um vetor 72
4.6 Função de cálculo da norma 72
4.7 Função que sincroniza *threads* no dispositivo 72
4.8 Função que multiplica dois vetores 72
4.9 Função que soma dois vetores 73
4.10 Função que multiplica um vetor por uma matriz 73

Lista de Símbolos

α	Constante de valor elevado (método das penalidades)
$\alpha, \beta, \ldots, \omega$	Escalares
$\bar{t}$	Condição de fronteira de Neumann
$\bar{x}$	Vetor x temporário (auxiliar em um processo iterativo)
Γ_t	Fronteira de Neumann
Γ_u	Fronteira de Dirichlet
$\hat{x}$	Tipicamente, solução exata de $Ax = b$
$\lambda_{\max}(A)$	Autovalor de A (em módulo) máximo
$\lambda_{\min}(A)$	Autovalor de A (em módulo) mínimo
$cond(A)$	Número de condição da matriz
$diag(A)$	Diagonal da matriz
$\|z\|_2$	*2-norm* ou norma euclidiana
$\|z\|_\infty$	Norma linha
$\|z\|_1$	Norma coluna
Ω	Domínio do problema
Ω_q^i	Subdomínio do nó i
ϕ_i	Função de forma do nó i
F	Vetor $n \times 1$ do sistema linear **KU**=**F**

K	Matriz $n \times n$ do sistema linear **KU**=**F**
$\tilde{a}, \ldots, \tilde{z}$	Vetores auxiliares
$A, \ldots, Z$	Matrizes
$a, \ldots, z$	Vetores
A^*	Matriz transposta conjugada
A^{-1}	Matriz inversa
A^{-T}	Matriz inversa da transposta
A^T	Matriz transposta
$a_{\cdot,j}$	j-ésima coluna da matriz A
$a_{i,j}$	Elemento da matriz
$a_{i,\cdot}$	i-ésima linha da matriz A
a_i	Elemento do vetor
D	Matriz diagonal
f	Densidade de carga (C/m^3)
F_i	Elemento (i) do vetor **F**
$I, I^{n\times n}$	Matriz identidade
k	Permissividade elétrica (C^2/Nm^2)
K_{ij}	Elemento (i, j) da matriz **K**
L, U	Matriz triangular inferior e superior, respectivamente
M	Pré-condicionador
Q	Matriz ortogonal
u	Potencial elétrico (V)
W_i	Função de teste para o nó i
x_i^j	j-ésimo elemento do vetor x na i-ésima iteração

RHS	*Right-Hand Side* ou vetor de termos independentes
GPGPU	*General Purpose GPU*
HPC	*High Performance Computing*

Sumário

Lista de Figuras vii

Lista de Tabelas ix

Lista de Códigos Fonte xi

Sumário xv

1 Introdução 1

1.1 Métodos sem malhas . . . 5
1.2 Objetivo . . . 7
1.3 Contribuições do trabalho . . . 8
1.4 Estrutura do texto . . . 10

2 Computação científica de alto desempenho 11

2.1 Modelos de execução . . . 14
2.2 Processamento em GPU . . . 14
2.2.1 História da computação em GPU . . . 15
2.2.2 Evolução do *hardware* das GPUs . . . 16
2.3 CUDA . . . 18
2.3.1 Arquitetura . . . 20
2.3.2 Otimização do processamento em GPU . . . 26
2.3.3 Paralelismo dinâmico usando CUDA . . . 28

3 Meshless Local Petrov-Galerkin (MLPG) 33

3.1 Representação do domínio . . . 34
3.2 Modelagem de problemas eletromagnéticos . . . 35
3.3 Forma fraca local . . . 36
3.3.1 Domínio de suporte . . . 38

3.4 Processo de discretização . . . 42
3.5 Imposição de condições de contorno . . . 43
3.6 Funções de teste . . . 44
3.7 Montagem da matriz de rigidez e do vetor F . . . 45

4 Solver **48**
4.1 Conceitos de algebra linear . . . 49
4.1.1 Matrizes esparsas . . . 50
4.1.2 Condicionamento numérico . . . 53
4.1.3 Operações de algebra linear . . . 54
4.2 Solução de sistemas lineares . . . 56
4.2.1 Métodos iterativos . . . 57
4.2.2 Critério de parada . . . 58
4.2.3 Etapa de inicialização . . . 58
4.2.4 Precondicionador . . . 59
4.3 Métodos iterativos não-estacionários . . . 60
4.3.1 Gradientes Conjugados (CG) . . . 62
4.3.2 Gradientes BiConjugados (BICG) . . . 67
4.3.3 Gradientes Conjugados Quadrático (CGS) . . . 73
4.3.4 Gradientes BiConjugados Estabilizado (BICG-Stab) . . . 74
4.3.5 Gradientes BiConjugados Estabilizado Aprimorado (I-BICG-Stab) . . . 76

5 Resultados experimentais **79**
5.1 Geração da nuvem de nós . . . 80
5.2 Perturbação nas coordenadas dos nós . . . 80
5.3 *Solver* . . . 84
5.3.1 Análise de *speedup* . . . 88

6 Conclusão **94**
6.1 Trabalhos futuros . . . 96

Referências Bibliográficas **98**

Capítulo 1

Introdução

Há uma grande variedade de problemas de engenharia que podem ser modelados por equações diferenciais parciais ordinárias, como por exemplo a determinação de potencial ao longo de redes elétricas, cálculo da tensão em estrutura metálica na construção civil, cálculo da razão de escoamento num sistema hidráulico com derivações, previsão de concentração de reagentes sujeitos a reações químicas simultâneas etc (Franco, 2007; Jeffrey, 2002). Na maioria das vezes, tais problemas são de difícil solução analítica, requerendo o uso de algum método numérico para aproximar as equações diferenciais parciais, tipicamente através de ferramentas computacionais.

São exemplos de métodos numéricos o Método de Elementos Finitos — FEM (Zienkiewicz et al., 2013), o Método de Diferenças Finitas — FDM (Taflove & Hagness, 2005) e também os Métodos sem Malha — *Meshless* (Liu, 2002). Todos eles trabalham com o domínio espacial do problema discretizado (Figura 1.1a), seja por um conjunto de nós espalhados pelo domínio (Figura 1.1b), como acontece em métodos sem malha, ou por nós conectados entre si segundo um padrão pré-estabelecido, como acontece em métodos como o FDM e FEM (Figuras 1.1c e 1.1d, respectivamente).

A malha, nos métodos que a utilizam, é necessária para definir dependência e conectividade entre os nós, que se agrupam em elementos (triangulares, quadrados etc), formando assim a base destes métodos. Quando existe uma estrutura pré-estabelecida (a Figura 1.1c apresenta uma malha que utiliza o quadrado como elemento estruturante), a localização dos vizinhos de qualquer elemento é previsível, uma vez que uma regra é seguida na constituição da estrutura

em grade (*grid*). Por outro lado, a fronteira do problema pode não estar conforme tal regra, o que faz com que os elementos limiares não se ajustem corretamente ao contorno.

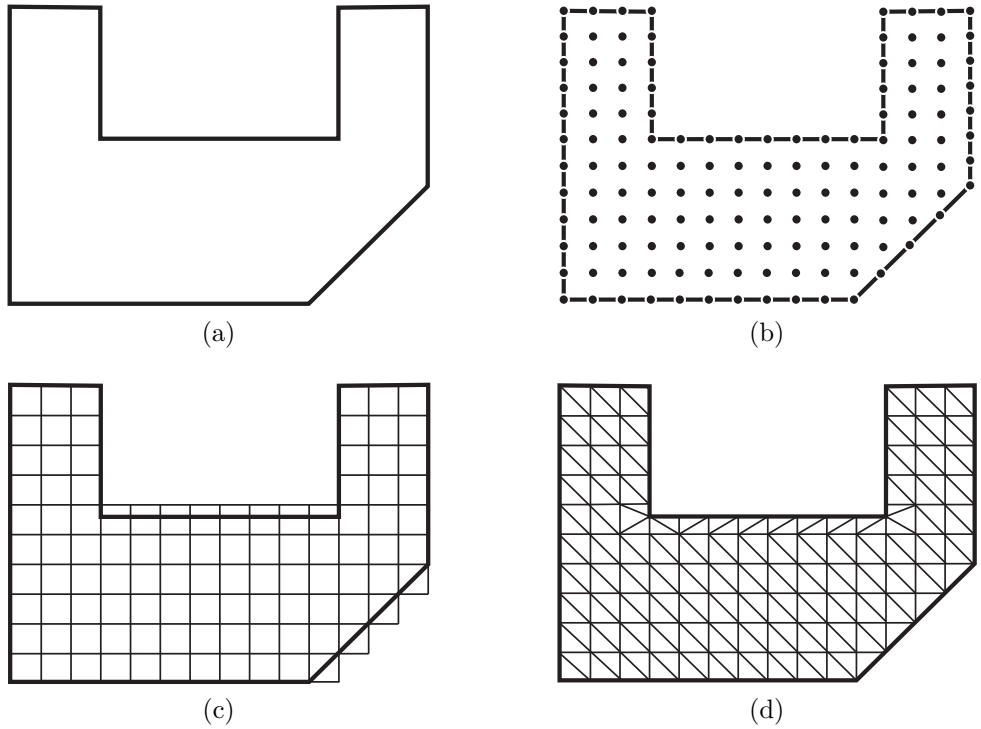
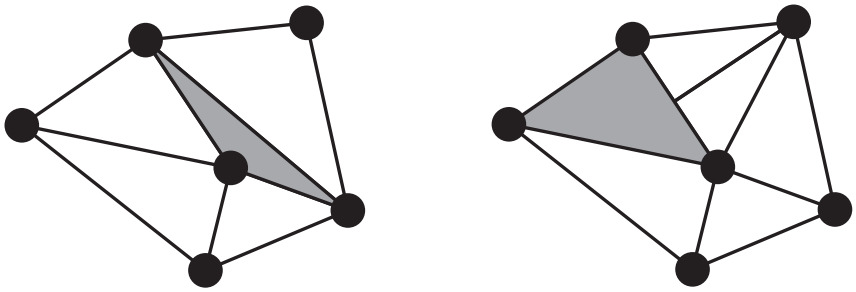


Figura 1.1: Comparação entre formas de discretização de um domínio Ω, cuja superfície é apresentada em (a).

A Figura 1.1d exemplifica uma discretização cujos elementos estruturantes podem assumir formas geométricas não regulares e que não seguem uma estrutura, como por exemplo, usando triângulos ou tetraedros com formas e tamanhos diferentes. Neste caso, a localização dos nós pode variar, permitindo a conformidade entre elementos limiares e fronteiras, além de uma representação com nível de detalhamento heterogêneo ao longo do domínio. Isto permite, por exemplo, aumentar a densidade de elementos em regiões críticas e contribui para a exatidão da representação do domínio. Este tipo de representação é utilizada pelo Método de Elementos Finitos (FEM).

A presença de malhas como essas, apesar de simplificar a solução, torna-a suscetível à qualidade da formação do elemento estruturante (Figura 1.2). Para os problemas em duas dimensões, discretizados usando malhas triangulares, garantir uma qualidade satisfatória da malha é um problema de soluções conhecidas. Mas, quando o domínio se deforma ao longo do tempo, faz-se necessário novo refinamento a cada instante analisado, o que pode inviabilizar a adoção

do método (Chen et al., 2017). Para problemas em três dimensões é ainda pior, necessitando muitas vezes de intervenção manual na geração da malha tridimensional (tetraédrica) de qualidade satisfatória (Amorim, 2011; Lima, 2011).



(a) (b)

Figura 1.2: Exemplos de problemas com elementos estruturantes (em destaque) de uma malha triangular. Um elemento distorcido é exemplificado em (a), apresentando um formato muito distante do triângulo equilátero desejado. Em (b) é exemplificado um elemento não conforme, constituído por quatro segmentos ao invés de três.

Já nos Métodos sem Malhas, pelo simples fato de não existir uma malha, os problemas relacionados deixam de existir. A discretização do domínio é feita espalhando-se nós ao longo de seu domínio e fronteira (Figura 1.1b), e um sistema de equações algébricas é estabelecido sem o uso de malha. Entretanto, a falta de informação de conectividade introduz dificuldades para estes métodos, sendo necessário um espalhamento de nós que garanta a representatividade de todo o domínio e o uso de uma estrutura de dados com informação de vizinhança entre nós. Outro problema é referente ao custo computacional, tipicamente mais elevado quando comparado a Métodos com Malhas (Kamranian et al., 2017; Amorim et al., 2019).

Uma análise usando quaisquer um desses métodos numéricos parte da discretização do domínio usando aproximações polinomiais locais. O processo envolve o cálculo das contribuições, montagem do sistema global (matriz de rigidez) e a solução do sistema de equações lineares resultante. Quando a geometria do problema é complexa ou é necessária uma maior precisão da aproximação do método empregado, aumentamos a quantidade de nós utilizados na discretização, causando um crescimento expressivo da matriz de rigidez e intensificando a exigência de memória. Por este motivo, as limitações de memória e custo computacional ainda restringem a precisão da solução ou mesmo o tamanho do problema físico tratado (Jiang, 1998).

Para contornar esse problema, foram propostos algoritmos eficientes tanto na etapa de montagem da matriz de rigidez global (Fonseca, 2011; Correa et al., 2015; Cecka et al., 2011; Kiss et al., 2012) quanto na solução dos sistemas de equações lineares esparsos gerados (Jiang, 1998; Golub & Ye, 1999; Bruaset & Langtangen, 1997; Benzi, 2002). No entanto, cedo ou tarde tais algoritmos enfrentam complicações com o crescimento da complexidade do problema tratado. Uma abordagem eficaz é subdividir os dados de entrada e fazer uso da computação paralela (Bolz et al., 2003; Cevahir et al., 2009; Refsnæs, 2010). Esta subdivisão, no entanto, deve ser pensada de tal forma que cada parte seja relativamente independente, pois a troca de dados entre as memórias envolvidas pode diminuir a eficiência de cálculo e inviabilizar a estratégia (Liu et al., 2007).

A ideia de divisão dos dados de entrada do problema foi aplicada na análise de elementos finitos de problemas de condução de calor por Hughes et al. (1983), em 1983, onde foram usados como unidade mínima os elementos que compunham a malha de discretização do problema, dando origem à abordagem *Element-by-Element* — EbE. Em seguida, esta estratégia foi aplicada na análise de elementos finitos para problemas de mecânica estrutural (Hughes et al., 1987). Baseado no EbE, Law (1986) forneceu um procedimento completo de solução de elementos finitos e também desenvolveu um algoritmo baseado no método do Gradiente Conjugado, EbE-CG, que poderia ser empregado em um sistema paralelo de memória distribuída. Muitos outros trabalhos surgiram fazendo uso desta abordagem, como por exemplo os apresentados por Ortiz et al. (1983), Winget & Hughes (1985) e Carey & Jiang (1986). Outros ainda inseriram o uso de pré-condicionador para acelerar a convergência do método iterativo adotado para solução do sistema linear resultante, como os trabalhos de Gustafsson & Lindskog (1986), Erhel et al. (1991) e Sheu et al. (1999).

Uma motivação importante para os primeiros trabalhos que propuseram dividir o problema era a limitação de *hardware* da época (Hughes et al., 1983), que inviabilizava o armazenamento do problema inteiro na memória principal. Nos computadores modernos, a motivação principal é outra. Com o surgimento de CPUs (*Central Processing Unit*) com múltiplos núcleos de processamento e das GPUs (*Graphics Processing Units*) com capacidade de computação de propósito geral, possibilitou-se que, a partir da divisão de grandes problemas, suas partes fossem computadas simultaneamente (ou paralelamente) (Almasi & Gottlieb, 1990).

Tradicionalmente, a diferença fundamental entre os processadores (CPUs) e GPUs é o fato de que as CPUs são otimizadas para cálculos sequenciais executados por aplicativos diversos, en-

quanto as GPUs são otimizadas para cálculos maciçamente paralelos. Há um tempo atrás essa divisão era bem mais nítida, já que as placas gráficas processavam apenas triângulos, texturas e efeitos simples. Entretanto, com a introdução do uso de *shaders* (pequenos aplicativos destinados a executarem tarefas específicas na composição das cenas) elas ganharam a capacidade de também executar código de programas de propósito geral em cada um dos seus núcleos, assim como nas CPUs (Nvidia, 2010). Desta forma, além do processamento de gráficos para o qual originalmente foram desenvolvidas, as GPUs são capazes de assumir o processamento de aplicações de propósito geral elaboradas que eram exclusivamente executadas em CPUs.

Neste novo contexto e também devido à popularização das GPUs, surgiram soluções baseadas no método EbE que fazem uso da grande quantidade de unidades de processamento e memória destas placas. É o caso, por exemplo, da abordagem estruturada em blocos apresentada por Refsnæs (2010), da implementação do EbE FEM em conjunto com o Método dos Gradientes Bi-conjugados apresentada por Kiss et al. (2012), e da técnica para a geração de grandes matrizes de elementos finitos em uma estação de trabalho equipada com *cluster* apresentada por Dziekonski et al. (2012). Métodos sem malha também vem explorando computação em GPU mesmo que somente em parte do processo, como os trabalhos de Correa et al. (2015), Shivanian (2015), Zhang et al. (2018) e Karatarakis et al. (2013).

1.1 Métodos sem malhas

Métodos sem malha são apresentados como alternativas principalmente quando os problemas tratados são tridimensionais, possuem geometria complexa ou deformações ao longo do tempo, pois nestes casos o controle de qualidade da malha passa a ser um problema de difícil solução e influencia a precisão dos resultados numéricos (Fonseca, 2011). O simples fato de não utilizar uma malha para estabelecer um sistema de equações algébricas ou não estar sujeito à determinadas propriedades dos elementos estruturantes, torna o método sem malha atrativo nestes casos.

Nestes métodos é usada uma nuvem de nós espalhados pelo domínio e fronteira do problema. A princípio, não há necessidade de nenhuma informação a respeito de conectividade entre nós. Por outro lado, a ausência desta informação dificulta a operação de localização de nós vizinhos

(normalmente nós dentro de um raio circular traçado a partir do nó abordado) (Fonseca, 2011), tarefa necessária para estes métodos. Outro complicador comum entre os métodos sem malhas é a necessidade de cálculos intensos para integrações da forma fraca (Correa, 2014).

Estas dificuldades fazem com que os custos computacionais de métodos sem malhas, quando comparado a métodos com malha, sejam elevados. Portanto, para garantir sua viabilidade frente aos métodos tradicionais, faz-se necessária maior preocupação com o desempenho computacional. Para isso, pode ser adotada a paralelização maciça em GPU, a técnica utilizada na computação científica de alto desempenho que será explorada neste trabalho.

Existem muitos tipos de métodos sem malhas, como por exemplo o *Smooth Particle Hydrodynamics* (SPH) (Gingold & Monaghan, 1977), *Element Free Galerkin* (EFG) (Belytschko et al., 1994), *Meshless Local Petrov-Galerkin* (MLPG) (Atluri & Zhu, 1998), *Point Interpolation Method* (PIM) (Liu, 2002), *Radial PIM* (RPIM) (Liu, 2002) etc. Em particular, escolhido para ser explorado neste trabalho, o MLPG é um método sem malha que utiliza a abordagem Petrov Galerkin na discretização da forma fraca e possui uma formulação local, o que faz dele um método **verdadeiramente sem malha** (Atluri & Zhu, 1998). Ele possui uma particularidade que o torna interessante para a paralelização, sobretudo para computação em GPUs: cada contribuição de nó para a montagem do sistema ocorre em uma única linha da matriz de rigidez (Atluri & Zhu, 1998). Essa propriedade é particularmente importante para este trabalho e será explorada no estágio de solução da matriz de rigidez, pois todos os resultados da etapa são armazenados na memória local da *thread* e a consolidação subsequente dispensa operações atômicas.

No MLPG, a forma fraca, avaliada em um domínio de quadratura local comumente usando formas simples que facilitam a integração pela quadratura de Gauss, é independente entre os nós do domínio e dispensa uma grade de integração como a utilizada no EFG. Cada um destes subdomínios pode ter qualquer tamanho ou forma geométrica, mas o domínio do problema deve ser todo coberto pela união de todos estes subdomínios (Fonseca, 2011). Existem também várias possibilidades para a escolha da função de teste, sendo que a escolhida para ser tratada neste trabalho é a função de Heaviside, cujo valor é unitário no interior do domínio e nulo na fronteira originando a variação do método chamada de MLPG 5 (Atluri & Zhu, 1998). Esta variação consegue boa precisão e permite a simplificação da forma fraca local, eliminando a necessidade de integração no interior dos subdomínios.

Para a solução de um problema usando o MLPG cria-se, inicialmente, a geometria do problema, os nós e os respectivos domínios de quadratura Ω_q. Para cada nó q do domínio, obtém-se seu domínio local correspondente, Ω_q. Para cada ponto de integração $\mathbf{x}_Q$ em Ω_q, determinam-se os nós de suporte e as funções de forma utilizando, por exemplo, o método de mínimos quadrados móveis — MLS em conjunto com o método da penalidade (este último necessário para imposição das condições de contorno de Dirichlet) (Atluri & Zhu, 1998; Liu, 2002). As equações locais são calculadas e inseridas no sistema global. Após montado o sistema, este é resolvido, obtendo-se os parâmetros de cada nó u_i. As aproximações $u^h(\mathbf{x})$ podem, enfim, ser calculadas em todo o domínio do problema.

A integração de um determinado subdomínio de quadratura Ω_q gera uma linha no sistema global. A integração de diferentes nós gera contribuições em linhas distintas do sistema, e cada subdomínio de quadratura pode ser integrado de forma independente e em qualquer ordem, sem necessitar de mecanismos de sincronização (Fonseca, 2011). Trata-se, portanto, de uma característica peculiar e necessária para paralelização usando GPU, o que motivou a adoção do método neste trabalho.

1.2 Objetivo

Métodos sem malhas possuem benefícios claros para determinados problemas da engenharia, sobretudo quando não se pode garantir a qualidade da malha. O desenvolvimento do método, por outro lado, é mais difícil comparado a métodos que possuem uma malha de suporte (De & Bathe, 2001) e isto reflete diretamente no custo computacional.

Para torná-lo atrativo também quanto ao seu desempenho, uma alternativa é fazer uso de técnicas da computação científica de alto desempenho, área onde as GPUs, com suas milhares de unidades de processamento independentes, vêm ganhando grande importância. Já existem soluções nesta arquitetura para tratar a geração das funções de forma, integração numérica, relação de interdependência entre nós e aplicação de condições de contorno, como os trabalhos de Nakata et al. (2010), Kosec & Zinterhof (2013), Zhang et al. (2018), Bergamaschi et al. (2009), Yokota et al. (2009), Gobbetti & Marton (2007) e Correa et al. (2015). No entanto, pesquisas que tratam da solução do problema completa em GPU, subdividindo-o em nós de

forma equivalente ao método dos de elementos finitos Elemento por elemento — EbE-FEM (Pikle et al., 2018; Kiss et al., 2012), ainda estão em fases iniciais. A complicação mais significativa é que, além de calcular as contribuições do nó, também é necessário resolver o sistema de equações sem qualquer movimentação dos dados até então gravados na memória.

Ao analisar a formulação do método *Meshless Local Petrov-Galerkin* (MLPG) (Atluri & Zhu, 1998) em um nível alto, a solução pode ser dividida nos seguintes etapas principais:

1. Geração de geometria do problema, distribuição de nós no domínio e geração de domínios em quadratura;
2. Montagem do sistema de equações lineares através da avaliação da forma fraca local em cada subdomínio;
3. Solução do sistema de equações lineares resultantes;
4. A aplicação dos resultados nos nós de interesse.

O objetivo deste trabalho é, portanto, apresentar uma solução de GPU de ponta a ponta, que engloba a discretização de problemas (etapa 1), a montagem da matriz de rigidez (etapa 2) e a solução do sistema linear (etapa 3). Para avaliar a viabilidade da solução e as métricas de desempenho, é utilizado o problema do capacitor de duas placas paralelas. Apesar da simplicidade do problema, os algoritmos são os mesmos usados para problemas complexos.

1.3 Contribuições do trabalho

A contribuição mais importante é o método *Meshless Local Petrov-Galerkin* (MLPG) ponta a ponta em GPU implementado e tirando proveito máximo da arquitetura SIMT (*Single Instruction Multiple Thread*) do dispositivo. Desde os cálculos iniciais até a entrega da solução para o problema tratado, as etapas integradas são computadas exclusivamente na GPU para evitar custos relacionados à movimentação e conversão de dados. A solução proposta usa a natureza substancialmente paralela do MLPG mantendo cada nó da nuvem em uma *thread* no dispositivo, permanecendo desta forma até a finalização da computação da última etapa do

processo com subsequente entrega dos valores de interesse nos respectivos pontos da nuvem. Dessa maneira, são necessárias estruturas auxiliares mínimas e poucos pontos de sincronização. Variações da solução são testadas para avaliar o impacto efetivo na GPU, suportando decisões mais precisas. Entre o que é testado, estão o posicionamento inicial dos nós no domínio, as técnicas de funções de forma do MLPG e quatro *solvers* diferentes da família dos Gradientes Conjugados.

Durante o processo para alcançar o objetivo do trabalho, alguns feitos foram realizados e listados abaixo.

Artigos publicados em periódicos

- Amorim, L. P., Mesquita, R. C., Goveia, T. D. S., & Correa, B. C. (2019). *Node-to-Node Realization of Meshless Local Petrov Galerkin (MLPG) Fully in GPU*. **IEEE Access**, 7, 151539-151557. DOI: 10.1109/ACCESS.2019.2948134.

Artigos publicados em anais de eventos

- Amorim, L. P., Goveia, T. D. S., Mesquita, R. C., & Baratta, I. *GPU finite element method computation strategy without mesh coloring*. **Proceedings of The IEEE 18th Biennial Conference on Electromagnetic Field Computations (CEFC2018)**, vol. 5, 2018.
- Baratta, I., Silva, E. J., Mesquita, R. C., & Amorim, L. P. *A Domain Decomposition Preconditioner for the Meshless-Local Petrov Galerkin Applied to the Helmholtz Equation*. **Proceedings of The IEEE 18th Biennial Conference on Electromagnetic Field Computations (CEFC2018)**, vol. 5, 2018.

1.4 Estrutura do texto

No Capítulo 2 é abordado o desenvolvimento usando CUDA, que permite a criação de aplicações de propósito geral para execução nas GPUs da NVIDIA. A solução desenvolvida é apresentada no Capítulo 3, que descreve o MLPG, e no Capítulo 4, onde são apresentados os *solvers* usados para a solução do sistema de equações lineares resultante do processamento das contribuições de cada nó da nuvem.

Os resultados individuais e consolidados de cada etapa da solução ponta a ponta são apresentados no Capítulo 5. Por fim, no Capítulo 6 são apresentadas as conclusões e propostas para trabalhos futuros.

Capítulo 2

Computação científica de alto desempenho

A computação científica é uma parte indispensável de quase toda investigação científica e desenvolvimento tecnológico realizados em laboratórios de universidades ou do setor privado (Mackie, 2008). É aplicada sobretudo na construção de modelos matemáticos, como em Anderson et al. (1999) e Davis (2009), e em análises quantitativas que fazem uso de técnicas computacionais para analisar e resolver problemas científicos. Na prática, são simulações de computador e implementações de métodos computacionais para análise numérica que podem fazer uso de estratégias desenvolvidas pela computação de alto desempenho (*High Performance Computing* - HPC) para garantir que seja viável computacionalmente (Idagawa, 2017). Neste ambiente, são utilizadas soluções específicas de *software* em um *hardware* de grande capacidade computacional de forma a atender às demandas de problemas complexos tanto em questões de tempo de execução quanto capacidade de armazenamento.

A evolução da capacidade da computação, tanto voltada para HPC quanto para a computação de propósito geral, segue a Lei de Moore: até meados dos anos 60 não havia nenhuma previsão sobre o futuro do *hardware*, quando Gordon E. Moore profetizou que o número de transistores dos *chips* teria um aumento de 100%, pelo mesmo custo, a cada período de 18 meses. Mas em meados da década de 2000 o comportamento exponencial de algumas características do *hardware* passaram a não seguir o mesmo comportamento, como por exemplo a frenquência e potência (detalhes do comportamento de cada uma destas propriedades ao longo dos anos podem ser observados na Figura 2.1). Consequentemente, o crescimento do desempenho

individual do núcleo do processador também mudou o comportamento. Assim, para garantir a continuidade da evolução da capacidade da computação de forma geral, a saída foi apostar na diminuição do tamanho dos transistores e no aumento do número de núcleos do equipamento.

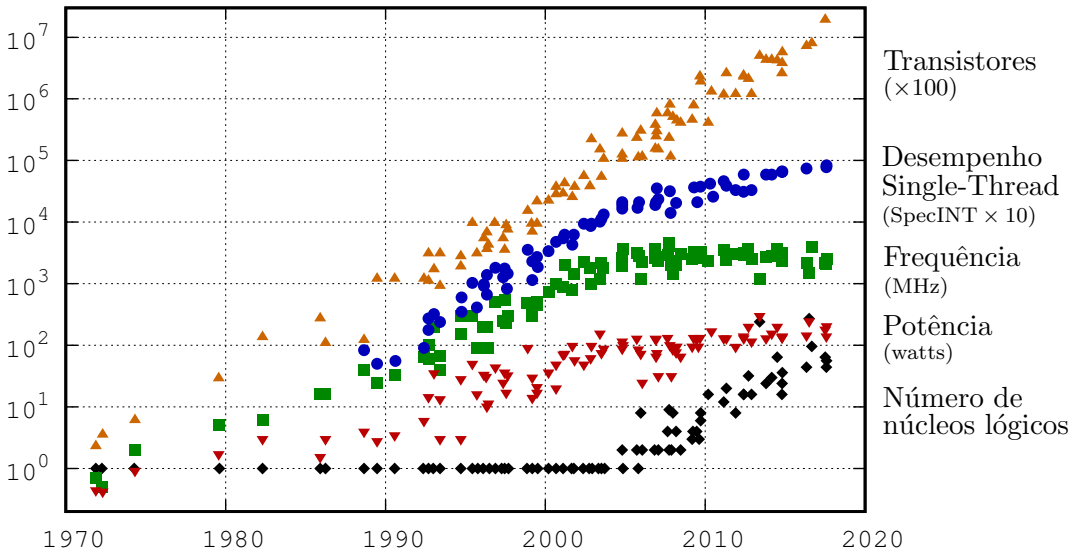


Figura 2.1: Dados de tendências dos últimos 42 anos a respeito do microprocessador. Dados originais até o ano de 2010 coletados e plotados por M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond e C. Batten. Dados coletados para 2010-2017 por K. Rupp (Rupp, 2018).

Com o surgimento de CPUs (*central processing unit*) com múltiplos núcleos de processamento e de GPUs (*graphics processing units*), abriu-se caminho para um tipo de computação em que muitos cálculos são realizados simultaneamente, com base no princípio de que os grandes problemas muitas vezes podem ser divididos em partes menores e, então, resolvidos simultaneamente (ou paralelamente) (Almasi & Gottlieb, 1990). Apesar de o paralelismo ter sido empregado por muitos anos principalmente na computação de alto desempenho, o interesse nele tem crescido nos últimos tempos devido às limitações físicas que impedem a escala de frequência na evolução do *hardware* (Almasi, 1994). Como exemplo importante desta limitação, temos o consumo de energia por computadores e, consequentemente, a geração de calor, que crescem junto com a frequência utilizada mesmo que não linearmente (Asanovic et al., 2006). Desta forma, a computação paralela se tornou um paradigma dominante na arquitetura de computadores.

Computadores paralelos podem ser classificados de acordo com o nível de paralelismo suportado pelo *hardware*. Em computadores multi-processados existem vários elementos de

processamento dentro de uma única máquina, enquanto nos *clusters* ou grades existem vários computadores interconectados trabalhando em uma mesma tarefa. Arquiteturas de computadores paralelos são, por vezes, usadas juntamente com as de processadores tradicionais para acelerar tarefas específicas.

Entretanto, fazer uso de uma arquitetura distribuída não é trivial e, na maior parte das vezes, exige uma engenharia específica no desenvolvimento do *software*. É necessário planejar a solução para que seja *multithreading*, de forma que cada *thread* seja colocada em execução paralela em cada processador, aumentando potencialmente a velocidade de execução. Este melhor desempenho é normalmente calculado usando:

Speedup, definido como relação entre o tempo gasto para execução da tarefa usando processador único, T_1, e N processadores, T_N, na forma (Baer, 2009):

$$\text{speedup} = \frac{T_1}{T_N} \tag{2.1}$$

Lei de Amdahl, lei que governa o *speedup* na programação paralela, que diz que ganho de desempenho obtido pela parte do sistema executada em paralelo está limitada pela fração de tempo que esta parte representa no tempo total de execução (Rodgers, 1985). Para isto, analisa-se o *speedup* e a fração de melhoria, m, que é a fração de tempo dentro do tempo total de execução da tarefa que pode sofrer alteração pelo uso da programação concorrente, na forma (Baer, 2009):

$$\text{ganho} = \frac{1}{(1-m) + (m/\text{speedup})} \tag{2.2}$$

Programas de computação paralela são mais difíceis de escrever do que os tradicionais sequenciais, pois a concorrência introduz diversas novas classes de potenciais erros de *software*, dos quais as condições de corrida são os mais comuns. Comunicação e sincronização entre as diferentes sub-tarefas são tipicamente alguns dos maiores obstáculos para um bom desempenho do programa paralelo.

2.1 Modelos de execução

Quando são tratados modelos de execução para HPC em GPU, dois modelos são referenciados:

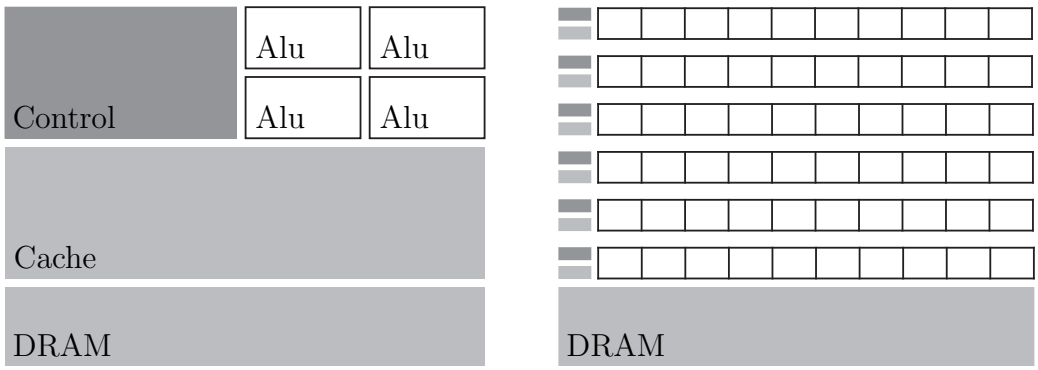
***Single Instruction Multiple Data* (SIMD),** que torna possível executar uma mesma operação paralelamente em várias unidades de processamento, variando-se apenas os dados de entrada para tais operaçoes. Um exemplo disso seria aplicar uma determinada instrução paralelamente à todas as posições de um mesmo vetor. Contudo, para grandes volumes de dados, evidencia-se a necessidade de que os dados sejam agrupados em blocos para permitir um escalonamento;

***Single Instruction Multiple Thread* (SIMT),** que preocupa-se com a alocação e escalonamento das *threads* na execução do programa. Para isto, foram criadas duas unidades, uma para alocação (*Streaming Multiprocessors* — SM) e outra para escalonamento (*warp*) (Kirk & Hwu, 2013). No SM são executadas n *threads* simultâneas (n é uma característica que varia segundo propriedades do *hardware*), mas o escalonamento é feito comumente em grupos (warp) de 32 *threads*, chamados warps, que executam simultâneamente uma mesma instrução (Lindholm et al., 2008).

2.2 Processamento em GPU

As GPUs, unidades de processamento gráfico, são desenvolvidos de forma a atuarem como co-processadores altamente paralelos. Normalmente possuem centenas de núcleos de processador e milhares de segmentos que funcionam simultaneamente sobre estes núcleos (ilustrados na Figura 2.2) e, por causa da capacidade de processamento intensivo, elas são potencialmente muito mais rápidas do que a CPU (Sanders & Kandrot, 2011; Nvidia, 2010; Storti & Yurtoglu, 2015).

No início, GPUs eram usadas apenas para fins gráficos. Mas agora estão se tornando cada vez mais populares para uma variedade de aplicações de uso geral, como em programas de álgebra linear em matrizes (Naumov et al., 2010; Dalton et al., 2014), de simulação, mode-



(a) CPU (b) GPU

Figura 2.2: A GPU, comparando com a CPU, dedica mais transistores para processamento de dados. Figura adaptada de (Nvidia, 2010).

lagem, classificação etc. Os programas concebidos para GPGPU (*General Purpose GPU*) são executados nos vários processadores usando várias *threads* simultaneamente, o que faz deles programas extremamente rápidos.

2.2.1 História da computação em GPU

As primeiras GPUs foram concebidas como aceleradores gráficos, que realizavam blocos de instruções fixas específicas. A partir do final da década de 1990, o *hardware* tornou-se cada vez mais programável. Desde então, os desenvolvedores de jogos deixaram de ser os únicos a fazer trabalhos inovadores com a tecnologia, dando espaço a pesquisadores interessados pelo desempenho nas operações de ponto flutuante, o que deu início à *General Purpose GPU*.

No entanto, o desenvolvimento de aplicações para as GPUs ainda era muito mais complexo do que para as CPUs, mesmo para aqueles familiarizados com linguagens de programação para placas gráficas, como OpenGL. Os desenvolvedores eram obrigados a mapear cálculos científicos sobre problemas que poderiam ser representados por triângulos e polígonos. GPGPU estava praticamente fora dos limites para aqueles que não tinham conhecimento das últimas APIs gráficas, até que um grupo de pesquisadores da Universidade de Stanford começou a

utilizar a GPU como um coprocessador, mais precisamente como um processador de *streaming* (Buck et al., 2004; Owens et al., 2007).

Em 2003, uma equipe de pesquisadores revelou o Brook (Buck et al., 2004), primeiro modelo de programação amplamente adotado para estender `C` com funções de construções de dados paralelo. O sistema compilador Brook expôs a GPU como um processador de propósito geral em uma linguagem de alto nível. Mais importante ainda, os programas de Brook não só eram mais fáceis de escrever do que o código GPU tradicional, mas eram também sete vezes mais rápidos do que o código semelhante existente. A NVIDIA, uma empresa especializada na manufatura de GPUs, sabia que um *hardware* rápido tinha que ser acoplado a um *software* intuitivo, e convidou Ian Buck, responsável pelo Brook, para se juntar à empresa e desenvolver uma solução para funcionar sem problemas na GPU. A NVIDIA CUDA, uma solução de *software* e *hardware* juntos, foi revelada em 2006 como a primeira solução do mundo para a computação de uso geral em GPUs.

Com a CUDA, tornou-se possível enviar código `C`, `C++` e `Fortran` direto para a GPU sem a necessidade de nenhuma linguagem *assembly*. Em pouco tempo, desenvolvedores em empresas como Adobe, ANSYS, Autodesk, MathWorks e Wolfram Research perceberam o potencial da nova tecnologia e começaram a criar soluções de propósito geral de computação científica e de engenharia em uma variedade de plataformas. No geral, as aplicações aceleradas por GPU desenvolvidas executavam (e executam) a parte sequencial de sua carga de trabalho na CPU — o que é otimizado para desempenho *single-threaded* — enquanto aceleram o processamento paralelo na GPU.

Atualmente, CUDA é utilizada por milhares de aplicações e trabalhos de pesquisa publicados e apoiados por uma base instalada de mais de 375 milhões de GPUs CUDA habilitadas em *notebooks*, estações de trabalho, *clusters* de computadores e supercomputadores (Storti & Yurtoglu, 2015; Kirk & Hwu, 2013; Sharma & Han, 2019).

2.2.2 Evolução do hardware das GPUs

A evolução histórica das GPUs considerando apenas uma de suas propriedades é superficial. Nesta seção abordaremos então três propriedades: quantidade de operações usando precisão

dupla, consumo em watts para esta computação e largura de banda. A escolha se deve às propriedades que mais afetam a solução do problema tratado neste trabalho.

A primeira evolução diz respeito à capacidade de computação bruta. GPUs são equipamentos muito bons para fornecer alto desempenho em termos de operações de ponto flutuante (FLOP) por segundo, tanto usando precisão simples quanto dupla. No entanto, a diferença no desempenho entre estas precisões nas GPUs atuais pode chegar a um fator de 32, enquanto que nas CPUs o fator é 2. Mesmo com uma diferença tão grande, é interessante ver é que as GPUs eram praticamente incapazes de lidar com a aritmética de precisão dupla antes de 2008, e demorou até 2010 até que se pudesse ver um ganho teórico significativo sobre o CPU. A Figura 2.3 apresenta a evolução do desempenho teórico quanto à computação em precisão dupla.

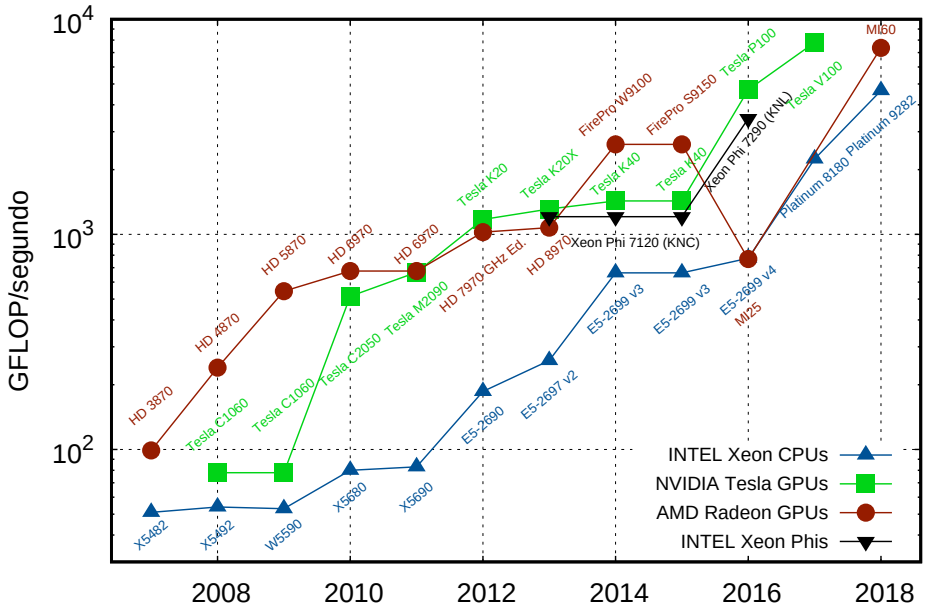


Figura 2.3: Evolução ao longo dos anos do desempenho teórico em precisão dupla (Rupp, 2016).

Como a energia é o fator limitante da computação atual, sobretudo para equipamentos móveis, analisar o consumo energético para realização da computação é importante. Indiretamente, existe a geração de calor e consequentemente, necessidade de dissipação do calor gerado. Esta propriedade e inclusive um dos grandes motivadores do desenvolvimento das arquiteturas paralelas. A Figura 2.4 apresenta a evolução neste sentido. É interessante saber que técnicas de frequência dinâmica de *clock*, como as usadas para CPUs, vem sendo adotadas recentemente pelos fornecedores de GPU e podem controlar ainda mais o consumo de energia a curto prazo.

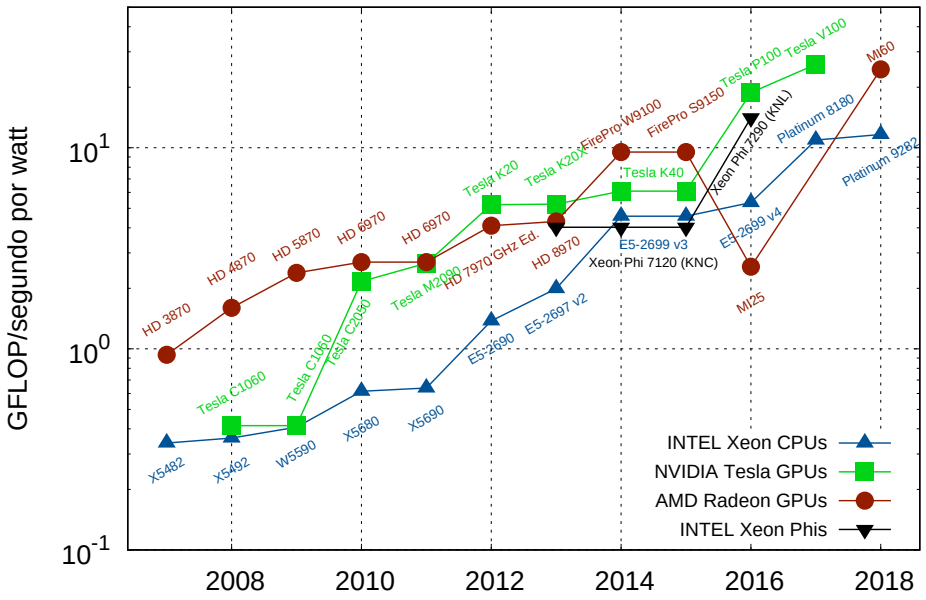


Figura 2.4: Evolução ao longo dos anos da quantidade de operações de ponto flutuante, em precisão dupla, por watt consumido (Rupp, 2016).

Para usar toda a capacidade computacional da GPU de forma eficiente, os dados necessários para a computação devem estar disponíveis no dispositivo. Para estarem lá, em algum momento eles precisam ser copiados da memória do *host* para a memória do dispositivo (*device*) e, eventualmente, copiados de volta. Estas cópias são gargalos em muitas operações, muitas vezes amenizadas por *cache* de dados. Mas, desconsiderando a particularidade de operações que reaproveitam dados previamente copiados, a transferência maciça de dados irá ocorrer e a largura de banda influenciará no tempo gasto para esta ação. A Figura 2.5 apresenta a evolução das GPUs neste sentido.

2.3 CUDA

CUDA é uma linguagem muito semelhante à linguagem C++, adicionadas extensões para que a linguagem utilize os recursos específicos de GPU, o que inclui novas chamadas de API e alguns novos qualificadores de tipo que se aplicam a funções e variáveis. CUDA tem funções específicas chamadas de *kernels*. Um *kernel* pode ser uma função ou um programa completo chamado pela CPU, executado n vezes, em paralelo, usando um número n de *threads* na

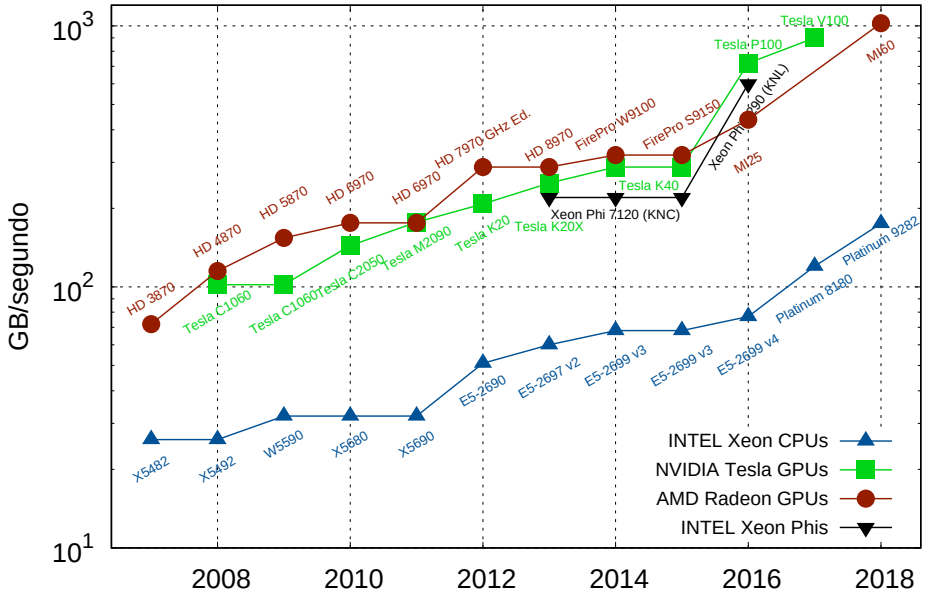


Figura 2.5: Evolução ao longo dos anos do limite de largura de banda de memória (Rupp, 2016).

GPU. CUDA também fornece recursos de memória compartilhada (como pode ser visto na Figura 2.6) e sincronização entre *threads*.

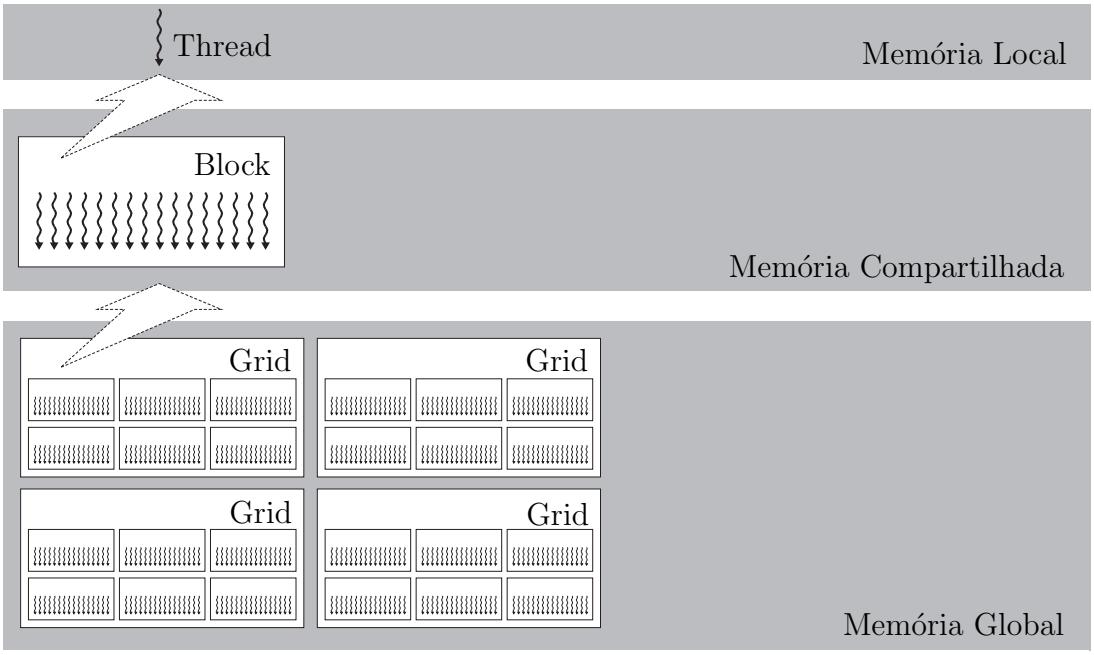


Figura 2.6: A hierarquia entre as estruturas CUDA e o modelo de acesso à memória.

CUDA é suportada em todas as GPUs da NVIDIA desde a G80, lançada em 2006, o que possibilita ser facilmente utilizadas em PCs, *notebooks* e servidores. A Tabela 2.1 lista os termos mais comuns quando falamos a respeito da arquitetura de GPU.

2.3.1 Arquitetura

CUDA, além de uma linguagem, é um paradigma de programação que combina execuções seriais e paralelas e, por isso, é considerada um tipo de programação heterogênea (Figura 2.7). A parte de código em `C` simples é executada de forma sequencial na CPU, também chamado de *host*.

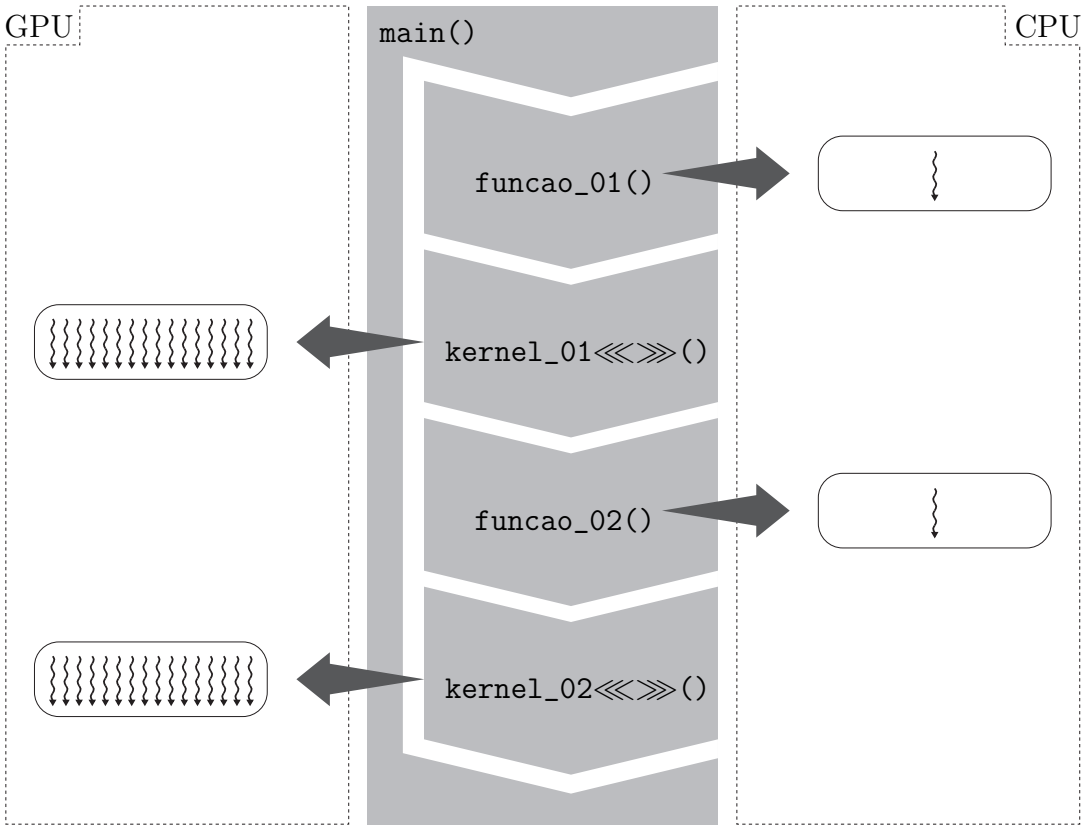


Figura 2.7: Arquitetura heterogênea de um programa em CUDA onde coexistem funções executadas em CPU (`funcao_xx()`) e em GPU (`kernel_xx<<<>>>()`).

Termo (em inglês)	**Significado**
Host	CPU.
Device	GPU.
Kernel	Função executada em paralelo no *device*.
Thread	Um processo no *device* que executa o *kernel*.
Block/Thread Block	Um conjunto de *threads* que compartilham um multiprocessador, o seu espaço de memória compartilhada e primitivas de sincronização.
Warp	*Threads* executadas de forma síncrona dentro de um *block*.
Grid	Um conjunto de *blocks* de execução de um único *kernel*.
Streaming Multiprocessor (SM)	Estrutura responsável por criar, gerenciar e executar as *threads*, utilizando o modelo de execução SIMT — *Single Instruction, Multiple Thread*.
Local Memory	Qualquer dispositivo de memória exclusiva para uma única *thread*. Na maioria das vezes, refere-se aos registradores *on-chip*, mas pode incluir memória *off-chip* em casos *overflow*.
Shared Memory	Memória *on-chip* que pode ser compartilhada entre as *threads* de um único *block*. Memória global *off-chip* que é acessível a todas as *threads*.
Coalesced Memory Access	Múltiplos acessos à memória global que são agrupados em uma transação única no dispositivo. Requer alinhamento de acesso à memória e contiguidade apropriada.

Tabela 2.1: Glossário de termos relativos à arquitetura GPU.

A execução paralela é expressa pela função *kernel*, um bloco de código executado por *threads* em paralelo na GPU, também chamada de *device*. O código do *kernel* é um bloco de código em `C` para apenas uma *thread*. A função *kernel* só pode ser chamada pela parte serial do código executada na CPU. Para que isto aconteça, uma configuração de execução deve ser especificada explicitamente quando esta função é chamada, ou seja, deve ser informado o número de *threads* em um *block* e o número de *blocks* em um *grid*. A hierarquia entre estas estruturas pode ser vista na Figura 2.6.

A estrutura em grid e blocks

Um *grid* consiste em um conjunto de *blocks* com uma, duas ou três dimensões. Cada *block*, por sua vez, consiste em um agrupamento de *threads* com uma ou duas dimensões. Um *block* é um conjunto de *threads* que são executadas em um único SM (*Streaming Multiprocessor*). A Figura 2.6 descreve uma estrutura com *grid* e *block* bidimensionais. Dentro de um *block*, as *threads* são organizadas em *warps*, normalmente em grupos de 32 delas, enviadas juntas para execução. As *threads* de um mesmo *block* são executadas no mesmo multiprocessador e podem se comunicar umas com as outras através de memória compartilhada. É possível, portanto, realizar sincronização entre elas (Developers, 2019).

Para declarar *grid* e *thread blocks* da CUDA é necessário usar um tipo de dado pré-definido como um vetor de inteiros que especifica as dimensões necessárias. Na chamada da função *kernel*, as variáveis que especificam *grid* e *block* são escritas usando três chaves angulares $\lll$`grid,block`$\ggg$. Nesta chamada, *grids* e *blocks* são criados dinamicamente. Os valores das variáveis deve ser menor do que o espaço total alocável, como será tratado com mais detalhes nas seções seguintes. Funções *kernel* tem sempre um tipo de retorno `void`, e o qualificador `__global__` , que significa dizer esta é uma função do *kernel* para ser executado em GPU.

CUDA fornece algumas variáveis internas para usar essa estrutura de forma eficiente. Para acessar o `id` (identificador único) de um `block` usamos a variável `block_Idx`, com valores no intervalo de 0 a `grid_Dim`-1. Intuitivamente, `grid_Dim` é usada para acessar a dimensão do *grid* enquanto a variável `block_Dim` é usada para acessar a dimensão do *block*. Cada *thread* é identificada de forma única dentro do *block* pela variável `thread_Idx`. A variável `WarpSize`

especifica o tamanho do *warp* de *threads*. Todas estas variáveis estão embutidas no *kernel* CUDA, com tamanhos máximos permitidos de cada dimensão do *grid* de 65535, e de 512, 512 e 64 para as dimensões X, Y e Z de cada *block*.

O número de *blocks* alocados para cada multiprocessador depende da necessidade de recursos, como memória, registrador ou *threads*. Quanto mais recursos são necessários, menos *blocks* podem ser alocados para cada multiprocessador de cada vez e, neste caso, os *blocks* restantes têm que esperar a sua vez de ser executado.

Toda a criação, execução e finalização das *threads* é automática e tratada pela GPU, sendo transparente ao programador. A ele cabe apenas especificar o número de *threads* num *block* e o número de *blocks* em um *grid*.

Controle de fluxo

Como a função *kernel* é executada no *device*, a memória necessária para sua execução precisa ser alocada antes que ela seja chamada. Se existem dados necessários ao *kernel* que estão disponíveis apenas na memória principal do computador, *host*, estes dados devem ser copiados para a memória do *device*.

A memória do *device* pode ser alocada tanto de forma linear quanto como matrizes CUDA. A alocação é realizada usando-se o qualificador `__device__` antes da especificação da variável. Isto significa que o espaço alocado passará a existir não no *host*, mas no *device*.

A API CUDA também possui funções para alocar e desalocar memória do *device* em tempo de execução, como `cudaMalloc()`, `cudaFree()` etc. Da mesma forma, após a execução da função *kernel*, dados da memória do *device* devem ser copiados de volta para a memória do *host* a fim de obter resultados. Para copiar dados entre *host* e *device*, a API CUDA fornece funções como `cudaMemCpyToSymbol()`, `cudaMemCpyFromSymbol()`, `cudaMemCpy()` etc. Mantendo tudo isto em mente, o fluxo de processamento é o seguinte:

1. Alocar memória no *host* e *device* separadamente. Memória do *device* é legível e gravável pelo *host* através das funções de cópia de memória. Exemplo de código:

   ```
   int array[total];
   __device__ int darray[total];
   ```

2. Copiar os dados do *host* para *device* utilizando API CUDA, se necessário. Exemplo de código:

   ```
   CudaMemCpy(darray, array, ...);
   ```

3. Executar paralelamente em cada núcleo do *device* a função *kernel*. Exemplo de código:

   ```
   funcArray<<<grid,block>>>(...);
   ```

4. Copiar os dados de volta do *device* para o *host* usando a API CUDA. Exemplo de código:

   ```
   CudaMemCpy(array, darray, ...);
   ```

Sabendo-se que a largura de banda entre a memória do *device* e *host* é muito menor quando comparada com a largura de banda entre espaços de memórias distintos dentro do mesmo *device* ou *host*, deve-se tentar minimizar a transferência de dados entre *device* e *host*. Isto pode ser feito fazendo-se uma cópia em lote de dados entre *device* e *host* ao invés de pequenas cópias, diminuindo o *overhead* total da transferência, ou então transferir apenas dados brutos para o *device*, deixando para ele o trabalho de criação e destruição de estruturas com os dados copiados.

Outra técnica utilizada é a de *pipelining*, que permite a sobreposição da transferência de dados com a execução do *kernel*. Desta forma, podemos ocultar a sobrecarga da transferência de dados e aumentar o desempenho geral (Sharma & Han, 2019).

Usando threads

Para fins de sincronização entre *threads*, a API CUDA fornece uma função de barreira implementada em *hardware*, `syncthreads()`, que atua como ponto de sincronização. As *threads* irão esperar no ponto de sincronização até que todas as demais o tenham atingido. A comunicação entre *threads*, se necessária, é possível através da memória compartilhada do *block*.

Ou seja, só é possível haver sincronização entre *threads* de um mesmo *block* executadas em um mesmo multiprocessador.

Para maximizar a utilização de todo o recurso computacional disponível, a escolha tanto do número de *threads* por *block* quanto do próprio número de *blocks* deve ser criteriosa. Um baixo número de *threads* por *block* causa latência de carga na memória do *device* e a atribuição de um único *block* por multiprocessador faz com que ele fique ocioso durante a sincronização de *threads*. Logo, o ideal seria que houvesse duas vezes mais *blocks* que multiprocessadores no *device*, pelo menos 100 *blocks* por *grid* (Nvidia, 2010), e um número de *threads* múltiplo da quantidade de *warps*, de forma a evitar *warp* sub populado.

Caso o *kernel* que está sendo executado possua um desvio condicional, é esperado que existam *threads* com o fluxo do algoritmo favorável à condição e outras desfavorável. Isso ocasiona uma execução não sincronizada das *threads* do *warp*. Para esses casos, as *threads* que possuam um mesmo fluxo permanecem em execução, enquanto que as com outro fluxo aguardam até o momento em que novamente possam estar sincronizadas (Developers, 2019).

Modelo de memória

Todos os multiprocessadores da GPU acessam uma memória global, também chamada de memória do *device*, normalmente de maior tamanho, usada tanto para distribuir quanto reunir informações necessárias às operações. Entretanto, este tipo de memória é lenta quando comparada a memórias do tipo *cache*. Já a memória compartilhada pelas *threads* de um mesmo *block*, também chamada de *parallel data cache* - PDC, é tipicamente rápida, tanto quanto o acesso a registradores (Developers, 2019).

A memória compartilhada, ao contrário da memória do *device*, é local para cada multiprocessador e permite uma sincronização mais eficiente. Ela é dividida em várias partes: cada *block* dentro do multiprocessador acessa a sua própria parte da memória compartilhada e esta parte da memória não é acessível por qualquer outro *block*, deste ou de algum outro multiprocessador. Todas as *threads* de um *block* compartilham essa parte da memória para operações de leitura e escrita, mas o desenvolvedor precisa usá-la com cuidado devido ao seu

tamanho reduzido. A declaração de variáveis usando memória compartilhada entre *threads* de um mesmo *block* é feita usando-se o qualificador `__shared__` .

Cada *thread* contém sua própria memória local, onde as variáveis locais (registradores) das funções do *kernel* são alocadas. Existe ainda a memória chamada de textura ou *cache* do multiprocessador, usada para acelerar as operações de leitura.

2.3.2 Otimização do processamento em GPU

Todas as *threads* têm pleno acesso a toda a memória global do dispositivo. É possível tanto a leitura quanto escrita em qualquer posição desta memória, mas isto faz com que o *hardware* serialize as transações de acesso. Outra característica do dispositivo que precisa ser considerada pelo programador é sua quantidade e capacidade dos SMs, de forma a não subaproveitar os recursos disponíveis. Algumas boas práticas, discutidas a seguir, precisam ser respeitadas para garantir o resultado esperado quanto ao desempenho da solução.

Simplificação dos kernels

Devido o modelo de execução em GPUs ser SIMT, temos o escalonamento por unidades chamadas *warps*, que são compostas por *threads* que executam uma mesma instrução. Entretanto, é possível que as *threads* de um mesmo *warp* deixem de ser síncronas na situação onde cada *thread* execute condições diferentes em seu fluxo de código no *kernel*. Dessa forma, um *kernel* rebuscado, com muitos laços e condições, diminui a eficiência da execução em *warps* devido à necessidade de re-sincronização entre as *threads*, gerando *overhead*. Portanto, do ponto de vista de tempo total de computação, é mais interessante gerar um *kernel* exclusivo que trate uma condição específica à um *kernel* com um condicional em seu fluxo (Developers, 2019).

Maximização da ocupação da GPU

As GPUs possuem um limite de *threads* e *warps* que podem executar simultâneamente em um SM, além do limite de registradores para cada *thread* e memória compartilhada (Developers, 2019). Essas restrições podem impactar no desempenho de algoritmos para GPGPU (Lobeiras et al., 2013), ao mesmo tempo que pode haver também uma situação de subaproveitamento destes recursos. Dessa forma, a fim de melhorar a ocupação e diminuir qualquer *overhead* envolvido, é possível e recomendável reutilizar a *thread* ou, pelo menos, reutilizar variáveis locais com uma correta utilização de escopos locais no código. Isto faz com que os recursos utilizados por elas sejam liberadas com o término do escopo, possibilitando o uso do recurso em um novo escopo local mais adiante no programa. A correta chamada dos kernels, com dimensionamento de *blocks* correto, também contribui para um melhor aproveitamento do dispositivo (Developers, 2019).

Minimização de acessos à memória global

Apesar do acesso à memória global da GPU ser o mais custoso, ainda assim é a memória mais abundante no dispositivo e é a única forma de comunicação com o *host*. É de se esperar que qualquer otimização precisa considerar a diminuição de requisições feitas a esse recurso. Uma forma de fazer isso é agrupando espaços de memória a serem acessadas pelas *threads* de um *warp* de forma que estejam contíguos (Developers, 2019; Cecka et al., 2011). Desta forma, é aproveitada a alta largura de banda desta memória, maximizando a quantidade de dados obtidos em cada acesso. Outra forma é utilizar memória compartilhada, de acesso menos custoso, como *cache* gerenciado pelo desenvolvedor. Ou seja, no início da execução os dados da memória principal são copiados para a memória compartilhada, e ao término os dados de resposta são copiados de volta. Nesta caso, o programador deve avaliar a necessidade de sincronização entre *threads* para garantir a consistência dos dados e também um possível subaproveitamento do dispositivo, como efeito colateral do aumento da memória compartilhada.

2.3.3 Paralelismo dinâmico usando CUDA

Paralelismo dinâmico é um recurso que pode ser utilizado na programação CUDA em dispositivos de *compute capability 3.5* ou superior (Nvidia, 2015), e permite um *kernel* CUDA criar e sincronizar com o novo trabalho diretamente na GPU, em qualquer ponto do programa. Este recurso é uma alternativa ao controle antes exclusivo exercido pelo *host* (CPU) do que é executado na GPU, permitindo que o próprio dispositivo decida sobre a execução de sub-tarefas em tempo de execução aproveitando os dados já disponíveis, sem abrir mão do balanceamento de carga. Antes deste recurso não era possível usar recursão, estrutura de laço irregular ou qualquer abordagem que poderia diferenciar comportamentos entre *threads* de uma *grid*.

O modelo de execução CUDA é baseada em *threads*, *thread blocks*, e *grids*, com *kernel* definindo o que será realizando em uma *thread* individualmente dentro do *thread blocks* ou *grid*. No momento da chamada de um *kernel* pelo *host*, as propriedades do *grid* são descritas por uma configuração de execução que tem uma sintaxe especial em CUDA. O que o recurso de paralelismo dinâmico inseriu foi a capacidade do dispositivo, através das *threads* já em execução, realizar também as tarefas de configuração, execução e sincronização de novas *grids* de *threads* vinculadas hierarquicamente, ou seja, uma *grid* mãe chamando a execução de uma *grid* filha.

O aninhamento entre as *grids* implica que uma *grid* mãe não pode ser considerada finalizada até que todas as *grids* filhas, executadas por ela, sejam também finalizadas, sendo que a própria execução se responsabiliza por esta sincronia dentro de um mesmo *block*. A Figura 2.8 ilustra esta situação: uma *thread* em execução na CPU executa uma *grid* em GPU, `Grid A`, e uma das *threads* da `Grid A` executa uma outra *grid*, `Grid B`. Todas as *threads* da `Grid A`, mesmo finalizadas, aguardam o término de todas as *threads* da `Grid B`, assim como a *thread* da CPU aguarda o término de todas as *threads* do `Grid A` para continuar seu trabalho ou ser considerada também finalizada.

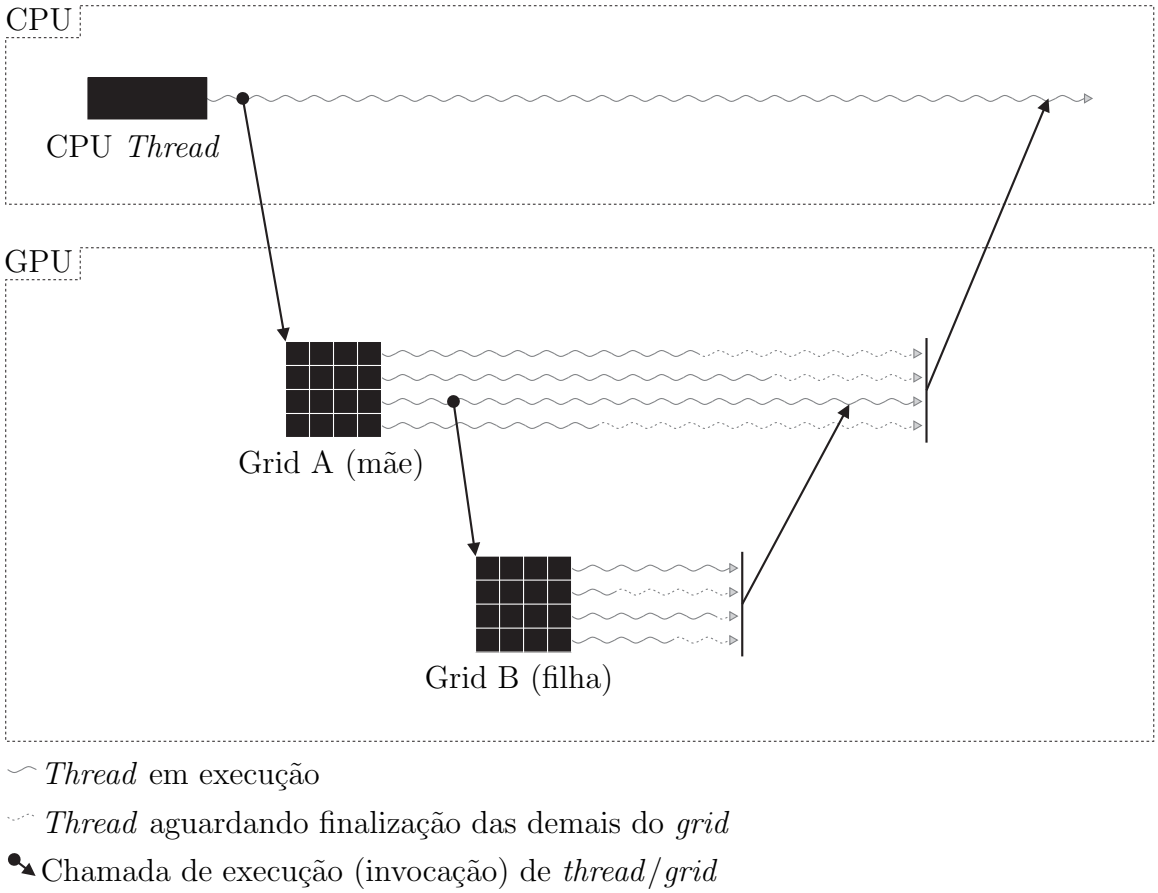


Figura 2.8: Execução aninhada de *grid* de *threads* em GPU

Aninhamento de grids de threads

O compartilhamento de informações entre as *threads*, de forma equivalente quando chamadas pelo *host*, só existe entre *threads* de um mesmo *block* na *grid*, independente desta ser a *grid* mãe ou filha. A sincronização de *threads*, no entanto, é uma operação possível para a *thread* da *grid* mãe, em um *stream* criado no mesmo *block* (usando a primitiva de sincronização `__syncthreads()`) ou implicitamente entre *threads* de um mesmo *block*.

Streams e *events* são recursos que CUDA oferece para controlar dependências entre *grids*, de forma que quando chamados em um mesmo *stream* os *grids* são executados em ordem. Já *events* controlam dependências entre *streams*. A sincronização é garantida implicitamente no próprio *block* com todas as dependências resolvidas de forma adequada. No entanto, o

comportamento das operações em um *stream* que for modificado fora do escopo do *block* é indefinido.

O paralelismo dinâmico permite a concorrência de forma mais fácil dentro de um programa, mas sem introduzir quaisquer novas garantias de concorrência dentro do modelo de execução CUDA, ou seja, não há nenhuma garantia de execução concorrente entre qualquer *thread blocks* no dispositivo. O mesmo problema acontece entre *thread blocks* e suas *grid* filhas, onde o começo da execução da filha só é garantido após uma sincronização explícita (usando, por exemplo, `cudaDeviceSynchronize()`).

Não há nenhum suporte multi-GPU a partir de um programa em execução do dispositivo, pois seu escopo de operação é o dispositivo sobre o qual está atualmente em execução. É permitida, no entanto, consultar propriedades de qualquer dispositivo CUDA no sistema.

Modelo de memória no paralelismo dinâmico

Grids mães e filhas partilham constantes e a mesma memória global, mas tem memórias local e compartilhada distintas. Quanto à memória global, existem dois momentos onde a consistência é garantida:

1. No momento que é chamada pela *grid* mãe;
2. Após a sincronização executada pela *thread* mãe sobre a *grid* filha.

O Código Fonte 2.1 apresenta um exemplo de programação dinâmica usando CUDA que aborda o problema da gestão de memória. Para tanto, ele parte de um vetor de valores inteiros com 255 posições, de nome `data[]`, cujos valores não foram inicializados pelo *host* (CPU).

No algoritmo, o *host* executa uma chamada de função (linha 35) a ser executada no dispositivo (GPU), de nome `parent_launch(...)`, que realiza o preenchimento do vetor de forma que cada posição receba como valor o próprio índice (linha 12). Depois, dentro ape-

nas da *thread* de índice '0', é executada uma nova chamada de função (linha 21), de nome `child_launch(...)`, que realiza incremento de valor em cada posição do mesmo vetor.

```
1  //Códido da função filha no dispositivo (GPU)
2  __global__ void child_launch(int *data) {
3
4    //Ação de incremento do valor do vetor na posição do próprio índice da
       thread
5    data[threadIdx.x] = data[threadIdx.x]+1;
6  }
7
8  //Códido da função principal (mãe) no dispositivo (GPU)
9  __global__ void parent_launch(int *data) {
10
11   //Atribuição do valor do próprio índice da thread como valor na
      respectiva posição do vetor
12   data[threadIdx.x] = threadIdx.x;
13
14   //Barreira de sincronização: a execução só continua após todas as posi
      ções do vetor, onde cada uma é tratadas por uma thread diferente,
      terem recebido valor
15   __syncthreads();
16
17   //Bloco a ser executado apenas pela thread de índice '0'
18   if (threadIdx.x == 0) {
19
20     //A thread mãe (apenas a de índice '0') executa chamada de função (
      threads filhas)
21     child_launch<<< 1, 256 >>>(data);
22
23     //Barreira de sincronização do dispositivo
24     cudaDeviceSynchronize();
25   }
26
27   //Nova barreira de sincronização
28   __syncthreads();
29 }
30
31 //Código executado no host (CPU)
32 void host_launch(int *data) {
33
34   //CPU executando chamada de função na GPU (threads mães)
35   parent_launch<<< 1, 256 >>>(data);
36 }
```

Código Fonte 2.1: Código fonte em CUDA exemplificando o paralelelismo dinâmico.

O método `child_launch(...)`, executado pelas *threads* da *grid* filha, só tem garantia de acesso a dados atualizados até antes de sua chamada. Em termos práticos, o comando

`__syncthreads()` da linha 15) garante que todas as *threads* da *grid* filha tenham visão consistente dos dados do vetor `data[]` (`data[0]=0, data[1]=1, ..., data[255]=255`). Se não fosse por este comando, somente `data[0]` seria visto pelas *threads* filhas.

Quando a *grid* filha termina sua execução e sincronização (linha 24), a sua *thread* mãe (cujo `threadIdx.x` é '0') terá acesso atualizado dos dados. No entanto, para as demais *threads* da mesma *grid* da *thread* mãe, esta atualização só se torna visível a partir da execução do comando `__syncthreads()` da linha 28.

Capítulo 3

Meshless Local Petrov-Galerkin (MLPG)

Nos métodos mais tradicionais, como o Método dos Elementos Finitos (FEM) (Hughes, 2000) e o Método das Diferenças Finitas (FDM) (Taflove & Hagness, 2005), são utilizadas malhas para auxiliar na solução do sistema linear gerado, as quais definem a relação de dependência e conectividade entre os nós (Fonseca, 2011). A presença desta malha simplifica a solução, uma vez que é dispensado o cálculo da vizinhança de nós, contudo, a obtenção e precisão do resultado se tornam suscetíveis à qualidade dos elementos que compõem a malha. Em problemas 2D discretizados usando malhas triangulares, garantir uma qualidade satisfatória da malha é um problema de soluções conhecidas. Mas quando temos domínios que se deformam ao longo do tempo, se faz necessário novo refinamento a cada instante, o que pode inviabilizar a adoção do método.

Nos métodos sem malhas, pelo simples fato de não necessitar de uma malha, estes problemas deixam de existir. A discretização do domínio é feita espalhando-se nós ao longo de sua área e contorno, e um sistema de equações algébricas é estabelecido sem o uso de uma malha. Entretanto, a falta de informação de conectividade introduz dificuldades para tais métodos, sendo necessário um espalhamento de nós que garanta a representatividade de todo o domínio e o uso de uma estrutura de dados com informação de vizinhança entre nós. Outro problema, motivador deste trabalho, é que o custo computacional é mais elevado quando comparado com métodos com malhas (Fonseca, 2011).

O *Meshless Local Petrov-Galerkin* (Atluri & Zhu, 1998) é um método sem malha que utiliza a abordagem Petrov-Galerkin na discretização da forma fraca e possui uma formulação local em relação ao nó, o que faz dele um método verdadeiramente sem malha. O método permite ainda que as funções de forma sejam escolhidas independente das funções de teste, possibilitando que a forma integral do método dos resíduos ponderados fique confinada a um subdomínio pequeno em torno de cada nó. Estas propriedades favorecem o paralelelismo do algoritmo no sentido de isolar e simplificar a computação de cada nó da nuvem, fato particularmente importante para o objetivo deste trabalho.

3.1 Representação do domínio

A discretização do domínio (Ω) e fronteiras (Γ) de um problema para o MLPG (Figura 3.1a), assim como para outros métodos sem malha, é realizada através de uma nuvem de nós dispostos no seu interior e ao longo do contorno (Figura 3.1b). A fronteira pode ser de dois tipos: onde se impõe a condição de contorno essencial — fronteira de Dirichlet (Γ_{u1} e Γ_{u2}), e onde se tem a condição de contorno natural — fronteira de Neumann (Γ_{t1} e Γ_{t2}).

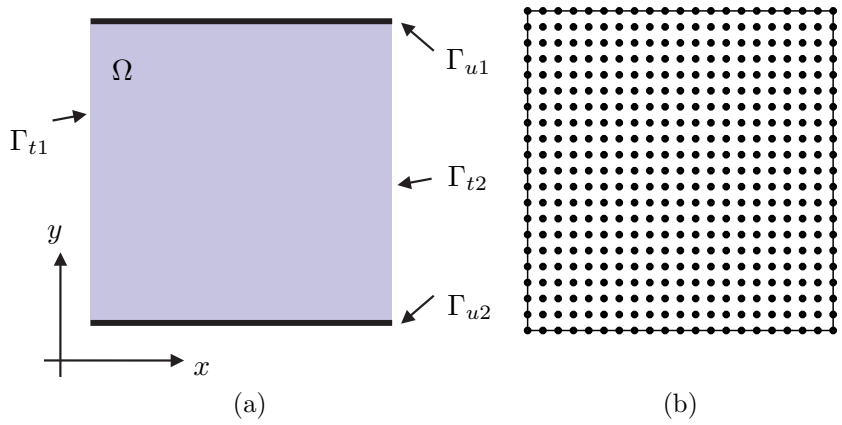


Figura 3.1: Em (a) é representado o domínio (Ω) e fronteiras (Γ, sendo Γ_{u1} e Γ_{u2} as fronteiras de Dirichlet e Γ_{t1} e Γ_{t2} as fronteiras de Neumann) do capacitor de duas placas. Em (b) o domínio é representado usando uma nuvem de nós.

Cada nó distribuído no domínio Ω possui seu próprio subdomínio (Ω_q) e fronteiras (Γ_q), de qualquer tamanho ou formato geométrico. No entanto, para simplificar, são utilizadas formas

como quadrados, retângulos, círculos e elipses para o caso bidimensional, e esferas, elipsóides e paralelepípedos para o caso tridimensional. Os subdomínios podem se sobrepor e a única restrição é que todo o domínio global seja coberto pela união dos subdomínios (Fonseca et al., 2010).

3.2 Modelagem de problemas eletromagnéticos

Os problemas eletromagnéticos, tais como o exemplo usado neste trabalho, são descritos matematicamente através das equações de Maxwell, das relações constitutivas dos materiais do domínio e das condições de contorno (Balanis, 1989).

A modelagem proposta neste trabalho resulta em uma equação diferencial que pode ser generalizada como uma função escalar $u : \Omega \rightarrow R$ que satisfaz:

$$\nabla \cdot (k \nabla u) = f \qquad em \quad \Omega \tag{3.1}$$

$$u = g \qquad em \quad \Gamma_u \tag{3.2}$$

$$-k \frac{\partial u}{\partial n} = \bar{t} \qquad em \quad \Gamma_t \tag{3.3}$$

onde,

k é a permissividade elétrica (C^2/Nm^2),

u é o potencial elétrico (V),

f é a densidade de carga (C/m^3),

Γ_u é a fronteira de Dirichlet,

$\bar{t}$ é a condição de fronteira de Neumann,

Γ_t é a fronteira de Neumann.

3.3 Forma fraca local

Para calcular a contribuição nodal na matriz de rigidez global através da integração da forma fraca apresentada adiante, é necessário que sejam definidos os subdomínio e fronteira de cada nó da nuvem. Considerando um nó i da nuvem, a fronteira Γ_q^i de seu subdomínio é a união de sua fronteira interna ao domínio global (Γ_{qi}^i) e a interseção entre as fronteiras do subdomínio e global ($\Gamma_{qu}^i \cup \Gamma_{qt}^i$) (Figura 3.2).

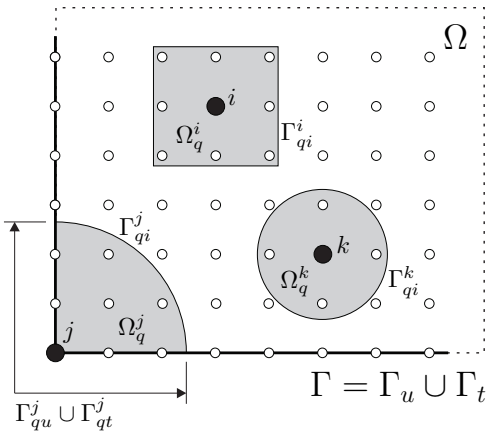


Figura 3.2: Representação de uma parte do domínio (Ω) usada pelo MLPG. Em destaque, os subdomínios Ω_q^i, Ω_q^j e Ω_q^k e suas fronteiras Γ_q^i, Γ_q^j e Γ_q^k, respectivamente para os nós i, j e k.

Para o subdomínio Ω_q^i, o problema é representado por meio do método de resíduos ponderados:

$$\int_{\Omega_q^i} W_i \left[\nabla \cdot (k \nabla u) - f\right] d\Omega = 0 \tag{3.4}$$

onde W_i é a função de teste associada ao nó i. Geralmente, funções com suporte compacto são usadas como funções de teste, o que significa que elas são diferentes de zero em uma região limitada, cancelando-se fora dessa região. Isso faz com que a integral seja calculada apenas na região onde a função de teste é diferente de zero. Além disso, para simplificar a implementação, o suporte da função de teste geralmente corresponde ao subdomínio do nó (Fonseca, 2011). Separando-se os termos da Equação 3.4, tem-se

$$\int_{\Omega_q^i} W_i \nabla \cdot (k \nabla u)\, d\Omega - \int_{\Omega_q^i} W_i f d\Omega = 0. \tag{3.5}$$

Aplicando-se a identidade vetorial $\nabla \cdot (g\mathbf{v}) = \nabla g \cdot \mathbf{v} + g\nabla \cdot \mathbf{v} \implies g\nabla \cdot \mathbf{v} = \nabla \cdot (g\mathbf{v}) - \nabla g \cdot \mathbf{v}$ no primeiro termo da Equação 3.5, tem-se

$$\int_{\Omega_q^i} W_i \nabla \cdot (k\nabla u)\, d\Omega = \int_{\Omega_q^i} \nabla \cdot (W_i k\nabla u)\, d\Omega - \int_{\Omega_q^i} \nabla W_i \cdot k\nabla u d\Omega. \tag{3.6}$$

Aplicando-se o teorema da divergência, $\int_\Omega \nabla \cdot \mathbf{A} d\Omega = \int_\Gamma \mathbf{A} \cdot \mathbf{n} d\Gamma$, onde $\Gamma = \partial\Omega$, no primeiro termo da direita na Equação 3.6, tem-se

$$\int_{\Omega_q^i} \nabla \cdot (W_i k\nabla u)\, d\Omega = \int_{\Gamma_q^i} (W_i k\nabla u) \cdot \mathbf{n} d\Gamma. \tag{3.7}$$

Substituindo-se as Equações 3.6 e 3.7 na Equação 3.5, tem-se

$$\int_{\Gamma_q^i} (W_i k\nabla u) \cdot \mathbf{n} d\Gamma - \int_{\Omega_q^i} \nabla W_i \cdot k\nabla u d\Omega - \int_{\Omega_q^i} W_i f d\Omega = 0. \tag{3.8}$$

Como $\Gamma_q = \Gamma_{qu} \cup \Gamma_{qt} \cup \Gamma_{qi}$ (Figura 3.2) e $k\nabla u \cdot \mathbf{n} = k(\partial u/\partial n) = \bar{t}$ em Γ_{qt} (condição de fronteira de Neumann), subdividindo-se o primeiro termo da Equação 3.8 tem-se

$$\int_{\Gamma_q^i} (W_i k\nabla u) \cdot \mathbf{n} d\Gamma = \int_{\Gamma_{qu}^i \cup \Gamma_{qi}^i} (W_i k\nabla u) \cdot \mathbf{n} d\Gamma + \int_{\Gamma_{qt}^i} W_i \bar{t} d\Gamma. \tag{3.9}$$

Assim, a partir da Equação 3.8, tem-se

$$\int_{\Gamma_{qu}^i \cup \Gamma_{qi}^i} (W_i k\nabla u) \cdot \mathbf{n} d\Gamma + \int_{\Gamma_{qt}^i} W_i \bar{t} d\Gamma - \int_{\Omega_q^i} \nabla W_i \cdot k\nabla u d\Omega - \int_{\Omega_q^i} W_i f d\Omega = 0, \tag{3.10}$$

que chamamos de **forma fraca** para o problema estático/magnetostático em duas dimensões.

A forma fraca é avaliada no domínio de quadratura local (Ω_q^i) e pode ser calculada independentemente para cada nó do domínio. Essa característica é essencial para o propósito deste trabalho. As funções de forma usadas para aproximar podem ser avaliadas por vários métodos, entre eles (Liu, 2009):

- Método dos mínimos quadrados móveis — MLS;

- Método de interpolação de pontos usando funções de base radial e polinômios de primeira ordem — RPIMp.

O MLS é considerado um das melhores formas para aproximar dados com precisão. Uma propriedade desse método é que as funções de forma não possuem a propriedade do delta de Kronecker. Como consequência, as condições essenciais de contorno não podem ser impostas diretamente, e um termo de penalidade é adicionado à forma fraca (Liu, 2002), como mostra a Seção 3.5.

Comparado ao MLS, o RPIMp tem uma vantagem significativa, como mostrado na Seção 3.5: não requer nenhum método adicional para impor as condições de contorno essenciais, pois possui a propriedade do delta de Kronecker. No entanto, se o método usar apenas funções radiais (como é o caso do RPIM), ele não aproxima as funções lineares de forma consistente. Portanto, para garantir consistência, um polinômio linear é adicionado à base do método (Fonseca et al., 2008).

3.3.1 Domínio de suporte

Para se calcular as integrações para o subdomínio de um nó i (Ω_q^i), é necessário determinar o conjunto de nós que será utilizado na geração das funções de forma nesse subdomínio. Este conjunto de nós é conhecido como domínio de suporte (Liu, 2002).

No MLS, cada nó tem associada uma pequena região, conhecida como domínio de influência. Essa é a região influenciada pelo nó, isto é, a região em que o nó participará da determinação das funções de forma. A Figura 3.3a mostra que os domínios de influência dos nós m e n interceptam o subdomínio do nó i e, portanto, participarão do seu domínio de suporte. Por outro lado, o nó k não participa deste domínio de suporte, pois seu domínio de influência não intercepta o subdomínio do nó i.

No RPIMp não existe o conceito de domínio de influência. Nesse caso, o que é feito é, arbitrariamente, definir que o domínio de suporte será composto por k-nós e buscar, então, os k-nós mais próximos do subdomínio de integração. Na Figura 3.3b, por exemplo, bastaria

determinar quais os nós que estão dentro de uma caixa de pesquisa de lado $2(R_a + R_b)$, em que R_b é o raio dos domínios de influência dos nós e R_a é a metade do lado que define o subdomínio do nó i. Os nós que se encontram a uma distância superior a $R_a + R_b$ do nó não influenciam na integração, pois as funções de aproximação e derivadas desses nós são nulas na região de integração.

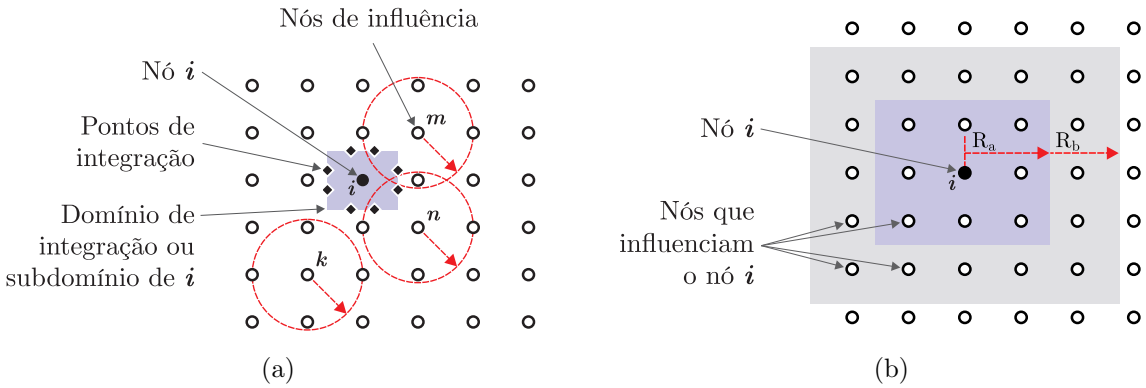


Figura 3.3: Determinação do domínio de suporte do MLS (a) e do domínio de suporte do RPIMp (b).

Determinação de vizinhaça entre os nós da nuvem

Nos métodos baseados em malha, como FEM e FDM, malhas são usadas como base para a identificação de elementos vizinhos, definindo a relação de dependência e conectividade entre os nós (Fonseca et al., 2010). Nos métodos sem malha (*meshless*), as informações de vizinhança entre os nós são construídas considerando a proximidade entre eles.

A determinação dos k-vizinhos de cada nó é uma das primeiras tarefas a serem realizadas, particularmente importante para determinar o conjunto de suporte de um ponto. Este é, portanto, um problema cujo custo computacional pode crescer de forma exponencial com o crescimento do número de nós e, por isso, diversas técnicas foram desenvolvidas com o intuito de viabilizar sua computação.

A solução desenvolvida pressupõe que as regiões vizinhas são circulares e que contenham um número k de nós, como mostra a Figura 3.4. Dessa maneira, a determinação da vizinhança pode ser reduzida ao problema conhecido dos vizinhos mais próximos (knn).

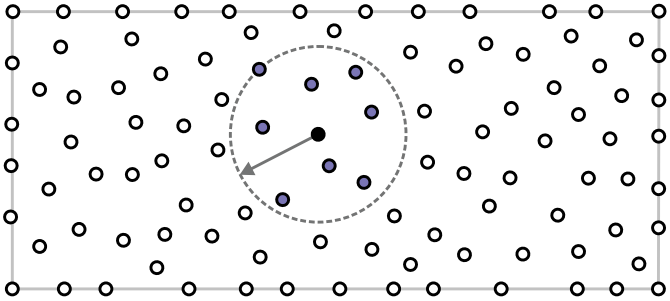


Figura 3.4: Representação de domínio e região de vizinhança (círculo tracejado) de um nó com $k = 9$.

Uma maneira simples de determinar os k-nós pertencentes a cada região de vizinhança seria calcular sua distância para todos os nós no domínio — solução que é chamada de Força Bruta (BF). No entanto, esse algoritmo é muito caro do ponto de vista computacional, na ordem de $O(n^2)$, onde n é o número de nós. Assim, muitos dos métodos propostos para otimizar o knn se concentram na redução da distância computada, utilizando estruturas de dados como árvores binárias.

Com o potencial das GPUs para a programação de propósito geral (GPGPU), novas possibilidades para otimização das soluções para esse tipo problema surgiram, permitindo que sejam viáveis mesmo em aplicações com grande quantidade de nós.

Método da força bruta

Um dos algoritmos usados na busca dos k-vizinhos mais próximos é comumente chamado de Força Bruta: para cada nó de interesse calcula-se de forma exaustiva sua distância para todos os nós do domínio considerado (Garcia et al., 2008). Considera-se um domínio $R = \{r_1, r_2, \cdots, r_n\}$, onde n é o número de nós do domínio. Para que sejam identificados os k-nós de interesse (nós vizinhos) nesse mesmo espaço, tem-se os seguintes passos realizados para cada nó $r_i \in R$:

1. Cálculo da distância euclidiana entre o nó de interesse r_i e r_j, com $j \in [1, n]$;
2. Ordenação de forma decrescente das distâncias calculadas;

3. Selecão dos k-nós do domínio com menores distâncias.

A escolha do método de ordenação é importante na eficiência desse algoritmo. Apesar de o algoritmo *Quicksort* ser conhecido como um dos que apresentam melhores desempenhos (Ye et al., 2010), CUDA não permite implementações recursivas e, portanto, ele não pode ser utilizado nessa aplicação. Dessa forma, optou-se pelo *Insertion Sort* modificado que resulta em um vetor com apenas os k-elementos de menores distâncias (Garcia et al., 2008).

O método é simples de implementar e altamente paralelizável. Portanto, adequado para implementação em GPUs. Por outro lado, é um método de grande complexidade computacional: $O(n^2)$ para calcular as distâncias e $O(n \log n)$ para ordená-las. Por este motivo, soluções que buscam alto desempenho precisam de outra forma de resolver o problema da busca por vizinhos.

Método dos grids

A estrutura do *grid* nada mais é do que uma divisão de domínio em células regulares, como quadrados. Depois que essa estrutura é criada e os nós são distribuídos pelas células, a pesquisa de vizinhos mais próximos é feita apenas nas células ao redor do nó de consulta. Assim, é reduzido o custo computacional de percorrer todo o domínio, como pode ser visto na Figura 3.5. O algoritmo segue as seguintes etapas, para cada nó do domínio:

1. Criação de um *grid* que decomponha a região que contém os nós de interesse;
2. Distribuição dos nós e registro de seus locais;
3. Pesquisa local em células adjacentes no *grid* para encontrar os k-nós vizinhos mais próximo do nó de interesse;

Cada uma dessas etapas pode ser realizada em paralelo na GPU. Uma *thread* pode ser iniciada para cada nó no *grid*. O mesmo pode ser feito na etapa de pesquisa dos k-vizinhos. A estrutura de dados de saída do algoritmo contém as coordenadas dos nós vizinhos mais próximos de cada nó da nuvem e o raio definido a partir da distância desse nó até seu k-vizinho (o nó mais distante do conjunto dos nós vizinhos).

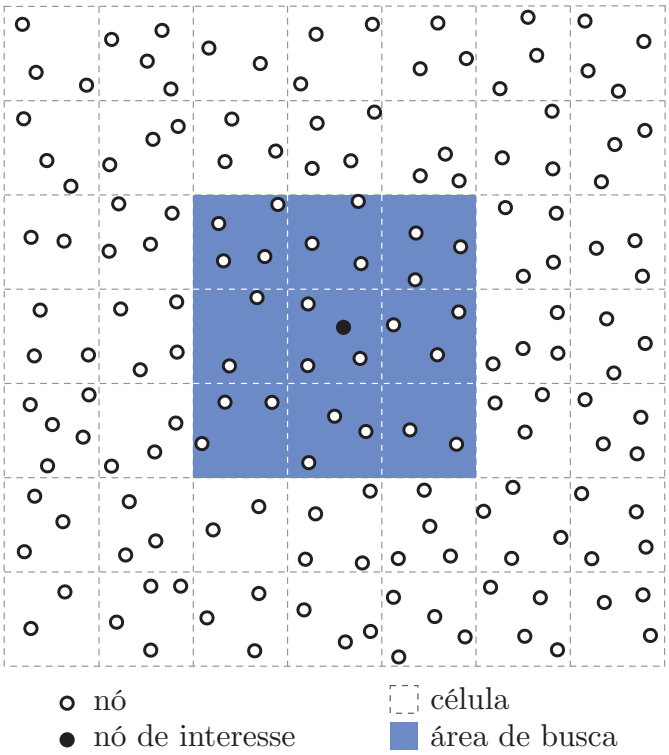


Figura 3.5: Otimização do algoritmo Knn usando subdivisão de espaço.

3.4 Processo de discretização

Para a discretização da forma fraca, a função u dada pela Equação 3.10 é aproximada por u^h:

$$u^h = \sum_{j=1}^{n} \phi_j u_j, \tag{3.11}$$

em que ϕ_j é a função de forma, u_j são os valores nodais e n é o número de nós distribuídos pelo domínio (Fonseca et al., 2010). Portanto:

$$\sum_{j=1}^{n} \left[\int_{\Gamma^i_{qu} \cup \Gamma^i_{qi}} W_i k \frac{\partial \phi_j}{\partial n} d\Gamma - \int_{\Omega^i_q} \nabla W_i \cdot k \nabla \phi_j d\Omega \right] u_j = \int_{\Omega^i_q} W_i f d\Omega - \int_{\Gamma^i_{qt}} W_i \bar{t} d\Gamma \tag{3.12}$$

ou, na forma matricial:

$$\sum_{j=1}^{n} K_{ij} u_j = F_i \Rightarrow \mathbf{Ku} = \mathbf{F}, \tag{3.13}$$

em que **K** é uma matriz $n \times n$ e **F** é um vetor $n \times 1$. O elemento (i, j) da matriz **K** é representado por K_{ij} e a i-ésima posição de **F** é representada por F_i . Então, K_{ij} e F_i são definidos como:

$$K_{ij} = \int_{\Gamma^i_{qu} \cup \Gamma^i_{qi}} W_i k \frac{\partial \phi_j}{\partial n} d\Gamma - \int_{\Omega^i_q} \nabla W_i \cdot k \nabla \phi_j d\Omega \tag{3.14}$$

$$F_i = \int_{\Omega^i_q} W_i f d\Omega - \int_{\Gamma^i_{qt}} W_i \bar{t} d\Gamma. \tag{3.15}$$

3.5 Imposição de condições de contorno

Se as funções de forma tiverem a propriedade do delta de Kronecker (como no RPIMp), a imposição da condição de contorno na fronteira de Dirichlet é direta, ou seja, os valores já conhecidos na fronteira Γ_u podem ser impostos diretamente no vetor **F** (Fonseca et al., 2010; Correa et al., 2015). Portanto, a ordem do sistema é igual ao número de nós desconhecidos, ou seja, número total de nós menos o número de nós em Γ_u e um novo termo é adicionado ao lado direito da Equação 3.15, resultando em:

$$F_i = \int_{\Omega^i_q} W_i f d\Omega - \int_{\Gamma^i_{qt}} W_i \bar{t} d\Gamma - \sum_{j=1}^{m} \left[\int_{\Gamma^i_{qu} \cup \Gamma^i_{qi}} W_i k \frac{\partial \phi_j}{\partial n} d\Gamma - \int_{\Omega^i_q} \nabla W_i \cdot k \nabla \phi_j d\Omega \right] g_j, \tag{3.16}$$

onde m é o número de nós em Γ_u e g_j é o valor conhecido de u no nó j em Γ_u.

No entanto, para funções de forma que não possuem a propriedade do delta de Kronecker, como as funções do MLS, o método das penalidades pode ser usados para forçar as condições de contorno (Liu, 2002). O método consiste em forçar o valor da solução na borda Dirichlet para os valores conhecidos, ou seja, $u = g$ em Γ_u. Um valor de penalidade α é escolhido e um termo de penalidade é adicionada à forma residual local (Equação 3.4), gerando:

$$\int_{\Omega^i_q} W_i \left[\nabla \cdot (k \nabla u) - f \right] d\Omega + \alpha \int_{\Gamma^i_{qu}} W_i (u - g) d\Gamma = 0. \tag{3.17}$$

Desenvolvendo as equações com o novo termo, os termos K_{ij} e F_i são definidos como:

$$K_{ij} = \int_{\Gamma^i_{qu} \cup \Gamma^i_{qi}} W_i k \frac{\partial \phi_j}{\partial n} d\Gamma - \int_{\Omega^i_q} \nabla W_i \cdot k \nabla \phi_j d\Omega + \alpha \int_{\Gamma^i_{qu}} W_i \phi_j d\Gamma \tag{3.18}$$

$$F_i = \int_{\Omega^i_q} W_i f d\Omega - \int_{\Gamma^i_{qt}} W_i \bar{t} d\Gamma + \alpha \int_{\Gamma^i_{qu}} W_i g d\Gamma. \tag{3.19}$$

Dessa forma, os nós de fronteira de Dirichlet fazem parte do sistema matricial, o que aumenta o custo computacional da solução. Além disso, o parâmetro α introduz valores altos na matriz, piorando o número de condição e afetando diretamente o solucionador usado posteriormente. Por esse motivo, o parâmetro α deve ser escolhido para minimizar esses efeitos, mas essa nem sempre é uma tarefa simples.

3.6 Funções de teste

Para funções de teste, independentemente da escolha das funções de forma, Atluri & Zhu (1998) apresenta seis sugestões que produzem as variações do MLPG (Tabela 3.1). Essas funções de teste MLPG têm a característica de zerar a uma certa distância, o que, associado ao suporte compacto das funções de forma, fornece a característica local do método.

O MLPG5 é particularmente interessante, e por isso escolhido para ser implementado neste trabalho, porque a função Heaviside é definida como sendo constante e igual a 1 dentro de Ω^i_q e 0 (zero) fora dele, ou seja:

$$W_i(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{x} \in \Omega^i_q \\ 0 & \text{if } \mathbf{x} \notin \Omega^i_q \end{cases}. \tag{3.20}$$

Escolhendo esta função de teste, as Equações 3.18 e 3.19 são simplificadas, pois $\nabla W = 0$. Dessa forma, usando o método de penalidade, são obtidas as Equações 3.21 e 3.22:

$$K_{ij} = \int_{\Gamma^i_{qu} \cup \Gamma^i_{qi}} k \frac{\partial \phi_j}{\partial n} d\Gamma + \alpha \int_{\Gamma^i_{qu}} \phi_j d\Gamma \tag{3.21}$$

$$F_i = \int_{\Omega^i_q} f d\Omega - \int_{\Gamma^i_{qt}} \bar{t} d\Gamma + \alpha \int_{\Gamma^i_{qu}} g d\Gamma, \tag{3.22}$$

Tabela 3.1: Métodos *Meshless Local Petrov-Galerkin* (MLPG) (Atluri & Zhu, 1998).

Métodos	**Função de teste em Ω_q^i**	**Integral para avaliar a forma fraca**
MLPG1	Função de peso MLS	Integral do domínio
MLPG2	Delta de Dirac $\delta(x, x_I)$	Nenhuma
MLPG3	Quadrados mínimos $\phi_{ii}^I(x)$	Integral do domínio
MLPG4	Solução fundamental u	Integral de fronteira singular
MLPG5	Função de Heaviside	Integral de fronteira regular
MLPG6	Igual à função de forma	Integral do domínio

e, quando a função de forma tiverem a propriedade delta de Kronecker, as Equações 3.23 e 3.24 são obtidas:

$$K_{ij} = \int_{\Gamma_{qu}^i \cup \Gamma_{qi}^i} k \frac{\partial \phi_j}{\partial n} d\Gamma \tag{3.23}$$

$$F_i = \int_{\Omega_q^i} f d\Omega - \int_{\Gamma_{qt}^i} \bar{t} d\Gamma - \sum_{j=1}^{m} \left[\int_{\Gamma_{qu}^i \cup \Gamma_{qi}^i} k \frac{\partial \phi_k}{\partial n} d\Gamma \right] g_k. \tag{3.24}$$

3.7 Montagem da matriz de rigidez e do vetor F

No MLPG, a matriz de rigidez é calculada nó por nó de forma independente, o que significa na prática que cada nó pode ser computado por uma *thread* (Figura 3.6). Todo o processo de montagem do sistema de equações pode ser feito em paralelo. O número de *threads* em execução será exatamente o número de subdomínios ou o número de avaliações de formas fracas locais.

A primeira tarefa é encontrar o domínio de suporte local do nó, feito através da varredura entre vizinhos do nó. Em sequência, as coordenadas dos nós no domínio de suporte são obtidas. Para avaliar as contribuições na matriz (Equações 3.21 e 3.22 ou Equações 3.23 e 3.24) a fronteira

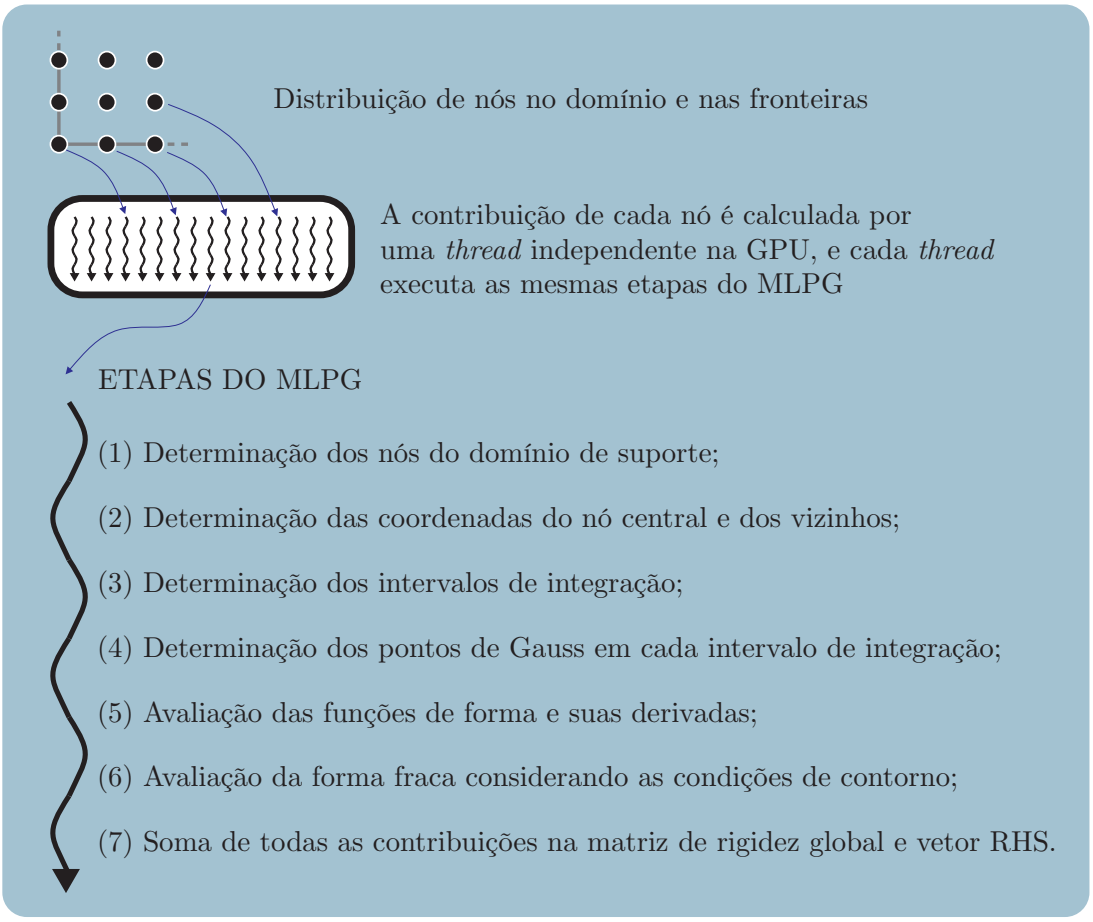


Figura 3.6: Montagem da matriz de rigidez através da avaliação da forma fraca local em cada subdomínio.

do subdomínio ($\Gamma^i_{qu} \cup \Gamma^i_{qi}$) é determinada. Uma integração numérica pelo método Gauss é realizada e, portanto, os pontos de integração devem ser determinados. Nesses pontos, os valores ϕ e $d\phi/dn$ são avaliados, ou seja, o valor da função de forma e sua derivada direcional normal nos pontos de integração são calculadas.

A forma fraca é então avaliada, de acordo com a função de forma adotada, e as integrações numéricas são realizadas. O MLPG possui uma particularidade que o torna interessante para a paralelização: cada contribuição de nó para a montagem do sistema ocorre em uma única linha da matriz de rigidez (Atluri & Zhu, 1998). Essa propriedade é particularmente importante para este trabalho e será explorada no estágio de solução do sistema linear resultante, pois todos os resultados da etapa são armazenados na memória local da *thread* e a consolidação subsequente dispensa operações atômicas. Outra informação importante é que os dados calculados até

então, no todo ou em parte, permanecerão inalterados durante toda a fase do solucionador devido às características dos métodos solucionadores utilizados.

No final do processamento das contribuições de um nó, a etapa da solução do sistema de equações com a respectiva linha adicionada poderia ser iniciada. No entanto, devido à arquitetura SIMT da GPU (*Single Instruction Multiple Thread*, Seção 2.1), o mais recomendado é aguardar o processamento dos n nós e, em seguida, começar a executar novas instruções. Esta ação não é um gargalo, pois os n nós estão sendo processados simultaneamente e tendem a terminar a etapa ao mesmo tempo.

Capítulo 4

Solver

Existem vários problemas de engenharia que são resolvidos a partir de análise linear (Franco, 2007; Jeffrey, 2002). Nestes casos, é necessária a solução de um sistema linear de equações simultâneas ($Ax = b$, $A \in \mathbb{R}^{n \times n}$ e $x, b \in \mathbb{R}^n$) por métodos especiais chamados de solucionadores, ou em inglês, *solvers*.

Este é o caso do sistema de equações lineares resultante do MLPG. No entanto, devido a matriz do sistema ser assimétrica e não positiva definida, temos um número reduzido de *solvers* que podem ser usados (Golub et al., 1996; Fletcher, 1976; Ghai et al., 2019). Além disso, o condicionamento da matriz pode ser ruim, pois aparecem elementos de valores mais altos do que a média dos restantes, especialmente ao usar o método de penalidade (Fonseca et al., 2010). Dadas estas propriedades do problema a ser resolvido, métodos iterativos baseados em subespaços de Krylov podem ser usados para aproximar a solução $\mathbf{u}$ do sistema $\mathbf{Ku} = \mathbf{F}$ (Equação 3.13, com $\mathbf{K} \in \mathbb{R}^{n \times n}$ e $\mathbf{u}, \mathbf{F} \in \mathbb{R}^n$), resultante do processo de discretização.

Ao longo de iterações dos *solvers* implementados, um resultado cada vez mais próximo do resultado exato é gerado,

$$x_i = x_0 + \mathcal{K}_i(r_0, A), \tag{4.1}$$

onde $i = [1, 2, \cdots]$ é o contador de iterações, x_0 é um valor inicial qualquer (geralmente um vetor de zeros), r_0 é o resíduo inicial dado por $r_0 = b - Ax_0$, e $\mathcal{K}_i(r_0, A)$ é o i-ésimo subespaço de Krylov em relação a r_0 e A (Saad, 2003).

4.1 Conceitos de algebra linear

Um sistema é dito linear quando é constituído por equações que possuem apenas uma única variável em cada termo, com este aparecendo na primeira potência. A solução para um sistema com n equações lineares e n variáveis, que chamaremos de sistema linear de ordem n, consiste em valores para as n variáveis tais que, quando substituídos, todas as equações são satisfeitas simultaneamente. A representação matricial do sistema é dada por:

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix},$$

ou simplesmente

$$Ax = b, \tag{4.2}$$

onde A é chamada de matriz dos coeficientes, x de vetor solução e b de vetor de termos independentes ou RHS (*right-hand side*). No entanto, sem investigação prévia não somos capazes de dizer se existe solução para um sistema linear arbitrário ou, uma vez existindo, se é de solução única. Quanto ao número de soluções, classificamos da seguinte forma (Golub et al., 1996; Trefethen & Bau, 1997; Demmel, 1997):

- **Sistema possível (ou consistente) determinado**, quando existe uma solução única;
- **Sistema possível (ou consistente) indeterminado**, quando existem mais de uma solução;
- **Sistema impossível (ou inconsistente)**, quando não existe solução.

Os *solvers* apresentados neste trabalho resolvem sistemas lineares de ordem n que tenham solução única, ou seja, $\det(A) \neq 0$.

4.1.1 Matrizes esparsas

Uma matriz esparsa é uma matriz essencialmente preenchida com elementos nulos (valor igual a zero), diferentemente da matriz densa, onde a maioria dos elementos são não nulos (Golub et al., 1996). Chamamos de dispersão ou densidade a fração entre elementos de valor zero e não-zero destas matrizes.

Quando tratamos de problemas da ciência ou engenharia que envolvem a resolução de equações diferenciais parciais, frequentemente aparecem matrizes de tamanhos expressivos, embora pouco densas. Ao armazenar e manipular tais matrizes em um computador é benéfico, e muitas vezes necessário, usar algoritmos especializados e estruturas de dados que tiram proveito desta esparsidade. As operações que utilizam estruturas e algoritmos de matriz densa são relativamente lentas e consomem grande quantidade de memória quando aplicadas a grandes matrizes esparsas.

No geral, dados esparsos são facilmente comprimíveis, e essa compressão quase sempre resulta em muito menos espaço gasto em memória para armazenamento dos dados. Dependendo da quantidade e distribuição das entradas não nulas, diferentes estruturas de dados podem ser usadas e obter significativa redução de consumo de memória quando comparado com o que seria necessário na abordagem não esparsa de armazenamento.

Formatos de armazenamento

Armazenar e manipular uma matriz esparsa em um computador é comum e, para isso, é necessário o uso de algoritmos e estruturas de dados que aproveitem sua esparsidade para reduzir a quantidade de memória usada e melhorar a velocidade de acesso (Golub et al., 1996; Moscovici et al., 2017).

Os formatos podem ser divididos em dois grupos: aqueles que suportam modificações eficientes e aqueles que suportam operações matriciais eficientes. O grupo de modificações eficientes inclui DOK (dicionário de chaves), LIL (lista de listas) e COO (lista de coordenadas) e é normalmente usado para a construção da matriz. Uma vez construída a matriz, ela é normalmente

convertida para outro formato como CSR (linha esparsa comprimida) ou CSC (coluna esparsa comprimida), que são mais eficientes para execução das operações matriciais.

Dicionário de Chaves (DOK) representa valores diferentes de zero como um dicionário mapeando tuplas `(linha, coluna)` para cada valor em uma tabela *hash* ou árvore de pesquisa binária, por exemplo. Este formato é bom quando a matriz esparsa é construída de forma incremental, e ruim para a iteração sobre valores diferentes de zero em uma ordem específica;

Lista de Listas (LIL) armazena uma lista encadeada por cada linha da matriz, onde cada entrada nesta lista armazena um índice de coluna e valor. Normalmente essas entradas são mantidas ordenadas por índice de coluna para possibilitar pesquisas mais rápidas. Este é um bom formato quando a construção da matriz é feita de forma incremental;

Lista de Coordenadas (COO) armazena uma lista encadeada de tuplas `(linha, coluna, valor)`. Talvez este seja o formato mais intuitivo, bom quando a construção da matriz é feita de forma incremental. Para melhorar os tempos de acesso aleatório, o ideal é que as entradas sejam classificadas por índices de linha ou de coluna;

Linha Esparsa Comprimida (CRS) é baseado no fato de que a informação de índice de linha é comprimida em relação ao formato COO. Ele é eficiente para realização de operações aritméticas, corte de linhas e produtos matriz-vetor.

No CRS os valores não nulos da matriz são armazenados um a um em um vetor (`valueArray`), do primeiro até a último, preservando a ordem. Um segundo vetor (`columIndexArray`) contém os índices das colunas onde cada valor não nulo apareceu, e um terceiro vetor (`rowPointerArray`) contém a posição no primeiro ou segundo vetor (`valueArray` ou `columIndexArray`) onde cada nova linha começa. Por exemplo, a matriz:

$$A = \begin{bmatrix} 1 & 0 & 0 & 3 & 0 \\ 5 & 7 & 0 & 0 & 2 \\ 3 & 0 & 2 & 4 & 0 \\ 0 & 0 & 6 & 9 & 0 \\ 0 & 0 & 0 & 0 & 4 \end{bmatrix}$$

é ser representada no formato CSR por:

$$\begin{aligned} \texttt{valueArray} &= [\,1\;3\;5\;7\;2\;3\;2\;4\;6\;9\;4\,] \\ \texttt{columIndexArray} &= [\,1\;4\;1\;2\;5\;1\;3\;4\;3\;4\;5\,] \\ \texttt{rowPointerArray} &= [\,1\;3\;6\;9\;11\;12\,] \end{aligned}$$

Enquanto o comprimento dos dois primeiros vetores é igual ao número de elementos não-nulos da matriz, o comprimento do terceiro vetor é igual ao número de linhas da matriz mais um. Dentro de cada linha, os elementos podem ainda ser armazenados em ordem arbitrária, o que pode ser muito conveniente;

Coluna Esparsa Comprimida (CCS) é semelhante ao CSR, mas substituindo-se a compressão originalmente feita no vetor de índices de linhas por uma compressão semelhante no vetor de índices de colunas. Este formato é eficiente para operações aritméticas, corte de colunas e produtos matriz-vetor. Este é o formato padrão para especificar uma matriz esparsa no MATLAB.

Outros esquemas podem ser utilizados aproveitando da esparsidade da matriz. Por exemplo, em matrizes diagonais, as diagonais não-nulas podem ser armazenadas separadamente, e em matrizes simétricas é necessário armazenar apenas os elementos da diagonal principal e da parte triangular superior (ou inferior) da matriz.

Apesar de ser o formato menos compactado, o uso de `COO` é justificado por permitir acesso aleatório às posições resultantes da matriz. Esse é um requisito essencial para a composição de valores realizada pelo MLPG, por exemplo. Considerando que cada nó da nuvem, processado em paralelo, gera contribuições para uma única linha da matriz, as inserções acontecem em paralelo, sem qualquer condição de corrida. Após esse estágio, é possível ainda alguma compactação, que é realizada sobre o vetor que armazena as coordenadas da linha por meio da função `cusparseXcoo2csr()` da biblioteca cuSPARSE (Naumov et al., 2010). Por ser uma manipulação de apenas um vetor, realizada apenas uma vez, o custo desta conversão é justificado.

4.1.2 Condicionamento numérico

A princípio, consideramos dois aspectos para solução de sistemas lineares: se existe uma solução e qual o melhor método a ser escolhido. Há, porém, um outro aspecto importante: se a solução é muito sensível a pequenas mudanças na matriz dos coeficientes ou no vetor de termos independentes (RHS). Quando isto acontece, dizemos que a matriz dos coeficientes é mal condicionada. Por outro lado, um sistema bem condicionado é aquele que uma pequena alteração na entrada gera uma também pequena alteração na saída.

Considere o exemplo dado por

$$\begin{bmatrix} 1 & 2 \\ 2 & 3.999 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 4 \\ 7.999 \end{bmatrix},$$

temos como solução $x = (2, 1)^T$ com resíduo $r = 0$. Realizando uma pequena perturbação no vetor de termos independentes, dada como

$$\begin{bmatrix} 1 & 2 \\ 2 & 3.999 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 4.001 \\ 7.998 \end{bmatrix},$$

temos o resíduo $r = (10^{-3}, -10^{-3})^T$, que poderia, dependendo do problema tratado, ser considerado razoavelmente baixo, o que é evidência de solução com boa aproximação. No entanto, usando um método exato obtemos $x = (-3.999, 4)^T$, bem diferente da primeira solução.

Portanto, a menos que o erro máximo tolerado seja menor que 10^{-3}, não seria questionada a solução dada para o sistema linear.

Número de Condição

O número de condição é um valor atribuído à uma matriz quadrada A qualquer, referente ao seu condicionamento. É dado na forma

$$cond(A) = \|A\| \cdot \|A^{-1}\|.$$

Quanto maior o número de condição, mais mal condicionada é a matriz A, e mais difícil é sua convergência para a solução. Por outro lado, quanto mais próximo de 1, mais bem condicionada é a matriz.

4.1.3 Operações de algebra linear

As operações algébricas essenciais do *solver* têm uma implementação otimizada para uso com CUDA utilizando as bibliotecas cuBLAS (Nvidia, 2016) e cuSPARSE (Naumov et al., 2010), esta última especializada em operações com matrizes esparsas. São operações triviais de adição, subtração, multiplicação e divisão, que são executadas simultaneamente em todas as posições $i = [1, \cdots, n]$ das estruturas de dados de entrada.

Atualizações de vetores ou operações entre vetores são perfeitamente paralelizáveis, e os produtos de vetores por matrizes podem ser implementados com comunicação apenas entre processadores próximos. No entanto, algumas operações como produtos interno, por exemplo, precisam consolidar dados de várias unidades de processamento (Gallopoulos et al., 2015; Kasmi et al., 2017). Desta forma, elas não podem ser implementadas eficientemente por operações paralelas devido a necessidade de comunicação global. Na prática, a operação exige sincronização e gera um gargalo no processo da solução, o que precisa ser evitado.

A Figura 4.1 mostra a parte do esforço computacional empregado em cada uma das operações mais críticas em uma única iteração de cada *solver*. São elas:

produto interno, que calcula o produto de dois vetores, x e y, segundo a operação $\sum_{i=1}^{n} (x[i] \times y[i])$. Para esta tarefa é utilizada a função `cublas<t>dot()`, disponível na cuBLAS. Esta é a operação que gera o maior gargalo em ambientes paralelos, pois precisa de consolidação de dados e, consequentemente, implica em sincronização;

axpy, que multiplica o vetor x por um escalar α, e o adiciona ao vetor y, substituindo valores originais de y pelo resultado da operação em cada posição. Portanto, a operação executada é $y[i] = \alpha \times x[i] + y[i]$. A função `cublas<t>axpy()`, disponível na cuBLAS, é usada. Ela é capaz de executar a operação de multiplicação de vetores por uma constante ao mesmo tempo em que realiza a soma dela com outro vetor. A ausência de

uma das duas ações não implica em uma redução ou incremento no custo computacional total, mas o uso conjunto é bastante comum;

scal, que multiplica um vetor y por um escalar α, segundo a operação $y[i] = \alpha \times y[i]$. A função `cublas<t>scal()`, também disponível na cuBLAS, é usada;

Ax, que executa a operação de multiplicação de uma matriz por um vetor, segundo a operação $y = \alpha \times op(A) \times x + \beta \times y$. Geralmente é a operação com o maior custo computacional (Figura 4.1) e, devido à natureza esparsa da matriz de entrada, a função possui sua própria estrutura de dados otimizada. A função `cusparse<t>csrmv()`, disponível na cuSPARSE, é usada. Ela permite o uso da matriz A, tanto em sua forma original quanto em sua transposição, A^T;

norma euclidiana, que calcula a norma euclidiana do vetor x e é usada para verificar a convergência dos *solvers*. O resultado é equivalente a $\sqrt{\sum_{i=1}^{n} x[i] \times x[i]}$. A função `cublas<t>nrm2()`, disponível na cuBLAS, é usada;

redução, que possui uma implementação computacionalmente otimizada da operação $\sum_{i=1}^{n} x[i]$, que soma todos os elementos do vetor para ter, ao final da operação, um valor único consolidado.

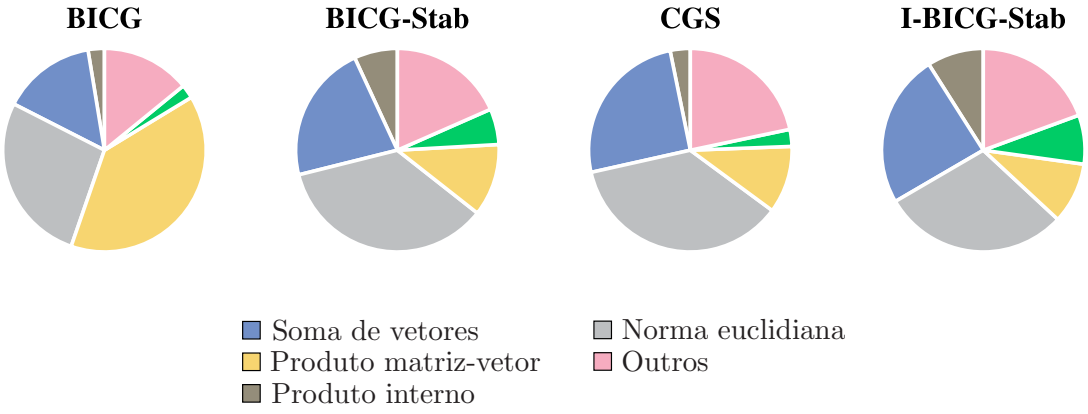


Figura 4.1: Distribuição do custo computacional das operações mais importantes de álgebra linear em cada *solver* implementado neste trabalho. Informações adquiridas da análise de *log* do aplicativo nvproof®, fornecido pela NVIDIA.

As funções das bibliotecas mencionadas são executadas de forma assíncrona com o *host*, e podem retornar o controle para o aplicativo no *host* antes que o resultado da operação solicitada tenha sido entregue. Se for necessário garantir que o resultado da função esteja

corretamente gravado no espaço de memória dedicado a ele, a instrução de sincronização deve ser executada explicitamente (`cudaDeviceSynchronize()`) (Naumov, 2011). Isto é necessário mesmo que a função retorne imediatamente, porque as variáveis que registram os resultados são passadas por referência, o que significa que seus valores podem continuar sendo alterados mesmo após o retorno, até que ocorra uma sincronização. Esse recurso permite que as funções da biblioteca sejam executadas de forma assíncrona usando *streams*, otimizando o uso do dispositivo enquanto operações distintas são chamadas e processadas.

4.2 Solução de sistemas lineares

Os *solvers* são métodos que calculam os valores de todos os elementos do vetor x (Equação 4.2), e podem ser subdivididos em dois grupos (Golub et al., 1996; Trefethen & Bau, 1997; Demmel, 1997):

Métodos exatos (ou diretos), capazes de encontrar resultado exato para x, resguardados erros de arredondamento, através de um número pré-determinado de operações;

Métodos iterativos, capazes de encontrar resultado aproximado para x a partir de uma precisão definida, através de um processo infinito convergente.

Métodos exatos, a princípio, produziriam resultados exatos salvo a imprecisão causada pelo número finito de dígitos utilizados para representação das variáveis. No entanto, como as operações realizadas são dependentes umas das outras, o erro de arredondamento ou de representação é propagado e acumulado. Isto é um problema sobretudo para sistemas de grande porte (valores elevados de n), que pode gerar um resultado sem significado. Já os métodos iterativos possuem erro de arredondamento não cumulativo e, portanto, tendem a gerar resultados melhores nestes casos. Ambos têm vantagens e desvantagens e devem ser escolhidos dependendo das características do sistema linear a ser resolvido.

Dentre os métodos exatos podemos citar o Método de Eliminação de Gauss, Decomposição LU, e Método de Cholesky (Watkins, 2010; Golub et al., 1996; Jeffrey, 2002; Franco, 2007).

Entretanto, como o foco deste trabalho é a solução de sistemas lineares gerados pelo MLPG, esparsos e de grande porte, abordaremos apenas os métodos iterativos.

4.2.1 Métodos iterativos

Os métodos iterativos foram inicialmente propostos na década de 50 como uma alternativa aos denominados métodos exatos (ou diretos) para solução de sistemas esparsos. Tais métodos apresentam melhores resultados nestes casos (Golub et al., 1996; Franco, 2007), inclusive com bons resultados na solução do sistema de equações com as características observadas da matriz de rigidez gerada pelo MLPG. Eles são indicados por:

- Apresentar uma estrutura eficiente de armazenamento em memória, armazenando somente elementos diferentes de zero (pequena parte do total de elementos);
- Auto corrigirem-se caso um erro seja cometido, e também por reduzirem os erros de arredondamento que eventualmente aparecem em métodos exatos;
- Não alterarem a estrutura original da matriz de coeficientes, A. Se fosse aplicado um método direto, por exemplo Decomposição LU, ocorreriam preenchimentos na matriz introduzindo valores onde originalmente era nulo.

Estes métodos são chamados **iterativos** por fornecerem uma sequência de aproximantes da solução, cada uma delas obtida a partir das anteriores através de um mesmo conjunto de operações. O método dito **estacionário** quando esse conjunto é sempre o mesmo, e **convergente** quando leva a um resultado, dado o erro máximo aceitável, com um número finito de iterações. Ou seja, dada uma sequência $x_k \in E$ e uma norma sobre E (E é um espaço vetorial), dizemos que a sequência x_k é convergente se:

$$\|x_k - \hat{x}\| \to 0, \text{ quando } k \to \infty.$$

4.2.2 Critério de parada

Para resolver um sistema linear $Ax = b$ através de um método iterativo qualquer, escolhemos arbitrariamente x_0 como aproximação inicial e, por meio das iterações do método escolhido, refinamos a solução até que o erro (resíduo) atenda à precisão solicitada ou atinja um dado número máximo de iterações. O teste do erro é dado por

$$\frac{\|x_{k+1} - x_k\|_\infty}{\|x_{k+1}\|_\infty} < \epsilon,$$

onde ϵ é a precisão pré-fixada e x_k e x_{k+1} são duas aproximações consecutivas.

4.2.3 Etapa de inicialização

As estruturas de suporte dos *solvers* são criadas e inicializadas na etapa de inicialização. No MLPG, apenas duas das estruturas já existem: a matriz de rigidez, A, e o vetor RHS, b. O vetor r é criado e recebe os valores residuais iniciais dados por $r_0 = b - Ax_0$. O vetor da solução, x, também deve ser criado, geralmente inicializado com zero e atualizado ao longo das iterações. Outras estruturas são exclusivas do *solver* escolhido, criadas e inicializadas no espaço de memória do dispositivo antes do início do processo iterativo, sem qualquer participação do *host*.

Outra tarefa crítica, que ocorre na inicialização, é a otimização da matriz esparsa A. Esta estrutura é criada no formato `COO` e, posteriormente, convertida para o formato `CSR`. Essa conversão é recomendada apenas após a conclusão de sua criação, pois seria caro manter esta otimização no estágio de composição da matriz. Essa é uma tarefa crítica aplicada apenas no vetor que identifica os índices de linha da matriz (a matriz esparsa é representada por três vetores, contendo a **linha**, **coluna** e **valor** de cada posição diferente de zero em A), possui baixo custo computacional e é justificável porque contribui para reduzir o custo de operações recorrentes.

4.2.4 Precondicionador

Foram realizados testes com dois pré-condicionadores na tentativa de acelerar a convergência do *solver*. Inicialmente com a fatoração LU-incompleta, através da função `csrilu0()` disponível na biblioteca cuSparse. Entretanto, verificou-se que esse pré-condicionador foi sensível ao tamanho da matriz, independentemente do seu número de condição, o que o torna impraticável para grandes sistemas de equações.

Outra estratégia foi com o pré-condicionador diagonal (pré-condicionador de Jacobi (Saad, 2003; Mehmood & Crowcroft, 2005; Khan & Topping, 1996)). Essa técnica de pré-condicionamento é fácil de implementar e paralelizar, tornando-a atraente para uso em GPUs (Liu et al., 2007; Kiss et al., 2012, 2013; Anzt et al., 2017). Mas, devido a ausência de dominância diagonal e aos altos valores para os números de condição da matriz de rigidez do MLPG-MLS e ainda maiores do MLPG-RPIMp, o uso deste pré-condicionador também provou ser ineficiente.

O impacto do uso dos pre-condicionadores testados é insignificante ou negativo na etapa do *solver*, conforme mostrado na Figura 4.2. Para compor o gráfico, o *solver* BICG-Stab foi escolhido arbitrariamente, mas o mesmo impacto é também observado nos outros.

Com base nesses resultados preliminares, nenhum pré-condicionador é usado neste trabalho, porque:

1. A convergência sem pré-condicionador é satisfatória para os casos de teste quando comparada com a convergência usando um dos pré-condicionadores;
2. A solução sem pré-condicionador mostrou ter sucesso em maior amplitude de casos de testes, convergindo para uma solução quando a solução pré-condicionada não convergiu;
3. A estrutura de dados permanece no mesmo formato e local computado pelo MLPG;
4. A solução sem pré-condicionamento permite que, como proposto inicialmente, o método iterativo seja processado imediatamente após a montagem da matriz de rigidez. Não há necessidade de um ponto de sincronização adicional ou qualquer movimentação de dados.

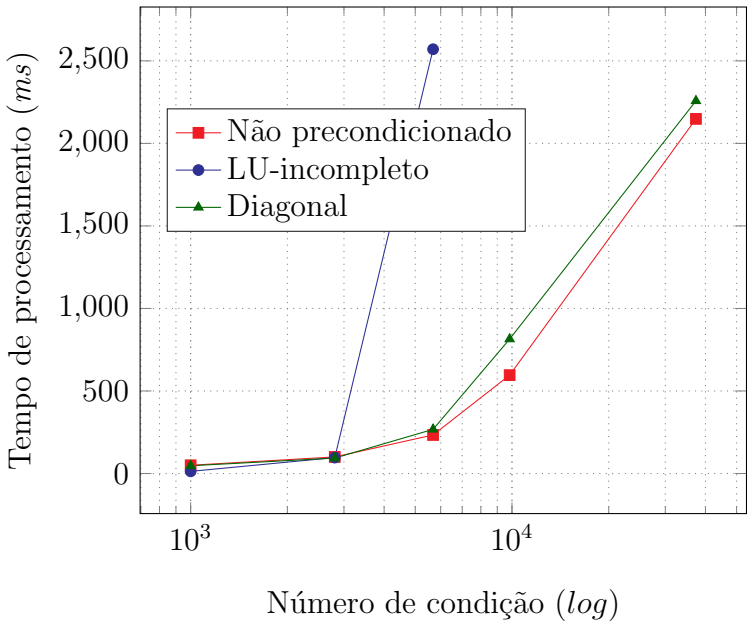


Figura 4.2: Avaliação de impacto do uso de pré-condicionador no tempo de processamento necessário para a convergência do BICG-Stab com a evolução do número da condição.

4.3 Métodos iterativos não-estacionários

Qualquer computação de um método baseado em subespaço de Krylov é dividida entre as operações de álgebra listadas na Seção 4.1.3. Normalmente elas são adequadas para para processamento paralelo com memória compartilhada (Dongarra et al., 1990; Anzt et al., 2017; Cheik Ahamed & Magoulès, 2017). Porém, para ambientes de memória compartilhada paralela, como nas GPUs, na qual a matriz e os vetores são distribuídos pelos processadores, algoritmos específicos devem garantir a eficiência. A comunicação global deve ser adiada e, se possível, evitada. Um exemplo disso é o cálculo do produto interno, que possui uma etapa de consolidação de dados que compõem a resposta.

As outras operações podem ser implementadas sem comunicação entre os processadores ou, quando uma matriz esparsa está envolvida, apenas a comunicação entre processadores próximos. O custo da comunicação se torna mais crítico quando o número de processadores paralelos ou o tamanho do sistema de equações lineares é aumentado. Isso pode acontecer porque o paralelismo implementado afeta a escalabilidade do algoritmo de maneira negativa (Yang & Brent, 2002a; Ghai et al., 2019).

A decisão sobre um *solver* é uma tarefa complexa, especialmente quando se espera que a convergência da solução seja independente do sistema de equações a ser resolvido. Isto é difícil de ocorrer, porque o rearranjo mínimo da nuvem de nós tem como resultado um sistema de equações lineares diferente, cujo processo de convergência apresenta outro comportamento, mesmo que o problema físico e o número total de nós permaneçam iguais. Ao mesmo tempo, para manter a solução inteiramente na GPU, a independência da CPU é a primeira propriedade a ser considerada para cada operação de álgebra linear executada, e também para o conjunto de operações que compreende a iteração.

Por esse motivo, o fator mais crítico para a análise é a necessidade de pontos de sincronização na iteração, preocupação do algoritmo I-BICG-Stab (Yang & Brent, 2002a; Ghai et al., 2019) escolhido para ser um dos *solvers* testados neste trabalho .

A Figura 4.3 mostra o número de pontos de sincronização para cada *solver* escolhido para ser usado neste trabalho, bem como o número de estruturas auxiliares necessárias. Quanto menos pontos de sincronização, mais adequado o *solver* tende a ser para a arquitetura de memória distribuída, e quanto menos estruturas auxiliares, menos memória extra é necessária no dispositivo.

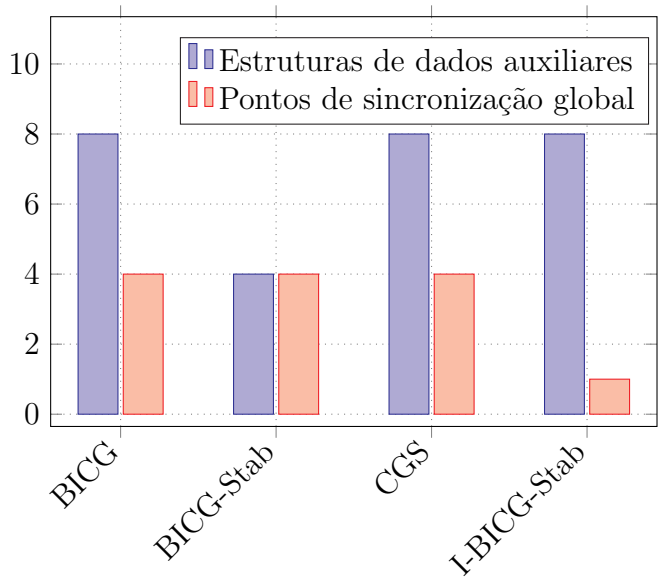


Figura 4.3: Comparativo dos *solvers* testados considerando a quantidade de vetores auxiliares criados para o processo iterativo e a quantidade de pontos de sincronização globais dentro da iteração.

4.3.1 Gradientes Conjugados (CG)

Dado um sistema linear $Ax = b$ onde $A_{n\times n}$ é positiva definida e r_n o vetor resíduo tal que $r = b - Ax$, o objetivo de um processo de relaxação (grupo do qual faz parte o Método dos Gradientes Conjugados abordado aqui) é fazer com que o resíduo se anule. Para verificar como é possível conseguir isto, vamos substituir o problema de encontrar a solução para o sistema linear $Ax = b$ pelo problema equivalente de encontrar um minimizador da função quadrática (Golub et al., 1996)

$$f(x) = \frac{1}{2}x^T Ax - b^T x, \tag{4.3}$$

onde $b, x \in \mathbb{R}^n$ e $A \in \mathbb{R}^{n\times n}$. Considerando, por exemplo

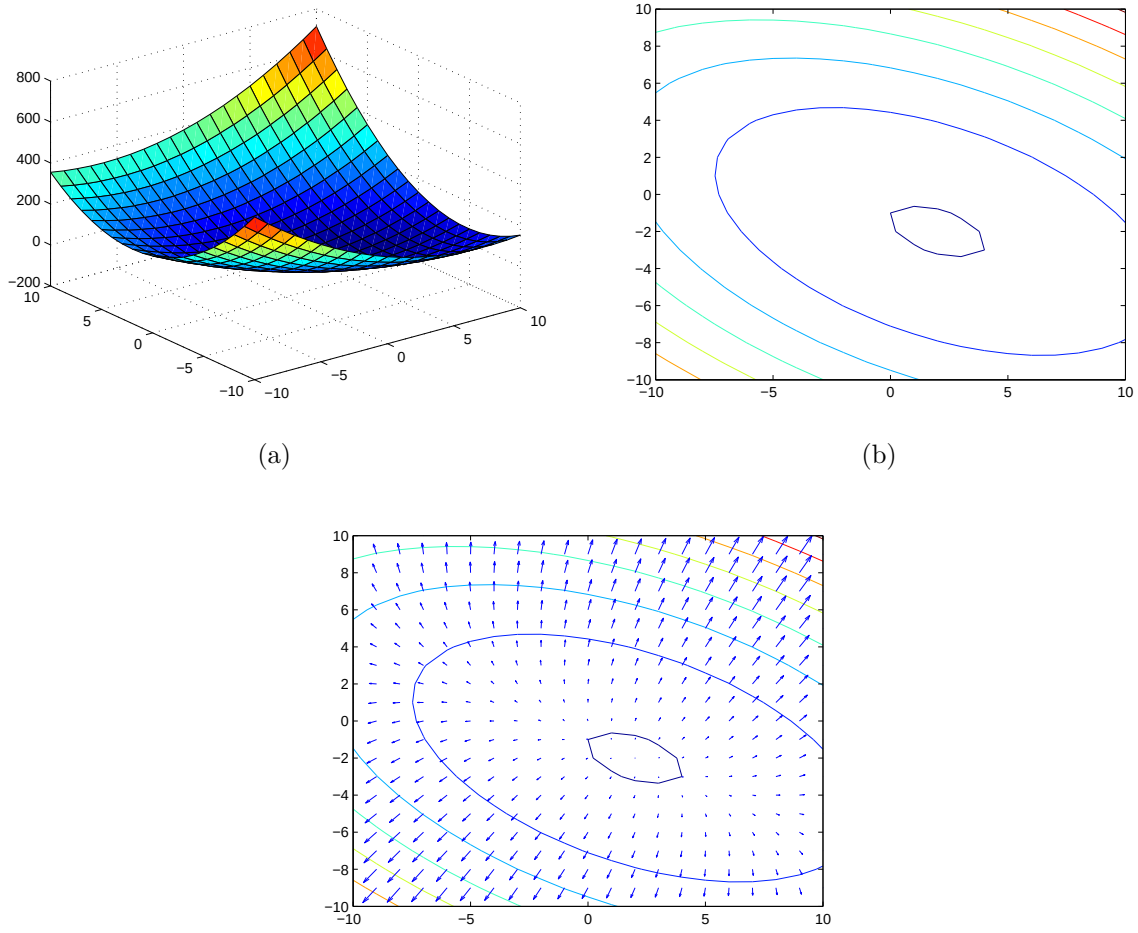
$$A = \begin{bmatrix} 3 & 2 \\ 2 & 6 \end{bmatrix} \quad b = \begin{bmatrix} 2 \\ -8 \end{bmatrix}, \tag{4.4}$$

temos o gráfico da Equação 4.3 apresentado na Figura 4.4. Supondo que A é uma matriz positiva definida, o ponto x tal que $Ax = b$ é ponto de mínimo de $f(x)$. Dessa forma, se encontrarmos o ponto de mínimo, que para o exemplo é o ponto $(2, -2)$, encontramos também a solução do sistema linear $Ax = b$.

Sabemos que o gradiente (∇) da função $f(x)$ em um dado ponto x é ortogonal à curva de nível (Figura 4.4c) e aponta a direção de maior crescimento da função. Fazendo $\nabla f(x) = 0$ temos o ponto crítico, ou seja, o ponto x tal que $Ax = b$. Temos então

$$\begin{aligned}
\nabla f(x) &= \left[\frac{\partial}{\partial x_1}f(x), \frac{\partial}{\partial x_2}f(x), \cdots, \frac{\partial}{\partial x_n}f(x)\right]^T \\
&= \begin{bmatrix} a_{11}x_1 & + & a_{12}x_2 & + & \cdots & + & a_{1n}x_n & - & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + & \cdots & + & a_{2n}x_n & - & b_2 \\ \vdots & & \vdots & & \vdots & & \vdots & & \vdots \\ a_{n1}x_1 & + & a_{n2}x_2 & + & \cdots & + & a_{nn}x_n & - & b_n \end{bmatrix} \\
&= Ax - b.
\end{aligned}$$

Dada uma aproximação inicial x_0 para a solução do sistema linear $Ax = b$, seria natural pensarmos em tomar a direção oposta a $\nabla f(x)$ para a correção de x_0 , pois $-\nabla f(x)$ aponta



(a) (b)

(c)

Figura 4.4: Representação gráfica (a), curvas de nível (b) e gradiente (c) da Equação 4.3 considerando os valores expressos na Equação 4.4.

na direção que $f(x)$ decresce mais rapidamente, e o ponto de mínimo é a solução do sistema. Logo,

$$-\nabla f(x_i) = b - Ax_i = r_i$$

nos diz a direção, restando descobrir o quanto caminhar. Sabemos que $x_i = x_{i-1} + \alpha r_{i-1}$ e que α minimiza $f(x_i)$ quando a derivada direcional é nula,

$$\frac{\partial f(x_i)}{\partial \alpha} = 0,$$

mas

$$\begin{aligned} \frac{\partial f(x_i)}{\partial \alpha} &= (f'(x_i))^T \frac{\partial x_i}{\partial \alpha} \\ &= (f'(x_i))^T r_{i-1}. \end{aligned} \tag{4.5}$$

Igualar a Equação 4.5 a zero sugere que α é escolhida de forma que r_{i-1} e $f'(x_i)$ sejam ortogonais, nos permitindo usar $f'(x_i) = -r_{i-1}$ para calcular o valor de α. Sendo assim, temos

$$(r_i)^T r_{i-1} = 0. \tag{4.6}$$

Substituindo r_i, temos

$$(b - Ax_i)^T r_{i-1} = 0.$$

Como $x_i = x_{i-1} + \alpha r_{i-1}$, temos:

$$\begin{aligned}
(b - A(x_{i-1} + \alpha r_{i-1}))^T r_{i-1} &= 0 \\
(b - Ax_{i-1})^T r_{i-1} - \alpha (Ax_{i-1})^T r_{i-1} &= 0 \\
(b - Ax_{i-1})^T r_{i-1} &= \alpha (Ax_{i-1})^T r_{i-1} \\
(r_{i-1})^T r_{i-1} &= \alpha (r_{i-1})^T A r_{i-1}.
\end{aligned}$$

Logo,

$$\alpha = \frac{(r_{i-1})^T r_{i-1}}{(r_{i-1})^T A r_{i-1}}. \tag{4.7}$$

Sendo a aproximação inicial x_0 um valor arbitrário (comumente fazendo $x_0 = 0$), chamamos de Método dos Gradientes (ou Método da Máxima Descida) o processo iterativo que diminui o resíduo $r_i = b - Ax_i$, dando um passo na direção de $-\nabla f(x_{i-1})$ de tamanho α_i (Equação 4.7). Para isso, corrigimos x dando um passo na direção α do tamanho do resíduo r, ou seja,

$$x_i = x_{i-1} + \alpha_i r_{i-1}. \tag{4.8}$$

O pseudocódigo do Método dos Gradientes é dado segundo o Algoritmo 1.

Dessa forma, construímos uma sequência $\{x_i\}$ que converge para a solução do sistema linear $Ax = b$, pois estamos considerando que o resíduo diminua em cada passo do processo iterativo, ou seja, $\|r_i\| < \|r_{i-1}\|$. Quando $i \to \infty$, temos $r_i \to 0$.

Algoritmo 1: Método dos Gradientes

Input : A, b
Output: x

Faça x_0 a partir de algum palpite inicial
for $i = 1, 2, \ldots$ **do**
    $r_{i-1} = b - Ax_{i-1}$
    $\alpha_i = \left((r_{i-1})^T r_{i-1}\right) / \left((r_{i-1})^T A r_{i-1}\right)$
    $x_i = x_{i-1} + \alpha_i r_{i-1}$
    if x_i é preciso o suficiente **then (PARE)**;
end for

É possível, no entanto, que uma direção que está sendo adotada na iteração i tenha sido usada em iterações anteriores, apresentando um comportamento de *zigzag* como mostrado na Figura 4.5. Este comportamento tende a tornar a velocidade de convergência deste método muito lenta.

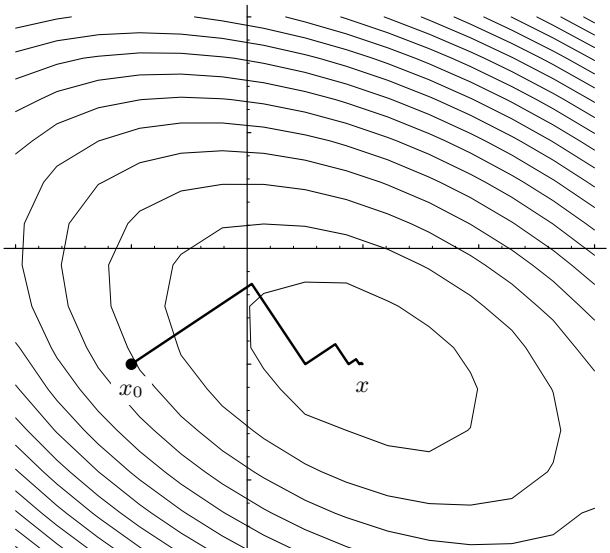


Figura 4.5: Evolução da precisão do cálculo de x ao longo das iterações no Método dos Gradientes.

Para evitar que se tome várias vezes uma mesma direção α para a correção de x, o Método dos Gradientes Conjugados propõe uma modificação no Método dos Gradientes: dada uma aproximação inicial x_0 para o sistema linear $Ax = b$, serão consideradas um conjunto de direções conjugadas $p_0, p_1, \ldots, p_{n-1}$ de forma que, em até n iterações, teremos uma aproximação

satisfatória para a solução do sistema. A fórmula recursiva é semelhante ao Método dos Gradientes (Equação 4.8) mas, ao invés do vetor resíduo, utiliza direções conjugadas dos gradientes para calcular direção e tamanho do passo.

Definição: duas direções x e y são ditas conjugadas se $x^T Ay = y^T Ax = 0$.

Na primeira iteração, o valor de α é dado como na Equação 4.7 e a direção é dada utilizando o resíduo, $r_0 = b - Ax_0$, da mesma forma que no Método dos Gradientes (Equação 4.8). Para as iterações $i > 0$, definimos como direção do passo, p_i, uma direção que seja conjugada à anterior, p_{i-1}, na forma

$$p_i = -r_{i-1} + \alpha p_{i-1}. \tag{4.9}$$

Para definir o valor de α considerando a definição de direção conjugada, fazemos

$$\begin{aligned} (p_i)^T A p_{i-1} &= 0 \\ (-r_{i-1} + \alpha p_{i-1})^T A p_{i-1} &= 0 \\ (-r_{i-1})^T A p_{i-1} + \alpha (p_{i-1})^T A p_{i-1} &= 0, \end{aligned}$$

ou seja,

$$\alpha = \frac{(r_{i-1})^T A p_{i-1})}{(p_{i-1})^T A p_{i-1}}. \tag{4.10}$$

Calculada a direção p_i (Equação 4.9), precisamos definir o tamanho do passo, q_i, definido como minimizador da função quadrática representada pela Equação 4.3, ou seja,

$$q_i = -\frac{(r_{i-1})^T p_i}{(Ap_i)^T p_i}. \tag{4.11}$$

Note que, para garantir a convergência da solução, α (Equação 4.10) e q_i (Equação 4.11) sempre são maiores que zero, e, consequentemente, $r_i \to 0$. O cálculo do resíduo, r, é então dado por

$$
\begin{aligned}
r_i &= Ax_i - b \\
&= A(x_{i-1} + q_i p_i) - b \\
&= Ax_{i-1} - b + q_i A p_i \\
&= r_{i-1} + q_i A p_i
\end{aligned}
$$

O pseudocódigo do Método dos Gradientes Conjugados é dado pelo Algoritmo 2.

Algoritmo 2: Método dos Gradientes Conjugados — CG

Input : A, b
Output: x

Faça x_0 a partir de algum palpite inicial
$r_0 = Ax_0 - b$
$p_1 = -r_0$
$q_1 = \left((r_0)^T r_0\right) / \left((Ar_0)^T r_0\right)$
$x_1 = x_0 + q_1 p_1$
$r_1 = r_0 + q_1 A p_1$
for $i = 2, 3, \ldots$ **do**
    $\alpha = \left((r_{i-1})^T r_{i-1}\right) / \left((r^{(i-2)})^T r^{(i-2)}\right)$
    $p_i = -r_{i-1} + \alpha p_{i-1}$
    $q_i = \left((r_{i-1})^T r_{i-1}\right) / \left((Ap_i)^T p_i\right)$
    $x_i = x_{i-1} + q_i p_i$
    if x_i é preciso o suficiente **then (PARE)**;
    $r_i = r_{i-1} + q_i A p_i$
end for

4.3.2 Gradientes BiConjugados (BICG)

O Método dos Gradientes Conjugados (Seção 4.3.1) não é adequado para sistemas lineares onde a matriz dos coeficientes, A, não seja simétrica, pois não é possível obter sequências ortogonais dos vetores dos resíduos com poucas iterações (Purcina & Saramago, 2007; Barrett et al., 1987).

Diante deste problema foi desenvolvido o Método dos Gradiente Bi-Conjugados (BICG), no qual a sequência de resíduos ortogonais é substituída por duas sequências mutuamente ortogonais, porém, com a desvantagem de não mais garantir a minimização do resíduo. Este é um ponto muito importante, pois conforme mostrado nos Método dos Gradientes Conjugados, desconsiderando o erro de arredondamento, estes métodos garantem que, se a matriz dos coeficientes é simétrica e positivo-definida, então a norma do vetor resíduo tende a zero. O BICG e suas derivações não se baseiam na teoria de otimização, e sim na teoria de Lanczos. Portanto, a minimização do resíduo não é garantida (Purcina & Saramago, 2007).

As fórmulas para atualização dos resíduos no BICG são semelhantes às dos Método dos Gradientes Conjugados, mas baseadas em A^T ao invés de A. As duas fórmulas de atualização das sequências dos resíduos e as direções de busca são

$$\begin{aligned} r_i &= r_{i-1} - \alpha_i A p_i \\ \tilde{r}_i &= \tilde{r}_{i-1} - \alpha_i A^T \tilde{p}_i, \end{aligned}$$

sendo que:

$$\begin{aligned} p_i &= r_{i-1} + \beta_{i-1} p_{i-1} \\ \tilde{p}_i &= \tilde{r}_{i-1} + \beta_{i-1} \tilde{p}_{i-1}. \end{aligned}$$

Os valores de α e β escolhidos como

$$\alpha_i = \frac{\tilde{r}^{(i-1)^T} r_{i-1}}{\tilde{p}^{(i)^T} A p_i} \quad \text{e} \quad \beta_i = \frac{\tilde{r}^{(i)^T} r_i}{\tilde{r}^{(i-1)^T} r_{i-1}},$$

garantem a relação de bi-ortogonalidade $\tilde{r}^{(i)^T} r^{(j)} = \tilde{p}^{(i)^T} A p^{(j)} = 0$, com $i \neq j$. O pseudocódigo do BICG sugerida por Barrett et al. (1987) é dado segundo o Algoritmo 3.

Para fins de exemplo da implementação realizada neste trabalho, o Código Fonte 4.1 apresenta a codificação na linguagem CUDA do Algoritmo 3. Nele, são realizadas chamadas para funções que:

- Aloca memória para um vetor (Código Fonte 4.2);
- Inicializa um vetor com zeros (Código Fonte 4.3);

Algoritmo 3: Método dos Gradientes BiConjugados Clássico — BICG

Input : A, b
Output: x

$\tilde{r}_0 = r_0 = b - Ax_0$
for $i = 1, 2, \ldots$ **do**
  $\rho_i = r_{i-1}^T \tilde{r}_{i-1}$
  if $\rho_i = 0$ **then (FALHOU)**;
  if $i = 1$ **then**
    $p_i = r_{i-1}$
    $\tilde{p}_i = \tilde{r}_{i-1}$
  else
    $\beta = \rho_i / \rho_{i-1}$
    $p_i = r_{i-1} + \beta p_{i-1}$
    $\tilde{p}_i = \tilde{r}_{i-1} + \beta \tilde{p}_{i-1}$
  end if
  $q_i = Ap_i$
  $\tilde{q}_i = A^T \tilde{p}_i$
  $\alpha = \rho_i / (\tilde{p}_i^T q_i)$
  $x_i = x_{i-1} + \alpha p_i$
  if x_i é preciso o suficiente **then (PARE)**;
  $r_i = r_{i-1} - \alpha q_i$
  $\tilde{r}_i = \tilde{r}_{i-1} - \alpha \tilde{q}_i$
end for

- Copia os dados de um vetor para outro (Código Fonte 4.4);
- Libera a memória alocada para um vetor (Código Fonte 4.5);
- Calcula a norma (Código Fonte 4.6);
- Sincroniza *threads* no dispositivo (Código Fonte 4.7);
- Multiplica dois vetores (Código Fonte 4.8);
- Soma dois vetores (Código Fonte 4.9);
- Multiplica um vetor por uma matriz (Código Fonte 4.10).

```
1
2   int bicg_classic() {
3
4     //Cria e inicializa o vetor solução
5     double *x = 0;
6     zera(x);
7
8     //Linha 1
9     double *r = 0;
10    double *r_tilde = 0;
11    cria(&r);
12    cria(&r_tilde);
13    copia(r, rhs);
14    copia(r_tilde, rhs);
15
16    //Criação e inicialização dos vetores auxiliares
17    double *p = 0;
18    double *p_tilde = 0;
19    double *q = 0;
20    double *q_tilde = 0;
21    cria(&p);
22    cria(&p_tilde);
23    cria(&q);
24    cria(&q_tilde);
25    zera(p);
26    zera(p_tilde);
27    zera(q);
28    zera(q_tilde);
29
30    //Criação e inicialização das escalares
31    double rho = 1;
32    double rho_anterior, alpha, beta, nrmr;
33
34    //Iterações do solver
35    int i;
36    for (i = 1; i < MAXIT; i++) {
37
38      //Linha 3
39      rho_anterior = rho;
40      rho = multiplica2Vetores(r, r_tilde);
41
42      //Linha 4
43      if (rho == 0) {
44        return -1;
45      }
46
47      //Linhas 5-12
48      if (i == 1) {
49        copia(p, r);
50        copia(p_tilde, r_tilde);
51      }else {
52        beta = (!rho_anterior) ? 0 : rho / rho_anterior;
53        soma2Vetores(p, 1, r, beta, p);
```

```
54          soma2Vetores(p_tilde, 1, r_tilde, beta, p_tilde);
55        }
56
57        //Linhas 13-14
58        multiplicaVetorPorA(q, p);
59        multiplicaVetorPorA(q_tilde, p_tilde, true);
60
61        //Linha 15
62        alpha = multiplica2Vetores(p_tilde, q);
63        alpha = (!alpha) ? 0 : rho / alpha;
64
65        //Linha 16
66        soma2Vetores(x, 1, x, alpha, p);
67
68        //Linhas 18-19
69        soma2Vetores(r, 1, r, -alpha, q);
70        soma2Vetores(r_tilde, 1, r_tilde, -alpha, q_tilde);
71
72        //Linha 17 (calcula norma sobre o vetor de resíduos, r)
73        norma(r, &nrmr);
74        if (nrmr < TOL) {
75          break;
76        }
77      }
78
79      //Sincroniza as threads
80      sincroniza();
81
82      //Libera memória no host e dispositivo
83      libera(p);
84      libera(p_tilde);
85      libera(q);
86      libera(q_tilde);
87      libera(r);
88      libera(r_tilde);
89
90      //Verifica se chegou ao MAXIT sem convergir para retornar
91      //código de erro, se for o caso
92      return (i < MAXIT) ? i : -1;
93    }
```

Código Fonte 4.1: Código fonte do Método dos Gradientes BiConjugados clássico — BICG. As indicações de linhas referem-se ao Algoritmo 3.

```
1   void cria(double **a) {
2     checkCudaErrors(cudaMalloc((void**)a, sizeof(double) * n));
3   }
```

Código Fonte 4.2: Código fonte da função que aloca memória para um vetor.

```
1  void zera(double *a) {
2    checkCudaErrors(cudaMemset((void*)a, 0, sizeof(double) * n));
3  }
```

Código Fonte 4.3: Código fonte da função que inicializa um vetor com zeros.

```
1  void copia(double *a_dest, const double *b_src) {
2    checkCudaErrors(cudaMemcpy(a_dest, b_src,
3        (size_t)(n * sizeof(b_src[0])), cudaMemcpyDeviceToDevice));
4  }
```

Código Fonte 4.4: Código fonte da função que copia os dados de um vetor para outro.

```
1  void libera(double *a) {
2    checkCudaErrors(cudaFree(a));
3  }
```

Código Fonte 4.5: Código fonte da função que libera a memória alocada para um vetor.

```
1  void norma(const double *r, double *nrmr) {
2    CUBLAScheckCudaErrors(cublasDnrm2(cublasHandle, n, r, 1, nrmr));
3  }
```

Código Fonte 4.6: Código fonte da função de cálculo da norma.

```
1  void sincroniza() {
2    cudaDeviceSynchronize();
3  }
```

Código Fonte 4.7: Código fonte da função que sincroniza *threads* no dispositivo.

```
1  double multiplica2Vetores(const double *a, const double *b) {
2    double t = 0;
3    CUBLAScheckCudaErrors(cublasDdot(cublasHandle, n, a, 1, b, 1, &t));
4    return t;
5  }
```

Código Fonte 4.8: Código fonte da função que multiplica dois vetores.

```
1  void soma2Vetores(double *r, const double ac, const double *a,
2          const double bc, const double *b) {
3    zera(vetTemp_A);
4    zera(vetTemp_B);
5    CUBLAScheckCudaErrors(cublasDaxpy(cublasHandle, n, &ac,
6        a, 1, vetTemp_A, 1));
7    CUBLAScheckCudaErrors(cublasDaxpy(cublasHandle, n, &bc,
8        b, 1, vetTemp_B, 1));
9    zera(r);
10   CUBLAScheckCudaErrors(cublasDaxpy(cublasHandle, n, &UM,
11       vetTemp_A, 1, r, 1));
12   CUBLAScheckCudaErrors(cublasDaxpy(cublasHandle, n, &UM,
13       vetTemp_B, 1, r, 1));
14 }
```

Código Fonte 4.9: Código fonte da função que soma dois vetores.

```
1  void multiplicaVetorPorA(double *a_dest, const double *b_src,
2          bool isTransposta = false) {
3    zera(a_dest);
4    checkCudaErrors(cusparseDcsrmv(cusparseHandle,
5      ((isTransposta) ?
6        CUSPARSE_OPERATION_TRANSPOSE :
7        CUSPARSE_OPERATION_NON_TRANSPOSE),
8      n, n, nz, &UM, descrA, data->K_aval,
9      data->nosCentrais_arow, data->IDs_acol, b_src, &ZERO, a_dest));
10 }
```

Código Fonte 4.10: Código fonte da função que multiplica um vetor por uma matriz.

4.3.3 Gradientes Conjugados Quadrático (CGS)

No Método dos Gradientes BiConjugados, BICG (Seção 4.3.2), o vetor de resíduo r_i pode ser considerado o produto de r_0 pelo i-ésimo grau polinomial de A, na forma

$$r_i = P_i(A)r_0.$$

Este mesmo polinomial satisfaz:

$$\tilde{r}_i = P_i(A)\tilde{r}_0,$$

de modo que

$$\rho_i = (\tilde{r}_i, r_i) = (P_i(A^T)\tilde{r}_0, P_i(A)r_0) = (\tilde{r}_i, P_i^2(A)r_0). \tag{4.12}$$

Isto sugere que, se $P_i(A)$ reduz r_0 a um vetor menor r_i, então pode ser vantajoso aplicar este operador de "contração" duas vezes, e computar

$$r_i = P_i^2(A)r_0. \tag{4.13}$$

A Equação 4.12 mostra que os coeficientes de iteração ainda podem ser recuperados a partir desses vetores, o que torna fácil encontrar a aproximação correspondente de x. Esta abordagem é chamada de Método dos Gradientes Conjugados Quadrático (*Conjugate Gradient Squared Method* - CGS) (Barrett et al., 1987).

O CGS chega a ser duas vezes mais rápido que o BICG, apesar de ter o mesmo número de operações por iteração, o que é coerente com a aplicação do operador de "contração" duas vezes. Além disto, o CGS não envolve o cálculo de A^T e, se este cálculo for impeditivo no BICG, o CGS pode ser atraente.

Entretanto, não há razão para o operador de "contração", mesmo reduzindo o resíduo inicial r_0, continuar fazendo o mesmo nas k vezes em que for aplicado, $r^{(k)} = P_k(A)r_0$. Isto é evidenciado pelo comportamento de convergência altamente irregular do CGS, principalmente se a situação inicial estava perto da solução.

O pseudocódigo do CGS sugerida por Barrett et al. (1987) é dado segundo o Algoritmo 4.

4.3.4 Gradientes BiConjugados Estabilizado (BICG-Stab)

O método dos Gradientes BiConjugados Estabilizado (BICG-Stab) foi desenvolvido para resolver sistemas lineares não simétricos enquanto evita certos padrões de convergência irregulares do Método dos Gradientes Conjugados Quadrático, CGS (Seção 4.3.3). Ao invés de aplicar o operador de "contração" $r_i = P_i^2(A)r_0$ (Equação 4.13), aplicamos

$$r_i = Q_i(A)P_i(A)r_0,$$

Algoritmo 4: Método dos Gradientes Conjugados Quadrático — CGS

Input : A, b
Output: x

$r_0 = b - Ax_0$
for $i = 1, 2, \ldots$ **do**
  $\rho_i = r_{i-1}^T r_0$
  if $\rho_i = 0$ **then (FALHOU)**;
  if $i = 1$ **then**
    $p_i = u_i = r_{i-1}$
  else
    $\beta = \rho_i / \rho_{i-1}$
    $u_i = r_{i-1} + \beta q_{i-1}$
    $p_i = u_i + \beta(q_{i-1} + \beta p_{i-1})$
  end if
  $t_i = Ap_i$
  $\alpha = \rho_i / (r_0^T t_i)$
  $q_i = u_i - \alpha t_i$
  $\tilde{u}_i = u_i + q_i$
  $x_i = x_{i-1} + \alpha \tilde{u}_i$
  if x_i é preciso o suficiente **then (PARE)**;
  $\tilde{q}_i = A\tilde{u}_i$
  $r_i = r_{i-1} - \alpha \tilde{q}_i$
end for

sendo que Q_i é o i-ésimo grau polinomial que descreve a atualização de descida mais íngreme. Como resultado, muitas vezes, observamos uma menor perda de precisão do resíduo.

A respeito de sua implementação, o BICG-Stab requer dois produtos matriz-vetor e quatro produtos internos, dois produtos internos a mais do que BICG e CGS. No entanto, o método não requer a matriz de transposição, o que no geral mantém aproximadamente a mesma velocidade de convergência.

O pseudocódigo do BICG-Stab sugerida por Barrett et al. (1987) é dado segundo o Algoritmo 5.

Algoritmo 5: Método dos Gradientes BiConjugados Estabilizado — BICG-Stab

Input : A, b
Output: x

$r_0 = b - Ax_0$
$\rho_0 = \alpha_0 = \omega_0 = 1$
$v_0 = p_0 = 0$
for $i = 1, 2, \ldots$ **do**
  $\rho_i = r_0^T r_{i-1}$
  $\beta = (\rho_i/\rho_{i-1})(\alpha_{i-1}/\omega_{i-1})$
  $p_i = r_{i-1} + \beta(p_{i-1} - \omega_{i-1} v_{i-1})$
  $v_i = Ap_i$
  $\alpha_i = \rho_i/(r_0^T v_i)$
  $s_i = r_{i-1} - \alpha_i v_i$
  $t_i = As_i$
  $\omega_i = (t_i^T s_i)/(t_i^T t_i)$
  $x_i = x_{i-1} + \alpha_i p_i + \omega_i s_i$
  if x_i é preciso o suficiente **then (PARE)**;
  $r_i = s_i - \omega_i t_i$
end for

4.3.5 Gradientes BiConjugados Estabilizado Aprimorado (I-BICG-Stab)

O Gradientes BiConjugado Estabilizado Aprimorado (*Improved Biconjugate Gradient Stabilized Method*) — I-BiCG-Stab (Algoritmo 6) (Yang & Brent, 2002a,b) é baseado BiCG-Stab, com operações reorganizadas para atender às recomendações para execução na GPU sem perda estabilidade numérica.

O objetivo principal da reorganização é permitir que todos os produtos internos, que normalmente exigem pontos de sincronização, sejam agrupados e, depois de computados todos eles, executem a sincronização apenas uma vez na iteração. Como são necessárias várias operações de produto interno em diferentes vetores, o tempo de comunicação necessário pode ser sobreposto de forma eficiente com o tempo computacional das atualizações do vetor. Portanto, o custo global da comunicação em computadores com memória distribuída paralela pode ser significativamente reduzido, mas nenhum ganho é percebido em *hardware* sequencial.

Algoritmo 6: Método dos Gradientes BiConjugado Estabilizado Aprimorado — I-BICG-Stab

Input : A, b
Output: x

1 $r_0 = b - Ax_0$
2 $u_0 = Ar_0$
3 $f_0 = A^T r_0$
4 $q_0 = v_0 = z_0 = 0$
5 $\sigma_{-1} = \pi_0 = \phi_0 = \tau_0 = 0$
6 $\sigma_0 = r_0^T u_0$
7 $\rho_0 = \alpha_0 = \omega_0 = 1$
8 **for** $i = 1, 2, \ldots$ **do**
9 $\quad \rho_i = \phi_{i-1} - \omega_{i-1}\sigma_{i-2} + \omega_{i-1}\alpha_{i-1}\pi_{i-1}$
10 $\quad \delta_i = (\rho_i/\rho_{i-1})\alpha_{i-1}$
11 $\quad \beta_i = \delta_i/\omega_{i-1}$
12 $\quad \tau_i = \sigma_{i-1} + \beta_i\tau_{i-1} - \delta_i\pi_{i-1}$
13 $\quad \alpha_i = \frac{\rho_i}{\tau_i}$
14 $\quad v_i = u_{i-1} + \beta_i v_{i-1} - \delta_i q_{i-1}$
15 $\quad q_i = Av_i$
16 $\quad s_i = r_{i-1} - \alpha_i v_i$
17 $\quad t_i = u_{i-1} - \alpha_i q_i$
18 $\quad z_i = \alpha_i r_{i-1} + \beta_i z_{i-1} - \alpha_i\delta_i v_{i-1}$
19 $\quad \phi_i = r_0^T s_i$
20 $\quad \pi_i = r_0^T q_i$
21 $\quad \gamma_i = f_0^T s_i$
22 $\quad \eta_i = f_0^T t_i$
23 $\quad \theta_i = s_i^T t_i$
24 $\quad \kappa_i = t_i^T t_i$
25 $\quad \omega_i = \theta_i/\kappa_i$
26 $\quad \sigma_i = \gamma_i - \omega_i\eta_i$
27 $\quad r_i = s_i - \omega_i t_i$
28 $\quad x_i = x_{i-1} + z_i + \omega_i s_i$
29 $\quad$ **if** x_i é preciso o suficiente **then (PARE)**;
30 $\quad u_i = Ar_i$
31 **end for**

Além disso, com agrupamento de operações BLAS semelhantes (subprogramas básicos de Álgebra Linear) em diferentes estruturas de dados, a paralelização total da operação é possível. Por exemplo, os cálculos dos produtos internos das linhas 19-24 podem ser calculados em

paralelo entre si. De forma equivalente, pode-se paralelizar as operações de atualização vetorial das linhas 16-17 e 27-28.

O I-BiCG-Stab aproveita as atualizações em vetores distintos, independentes entre si, para realizar paralelismo, mas sem exigir novas estruturas de dados (Figura 4.3) ou qualquer comunicação. A quantidade de pontos de sincronização, no entanto, foi reduzida de 4 para apenas 1 em cada iteração.

Capítulo 5

Resultados experimentais

Todos os resultados foram obtidos executando a aplicação desenvolvida em um servidor que contém $2\times$ Intel® Xeon® CPU E5-2620 v2 a 2.10 GHz (6 núcleos por soquete e 24 CPUs), com 64 GB de RAM e GPU NVIDIA® k40m (com 12 GB de GDDR5 SDRAM e 2880 núcleos Cuda a 745 MHz), executando o Ubuntu 18.04.1 LTS 64 *bits* e as ferramentas de compilação Cuda, versão 10.0.130.

A premissa mais importante para entender os resultados é que toda a solução é computada apenas na GPU. Além disso, nenhuma conversão ou transferência de dados desnecessária é realizada. Em um método baseado apenas em nós (como ocorre nos métodos sem malha), os dados diretos e indiretos relacionados ao nó precisam ser mantidos na mesma forma e no mesmo local físico na memória durante todo o processo.

Para fins de referência e avaliação de desempenho, os algoritmos mais importantes também foram implementados na versão sequencial para execução em CPU de *thread* única. Em todos os casos de teste cujo valor medido apresentou alguma variação, foram realizadas três execuções para se obter a média aritmética. Os resultados comparativos são apresentados a seguir.

5.1 Geração da nuvem de nós

O primeiro desafio é a geração da nuvem de nós. Supõe-se que a nuvem não exista previamente. Assim sendo, como apenas o domínio do problema é fornecido, a geração, o possível refinamento e a relação de vizinhança também devem ser processados.

O estágio de distribuição dos nós do domínio e até o ajuste de suas coordenadas para garantir uma melhor cobertura são tipicamente $O(n)$ (n é o número de nós). No entanto, a definição de vizinhança em um algoritmo de força bruta teria um custo de $O(n^2)$ em sua versão sequencial. Além disso, ele requer memória compartilhada para acessar as coordenadas de todos os outros nós da nuvem.

O algoritmo k-vizinhos mais próximos (knn) usando a divisão regular do domínio em células (*grid*), conforme mostrado na Seção 3.3.1, é usado. O espaço de pesquisa é pequeno e a tarefa pode ser executada com um custo computacional próximo ao linear.

A escolha do algoritmo knn a ser adotado é puramente uma questão de desempenho, pois o resultado obtido neste estágio é o mesmo independente do método utilizado. A Seção 5.3.1 discute esses problemas em detalhes.

5.2 Perturbação nas coordenadas dos nós

Uma análise que foi realizada é sobre os diferentes arranjos de nós impactam a solução quando o número de nós é mantido.

Um deslocamento aleatório (perturbação) nas coordenadas dos nós é introduzido para avaliar como o *solver* é afetado por nuvens de nós não uniformes. O máximo de perturbação é parametrizado por uma porcentagem da distância entre dois nós vizinhos da distribuição uniforme.

Por exemplo, quando há uma perturbação de 50% em uma nuvem com uma distância média entre nós de 10 u.n. (unidades de medida), é criado um um raio imaginário à partir da coordenada original que circunda 5 u.n. (50% de 10 u.n.). Nesse domínio do círculo, a nova coordenada para o nó em questão é escolhida aleatoriamente, garantindo que não haja sobreposição entre os nós vizinhos. O mesmo é feito para os nós de contorno, mas assegurando que o deslocamento ocorra mantendo-o na borda. A Figura 5.1 mostra fragmentos de três domínios com valores diferentes atribuídos ao deslocamento máximo, mas sem alterar o número total de nós.

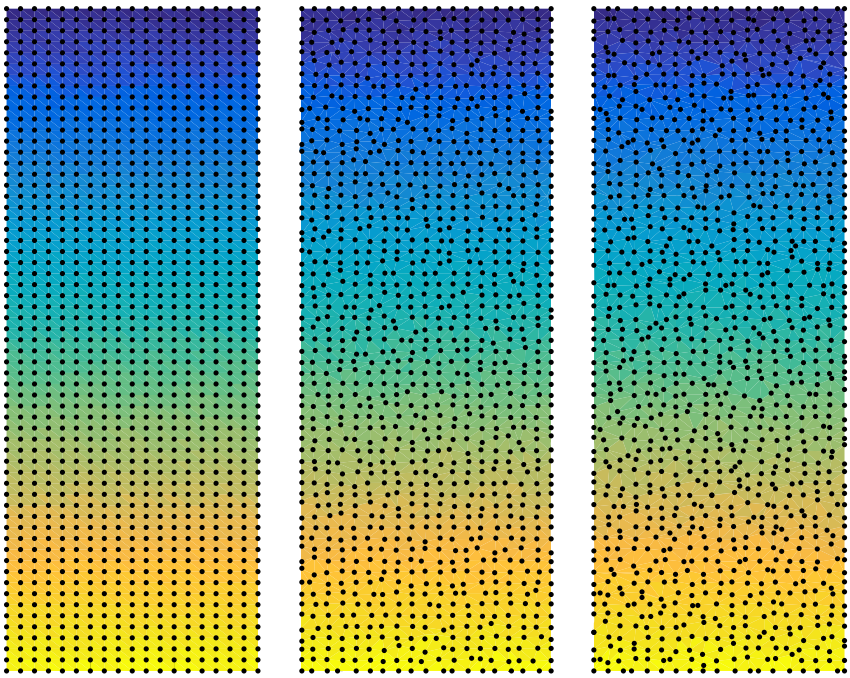


Figura 5.1: Comparação entre perturbações de 0%, 20% e 40%, respectivamente, nas coordenadas dos nós da mesma nuvem.

A matriz de rigidez, resultante da mesma nuvem de nós mas variando o grau de perturbação, é analisada quanto ao seu número de condição (Figura 5.2) e também quanto ao número de iterações do *solver* até convergir para a solução (Figura 5.3). O impacto do crescimento da perturbação é perceptível, exigindo mais iterações do *solver* para convergência. Apesar disso, não há uma alteração na magnitude do número da condição e, consequentemente, o conjunto de *solvers* capazes de resolver o sistema é mantido. Observe que o CGS não aparece no gráfico, pois falhou ao convergir para os casos testados.

O MLPG possui propriedades importantes para execução em um ambiente de processamento paralelo e memória distribuída (discutidas na Seção 3), que permite a computação de um nó por

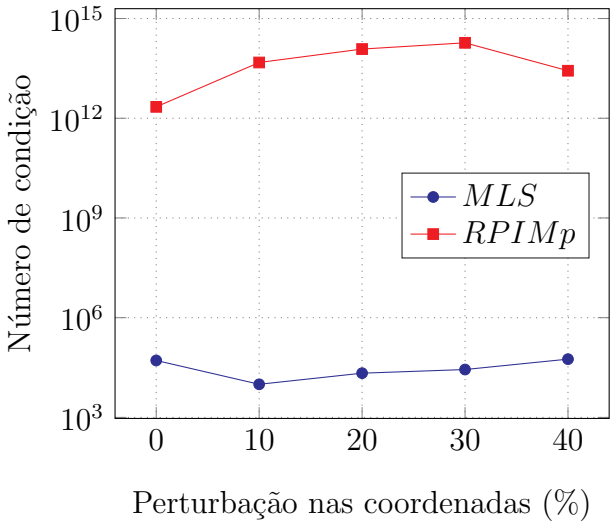


Figura 5.2: Evolução do número da condição da matriz de rigidez global ao inserir uma perturbação nas coordenadas dos nós da nuvem. Uma nuvem com 10201 nós e 9 nós no domínio de suporte é usada.

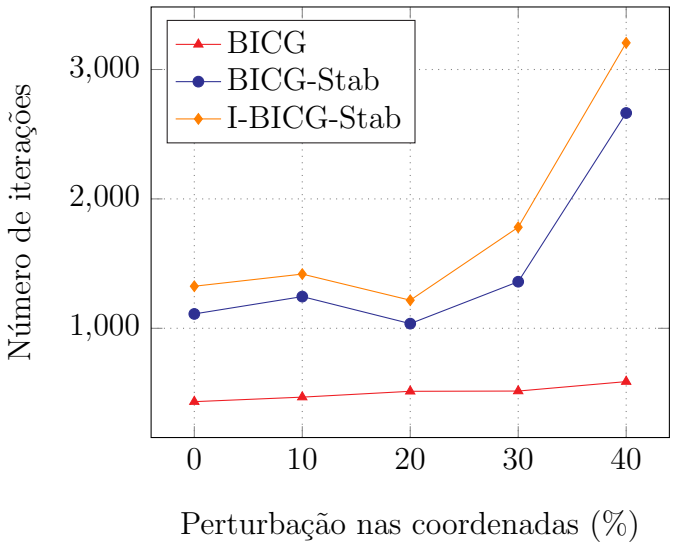


Figura 5.3: Evolução do número de iterações dos *solvers* necessárias para convergência no MLPG-MLS com o mesmo número de nós na nuvem e 9 nós no domínio de suporte, ao inserir uma perturbação nas coordenadas dos nós.

thread, sem comunicação entre elas (Fonseca et al., 2010). Além disso, os valores resultantes desta etapa referem-se a uma única linha da matriz de rigidez e afetam um pequeno grupo de valores no vetor F (RHS). Isto gera duas implicações:

1. Os dados resultantes da computação, que são uma linha da matriz de rigidez, e os valores correspondentes no vetor F (RHS), são mantidos na mesma estrutura de dados que é usada no *solver*;

2. Cada nó pode ser computado em um núcleo de GPU.

Existe, então, um processo trivialmente paralelo que, apesar de ter cálculos mais complexos quando comparado ao FEM, possui melhores propriedades para o processamento em GPU. Isso ocorre devido ao fato de não ser necessária a coloração ou qualquer tratamento de condição de corrida (operações atômicas).

A aceleração do algoritmo é mais alta e possui custo computacional linearmente proporcional ao número de nós da nuvem de entrada. Esse comportamento se deve ao baixo custo computacional das operações por nós, o que torna a sobrecarga das criações de *threads* mais expressivas. Esse comportamento é refletido no crescimento linear no gráfico mostrado na Figura 5.4.

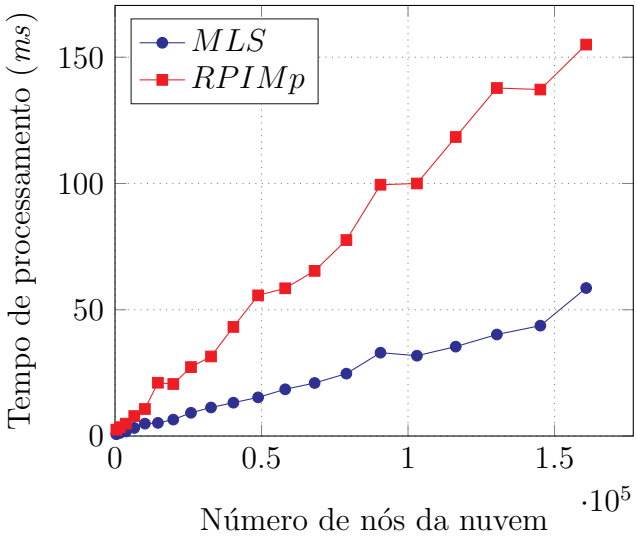


Figura 5.4: Comportamento essencialmente linear do tempo de processamento da GPU do estágio de montagem da matriz de rigidez global e vetor F (RHS) diante do crescimento do número de nós da nuvem.

O impacto do número de nós do domínio de suporte no sistema de equações resultante também pode ser analisado. No MLPG-RPIMp, o domínio de suporte do nó é de definição trivial e refere-se aos n nós mais próximos do nó analisado (distância euclidiana). No MLPG-MLS essa definição é mais criteriosa, na qual o conjunto de n nós mais próximos é definido a

partir da avaliação de pertencimento do nó em questão ao domínio de influência de cada nó vizinho avaliado. Somente então, é decidida a sua inclusão ou não no subconjunto de nós que influenciam o nó analisado.

O impacto do crescimento do domínio de suporte no condicionamento da matriz, sobretudo usando MLS, é perceptível (Figura 5.5), mas não de magnitude significativa ou de forma que um comportamento possa ser definido. Uma análise neste sentido é feita na Figura 5.6, na qual é mostrado o número de condição da matriz em função do número de nós. O número de condição aumenta com o número de nós, mas segundo os testes realizados seu valor tende a se estabilizar para um número maior de nós.

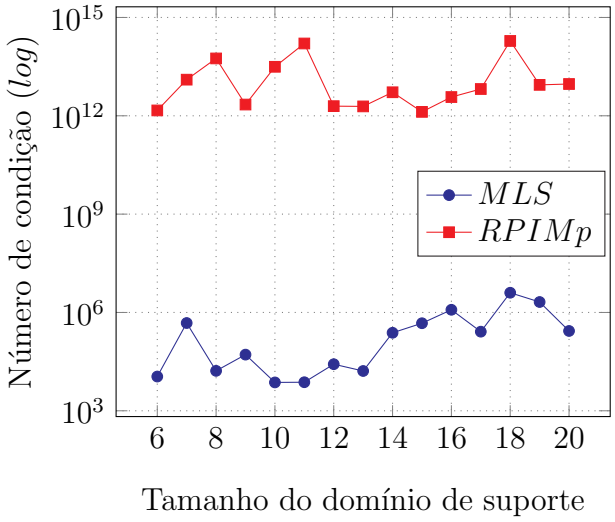


Figura 5.5: Evolução do número de condição da matriz de rigidez global diante da variação do número de nós no domínio de suporte. Uma nuvem de 10201 nós com distribuição regular é usada.

5.3 *Solver*

São implementados quatro *solvers* dentre os mais utilizados na literatura e que melhor convergiram para a solução de problemas de MLPG. É importante enfatizar que nenhum deles pode resolver todos os casos, sendo necessário alterar os parâmetros do MLPG para garantir a convergência. Todos os algoritmos dos *solvers* foram recriados usando computação paralela e aproveitam o formato das estruturas de dados e sua localização na memória do dispositivo,

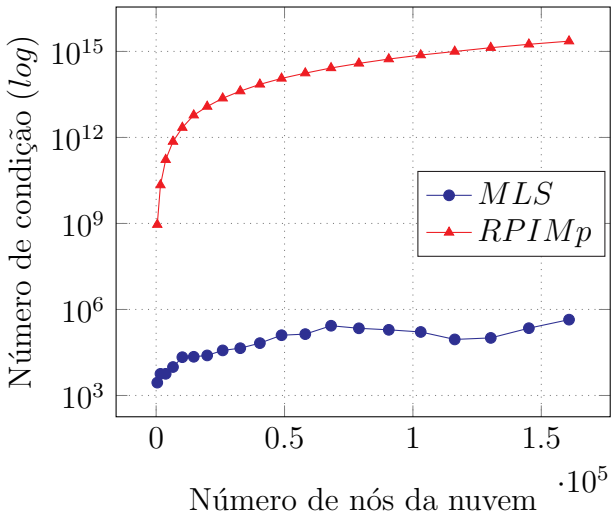


Figura 5.6: Evolução do número de condição da matriz de rigidez global diante da variação do número de nós da nuvem. Uma nuvem com distribuição regular e 9 nós no domínio de suporte são usados.

alocada nas etapas anteriores. Além da diferença de *hardware*, os algoritmos são diferentes devido à maneira como os dados são representados em cada ambiente. No entanto, não existe uma diferença significativa em termos do número de iterações necessárias para a convergência na CPU e nas mesmas versões do algoritmo GPU.

Os *solvers* implementados são computacionalmente semelhantes e têm as mesmas instruções básicas (Seção 4.1.3), mas as pequenas diferenças causam comportamentos ligeiramente diferentes.

A evolução do custo computacional por iteracão à medida que a quantidade de nós da nuvem aumenta é similar em todos os *solvers* selecionados (Figura 5.7). Isto é esperado, uma vez que o conjunto de operações internas também é similar. Nesta análise, a variação entre o melhor (BICG-Stab) e o pior (I-BICG-Stab) é de cerca de 20% da medida total, e tal diferença diz respeito às operações internas de cada *solver*. Esta é uma métrica que sozinha é inconclusiva, sobretudo quando mostra tão pouca variação entre as soluções.

A segunda análise diz respeito ao número de iterações até que a convergência para uma solução seja alcançada (Figura 5.8). Como já discutido na Seção 4.3.3, o CGS converge apenas para um pequeno número de nós da nuvem e não é uma solução viável para os problemas em questão.

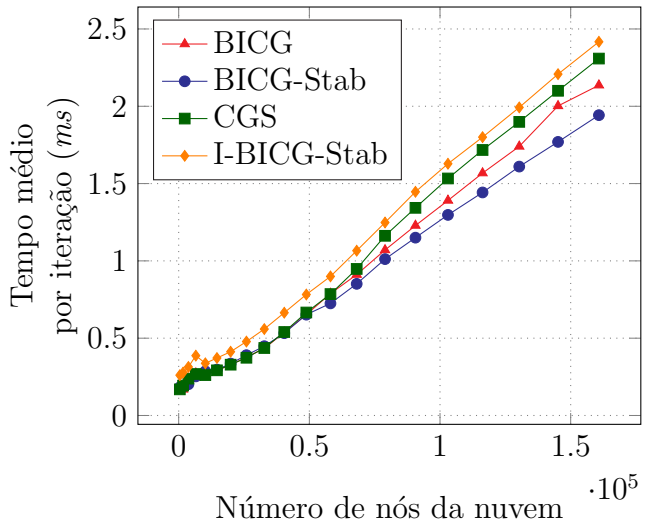


Figura 5.7: Evolução do tempo de execução da iteração na GPU de cada solucionador diante do crescimento do número de nós da nuvem.

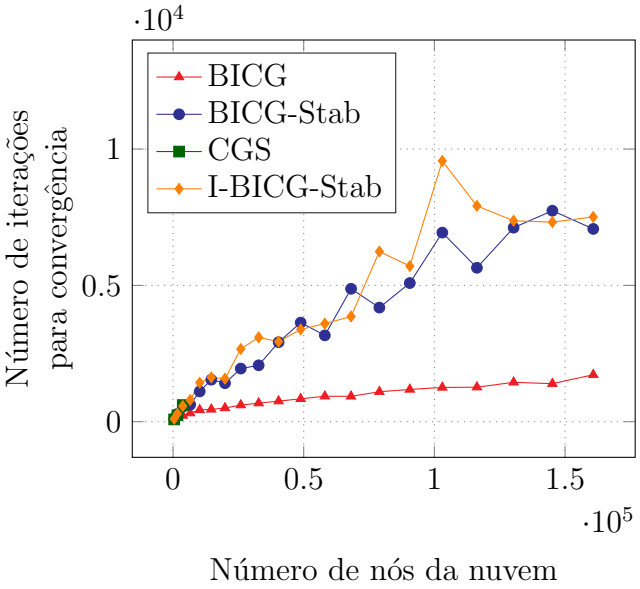


Figura 5.8: Número de iterações necessárias para a convergência de cada solucionador diante do crescimento no número de nós da nuvem. O MLPG-MLS com o domínio de suporte contendo 9 nós é usado nesta análise. Observe que casos de não convergência (valores ausentes na linha referente ao CGS) são excluídos para não prejudicar a visualização.

Os outros *solvers* foram capazes de resolver todos os casos de teste apresentados, mas o BICG se destaca como uma solução que converge cerca de 3× mais rápido que os outros, BICG-Stab e I-BICG-Stab, os quais possuem resultados semelhantes.

O tempo total para cada *solver* convergir é mostrado na Figura 5.9. Considerando a semelhança de tempo por iteração, o destaque permanece com o *solver* que exigiu menos iterações, o BICG, que convergiu para o resultado cerca de 3× mais rápido que os outros.

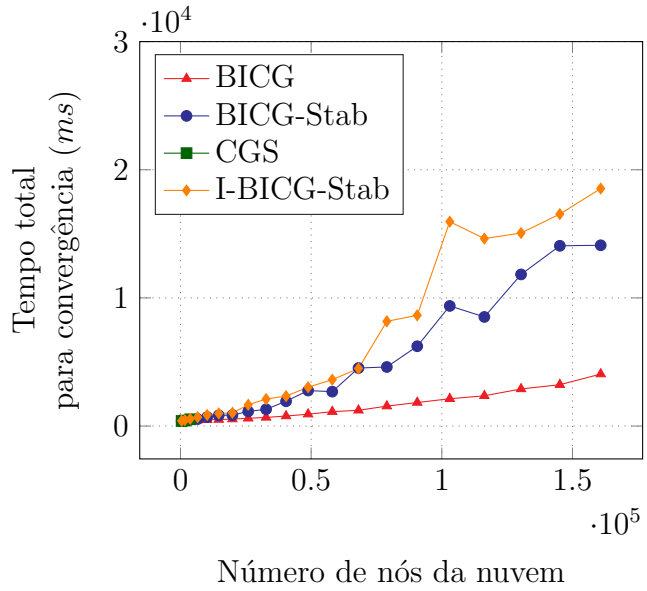


Figura 5.9: Evolução do tempo total para convergência dos *solvers* diante do crescimento do número de nós da nuvem. Observe que casos de não convergência (valores ausentes na linha referente ao CGS) são excluídos para não prejudicar a visualização.

O I-BICG-Stab, baseado no BICG-Stab, é semelhante à sua referência quanto ao número de iterações para convergência, o que era esperado, uma vez que eles executam basicamente as mesmas operações. Como o tempo de iteração no I-BICG-Stab é um pouco superior, o tempo total para chegar à solução, consequentemente, também é maior. A diferença entre eles, em relação à implementação, é que o I-BICG-Stab opta por reduzir os pontos de sincronização no dispositivo em detrimento do número de operações internas. A expectativa era que, dessa maneira, o paralelismo fosse mais eficiente e o tempo final fosse menor, o que não pôde ser verificado na prática.

Sem dúvida, o BICG obteve os melhores resultados finais. No entanto, considerando o ambiente de memória substancialmente paralelo e distribuído que é objeto deste trabalho, o I-BICG-Stab é um *solver* interessante, pois possui eficácia e consumo de memória equivalentes a outros métodos testados e, ao mesmo tempo, requer menos pontos de sincronização de *threads* em seu Kernel CUDA. Isso significa menos sobrecarga e menos tempo de espera na computação de *threads*, intensificando o uso de *hardware* em operações algébricas do *solver*.

5.3.1 Análise de speedup

A análise de desempenho da solução e *speedup* são apresentadas de duas maneiras: individualmente para cada estágio e, em seguida, na avaliação da solução completa. Em todos os casos, estará presente a comparação entre algoritmos equivalentes para execução sequencial em uma única *thread* na CPU (em azul) e execução paralela com muitas *threads* para GPU (em vermelho).

O primeiro estágio diz respeito à distribuição dos nós e à definição de vizinhança (k-vizinhos mais próximos, knn), que são importantes para a criação do domínio de suporte do MLPG. Três algoritmos são testados (seções 3.3.1 e 3.3.1): força bruta para CPU (CPU knn-BF), força bruta para GPU (GPU knn-BF) e uma solução otimizada para GPU usando *grid* (GPU knn-Grid). O número de vizinhos a serem encontrados é definido como 20 para uma comparação coerente, e os resultados são mostrados na Figura 5.10.

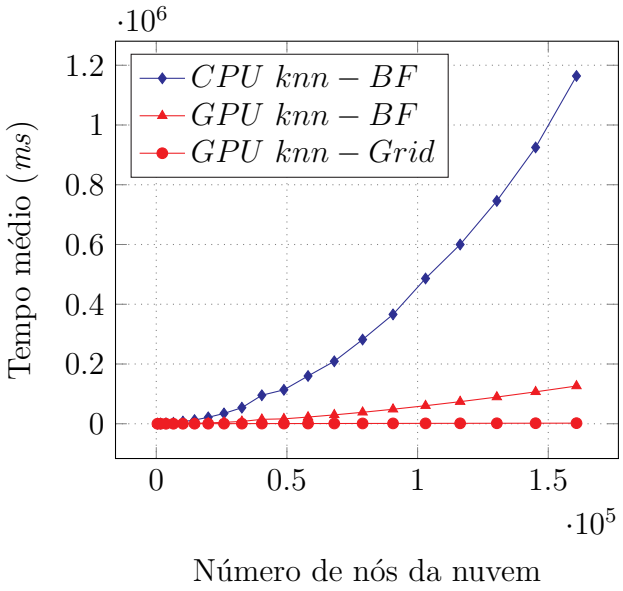


Figura 5.10: Comparação de tempo de execução entre o algoritmo knn-BF ("força bruta", com as versões CPU, thread único e GPU, muitos threads) e knn-Grid da GPU. Uma distribuição de nós regular e vizinhos de 20 nós são usados.

Como esperado de um algoritmo $O(n^2)$, o crescimento do tempo de processamento da CPU knn-BF é muito mais sensível ao aumento do número de nós do que os outros. Uma implementação adaptada para a GPU equivalente a ela, chamada GPU knn-BF, é apresentada nos gráficos para que todos os algoritmos tenham a mesma implementação em comparação

nas duas plataformas. O tempo de processamento da versão da GPU também é sensível ao crescimento do número de nós, mas em uma escala muito menor.

Para esse estágio específico, é avaliada uma solução criada para explorar o paralelismo da GPU, GPU knn-Grid, que possui um comportamento quase insensível ao crescimento do número de nós. Uma mudança no comportamento da curva knn-Grid da GPU (na Figura 5.10) é esperada apenas quando o número de células da *grid* exceder o número de *threads* na GPU. No entanto, devido ao *hardware* usado, o impacto não pôde ser demonstrado.

O próximo estágio é calcular a contribuição do nó para a matriz do sistema (montagem da matriz de rigidez). Duas funções de forma, MLS e RPIMp, implementadas de maneira semelhante na CPU e GPU, são analisadas. Os resultados são mostrados na Figura 5.11.

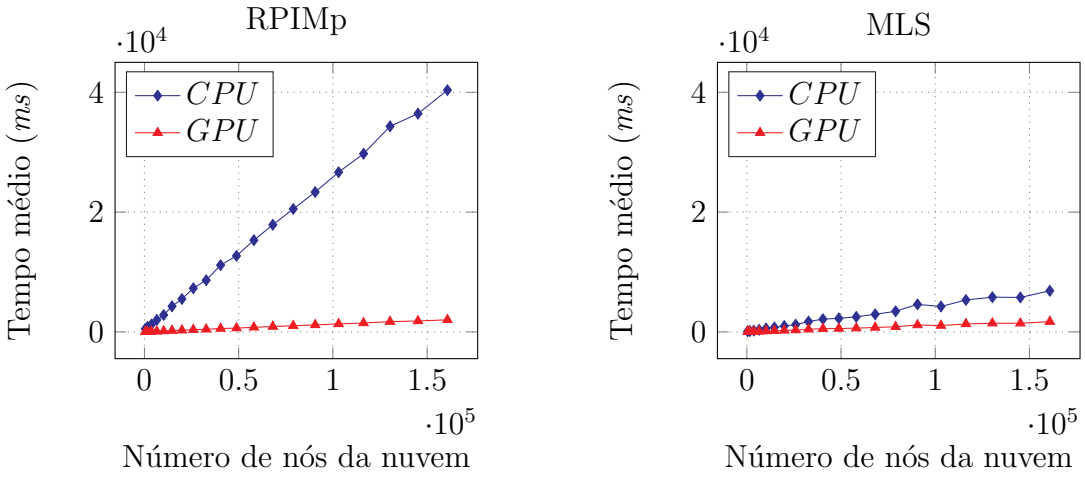


Figura 5.11: Comparação de tempo de execução entre a GPU (grupo de *threads*) e a CPU (*thread* única) no estágio de montagem da matriz de rigidez global e vetor F (RHS). Uma distribuição regular com 9 nós no domínio de suporte é usada.

É importante notar que os algoritmos têm um comportamento bastante linear nas duas plataformas. No entanto, a inclinação da curva na GPU é muito menos acentuada, sendo quase insensível ao crescimento no número de nós. Isso ocorre porque cada nó é tratado em um *thread* da GPU. Outra questão a considerar é que o comportamento dessas curvas é independente do problema que está sendo resolvido, sendo influenciado exclusivamente pelo número de nós da nuvem e pelo domínio de suporte.

O próximo estágio, mostrado na Figura 5.12, é o *solver*. Para entender os resultados mostrados, é preciso saber que o tempo medido é a execução de uma iteração de cada método. Dessa forma, é possível avaliar o impacto no crescimento do número de nós, independentemente do número da condição do sistema de equações a ser resolvido.

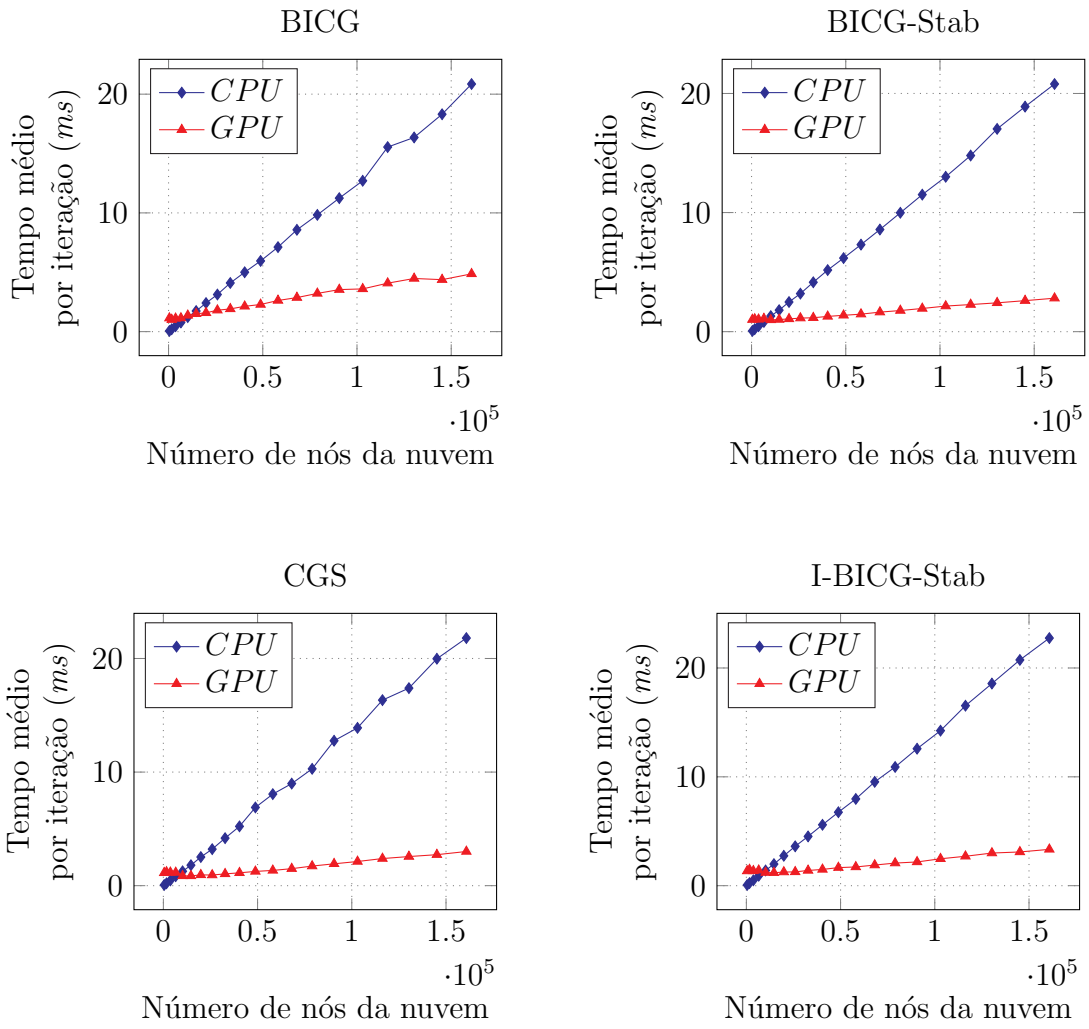


Figura 5.12: Comparação de tempo de execução entre a GPU (grupo de *threads*) e a CPU (*thread* única) no estágio do solucionador. O MLPG-MLS com uma distribuição regular de nós e 9 nós no domínio de suporte é usado.

Um fato importante é que, devido ao alto número de condição do sistema, algumas instâncias do problema não convergiram para a solução, sobretudo quando o MLPG-RPIMp é usado. A não convergência se deve à característica de todos os *solvers* baseados no sub-espaço de Krylov: o acúmulo de erro entre as iterações.

Os quatro *solvers* implementados são analisados individualmente em duas versões, para CPU e GPU, mas executando o mesmo conjunto e sequência de instruções (Figura 5.12). Ou seja, o que afeta o comportamento das curvas dos gráficos é como as operações básicas são implementadas em cada arquitetura (Seção 4.1.3). Uma observação importante é que, até aproximadamente 10.000 nós na nuvem, as versões da CPU são mais eficientes. Isso é esperado porque os estágios de inicialização do algoritmo são computacionalmente mais caros na GPU e são justificados apenas em problemas que exigem um alto custo computacional. Depois disso, a versão do *solver* implementada em GPU é muito menos sensível ao crescimento do número de nós da nuvem.

A evolução do tempo percentual gasto em cada estágio da solução completa na CPU versus GPU é analisada na Figura 5.13, e o tempo gasto no processamento total é analisado na Figura 5.14. O conjunto knn-BF, MLPG-MLS e I-BICG-Stab é selecionado por permitir uma comparação honesta, uma vez que foram implementadas versões equivalentes para CPU e GPU. Nesses gráficos, conforme o esperado, usando um algoritmo de força bruta (BF), a etapa de cálculo da vizinhança representa aproximadamente 80% do tempo total de processamento. A montagem do sistema de equações tem uma porcentagem quase irrelevante e o *solver* representa o restante. A informação mais importante mostrada é o *speedup* alcançado de 7×, usando GPU.

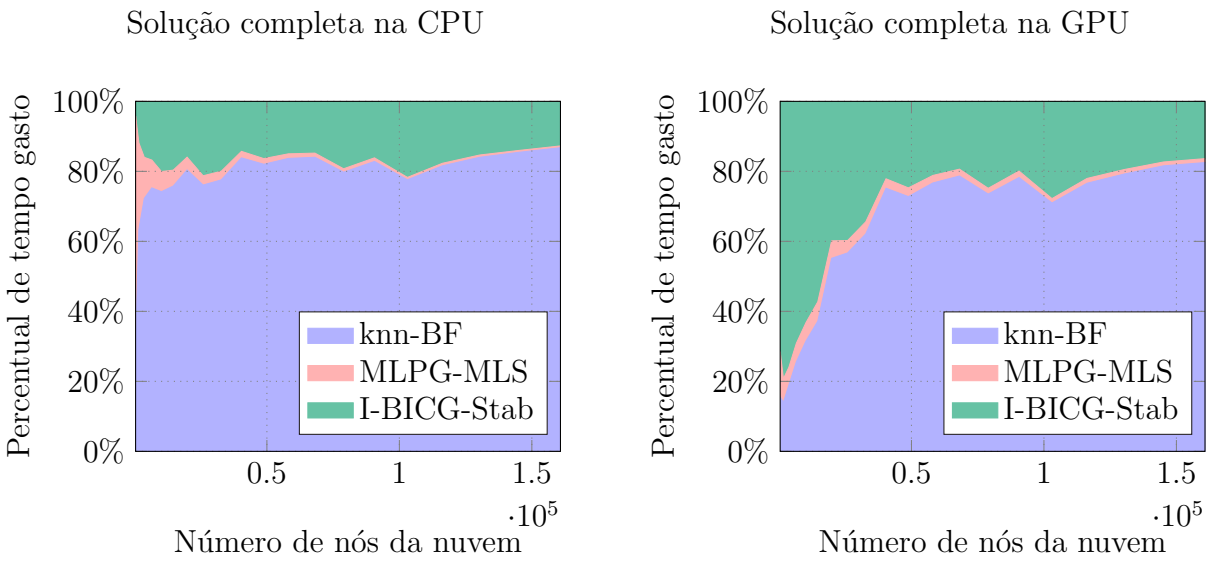


Figura 5.13: Evolução do tempo percentual gasto em cada estágio da solução completa na CPU *versus* GPU, usando o mesmo conjunto de soluções.

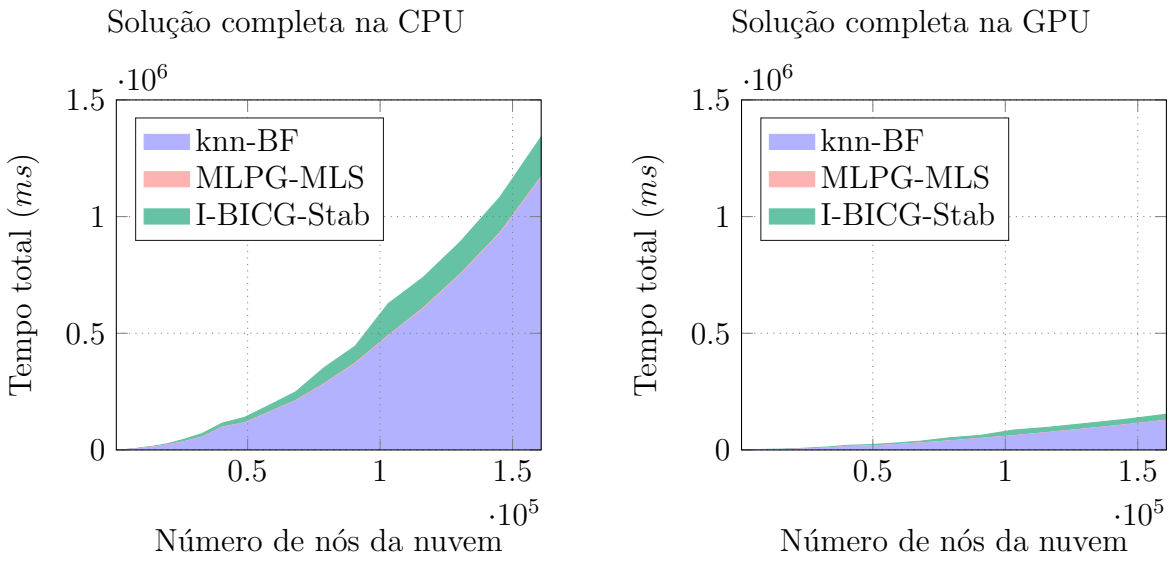


Figura 5.14: Evolução do tempo gasto em cada estágio da solução completa na GPU *versus* CPU, usando o mesmo conjunto de soluções.

Por fim, a Figura 5.15 e a Figura 5.16 mostram o resultado final deste trabalho: a distribuição percentual e o tempo gasto em cada estágio da solução. São selecionados o algoritmo knn-Grid para processamento de vizinhos, o MLPG-MLS para compor a matriz de rigidez e dois *solvers*, o BICG e o I-BICG-Stab. Nesses gráficos, o tempo total de processamento é mostrado usando o conjunto de soluções executado na GPU. O mesmo algoritmo de definição de vizinhança (knn-Grid, mais eficiente que o BF) e a mesma montagem do sistema de equações são usados, permitindo melhor visualização do impacto da escolha do *solver*. Por não existir solução sequencial equivalente de todo o conjunto, não é apropriado analisar *speedup*. Mas é possível comparar as duas configurações finais completas em GPU e ver que a solução definida com o BICG converge até 5× mais rápido do que com o I-BICG-Stab.

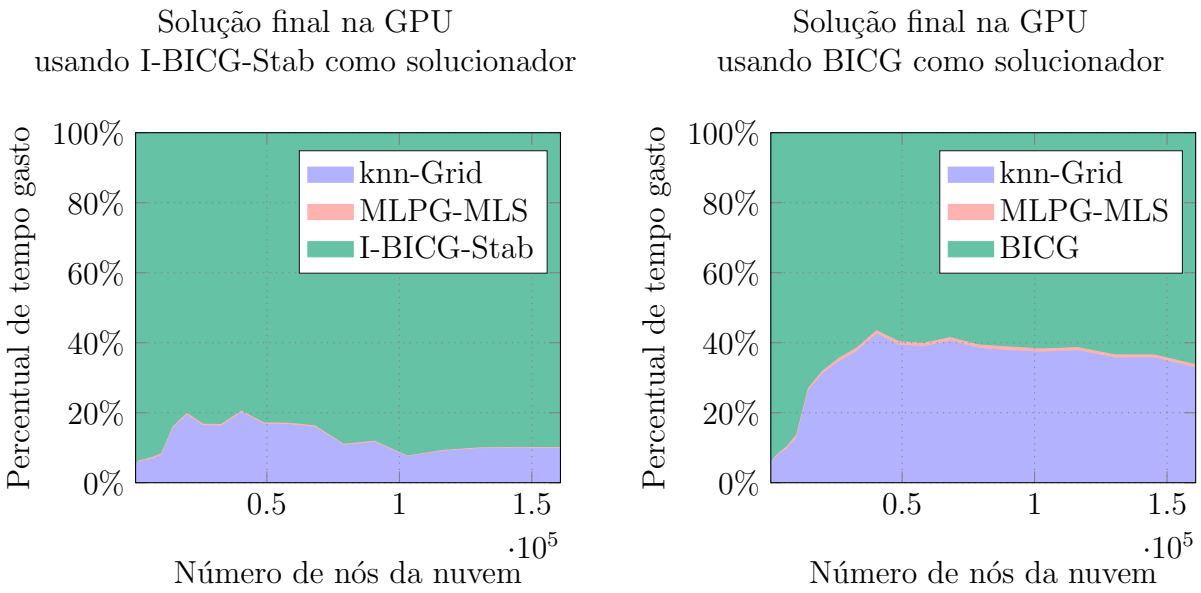


Figura 5.15: Evolução do percentual de tempo gasto em cada estágio da solução completa de GPU para um exemplo de problem reala. À esquerda, a solução usando I-BICG-Stab e, à direita, BICG como solucionador.

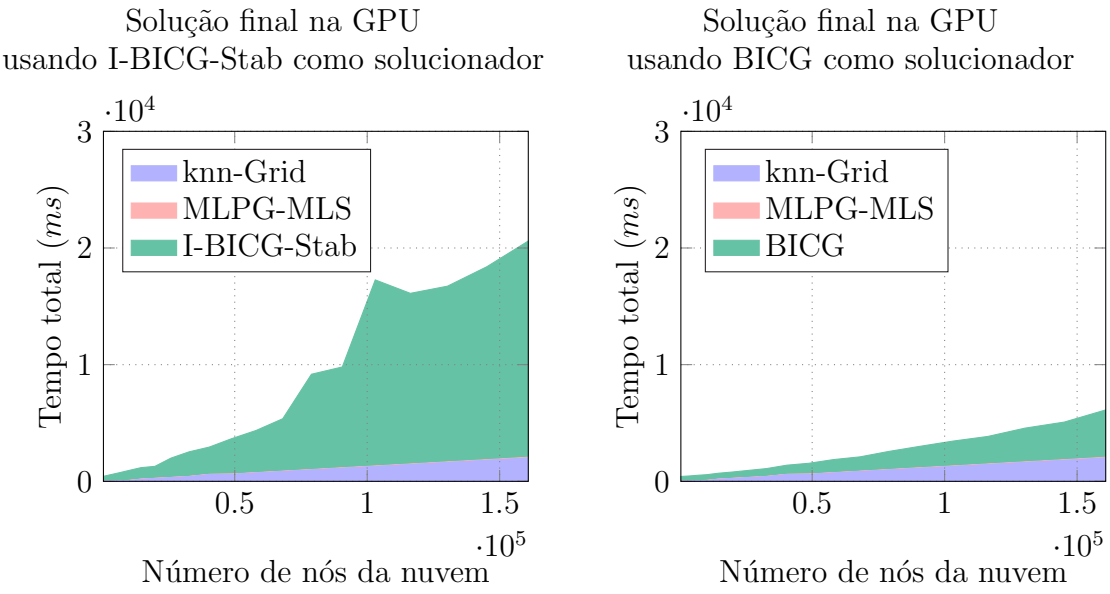


Figura 5.16: Evolução do tempo gasto em cada estágio da solução completa de GPU para um exemplo de problema real. À esquerda, a solução usando I-BICG-Stab e, à direita, BICG como solucionador.

Capítulo 6

Conclusão

Os métodos sem malha são uma alternativa ao FEM para solucionar numericamente problemas de valores de contorno. Esses métodos são mais atraentes quando os problemas de interesse possuem propriedades que tornam impraticável o uso de uma malha ou mesmo a garantia de sua qualidade, como por exemplo em problemas com muitos detalhes na sua geometria, com deformações ao longo do tempo ou problemas tridimensionais. Apesar disso, os métodos sem malha têm um custo computacional mais alto e o sistema linear gerado geralmente tem um número de condição também mais alto. Para tornar estes métodos mais competitivos, as GPUs vêm sendo utilizadas na computação destes, de forma a viabilizar sua utilização, sobretudo ao tornar o tempo de processamento menos sensível ao crescimento do número de nós obtidos na etapa de discretização. Neste trabalho, é apresentada uma solução ponta a ponta em GPU incluindo a discretização do problema, montagem da matriz de rigidez e subsequente solução do sistema de equações lineares.

Para a etapa de discretização, foram distribuídos nós em um grid regular com posterior recolocação para ajustar o contorno do domínio e a perturbação desejada (com objetivo de tornar a distribuição menos regular). O algoritmo knn com um grid de suporte é implementado para a definição de vizinhança, sendo capaz de realizar buscas locais de forma independente para cada nó. O resultado foi tornar o tempo da tarefa próximo de ser constante (uma curva de baixa inclinação no gráfico), praticamente insensível ao crescimento da quantidade de nós.

A etapa de montagem da matriz de rigidez é realizada pelo método MLPG e utiliza uma formulação local para a forma fraca, o que possibilita grande independência entre subdomínios

e dispensa o uso de malhas auxiliares para integração — sendo portanto caracterizado como um método verdadeiramente sem malha. Duas funções de forma são avaliadas, o MLS e o RPIMp. Esta escolha afeta diretamente o condicionamento do sistema de equações a ser resolvido posteriormente. O MLPG-RPIMp, apesar de possuir a propriedade do delta de Kronecker e definição de domínio de suporte de fácil computação, resulta em um sistema de equações com número de condição muito expressivo, tornando-o facilmente sem solução usando qualquer um dos *solvers* testados. Já o MLPG-MLS, que exige uma técnica especial para impor as condições de contorno de Dirichlet (penalidades, por exemplo), alcança melhores resultados do *solver* devido ao melhor condicionamento de sua matriz de rigidez.

Em geral, o MLPG é adequado para cálculos em GPU, pois permite que todos os nós sejam computados simultaneamente, sem a necessidade de tratar a condição de corrida, comum na computação paralela. Cada nó contribui para uma única linha da matriz de rigidez e afeta um pequeno grupo de valores no vetor RHS. Essa propriedade foi explorada para que cada nó fosse computado paralelamente por uma *thread* CUDA, resultando em um tempo de processamento quase insensível ao número de nós.

Em relação à solução de sistema linear resultante, o número de condição pode ser visto como o principal desafio. A escolha da família de algoritmos dos Gradientes Conjugados se deve à capacidade de convergir para uma solução ao mesmo tempo em que é adequado para a arquitetura da GPU. Foram testados os métodos BICG, BICG-Stab, CGS e I-BICG-Stab. O CGS provou ser o menos eficaz, enquanto os outros foram capazes de resolver o mesmo conjunto de problemas. O algoritmo I-BICG-Stab possui menos pontos de sincronização, mas o BICG devido à sua simplicidade, apresentou o menor tempo total de processamento. Apesar disso, em algumas situações, um grande número de iterações foi necessário para alcançar a convergência.

A solução em GPU apresentada neste trabalho aproveita a arquitetura SIMD (*Single Instruction Multiple Data*) da GPU. Existem pontos de sincronização, mas a estrutura e a localização dos dados entre as etapas são preservadas e poucas operações atômicas são necessárias.

Como é típico nos trabalhos de solução em GPU, o resultado mais aparente é o *speedup* alcançado de 7×. Este valor pode ser ainda mais expressivo com o crescimento do número de nós da nuvem durante a discretização de domínio. Contudo, a contribuição mais significativa é

a disponibilidade da solução de GPU de fim-a-fim do *Meshless Local* Petrov-Galerkin (MLPG) sem intermediação vinda da CPU durante o processo.

Na prática, isso significa que problemas com um grande número de nós, cuja computação em CPU é inviável devido à grande quantidade de operações, podem ser resolvidos com a solução em GPU. Este fato é perceptível ao analisar resultados comparativos entre as soluções completas implementadas para CPU e GPU. A versão em CPU se torna rapidamente impraticável com o aumento do número de nós, enquanto a versão em GPU apresenta um incremento no tempo de processamento muito menos significativo.

A memória total usada, embora em *hardwares* diferentes, não possui diferenças significativas e a precisão da resposta é a mesma.

6.1 Trabalhos futuros

Neste trabalho foi mostrado que GPU pode ser usada para tornar o método sem malha computacionalmente competitivo quando comparado a outros métodos numéricos concorrentes. São sugestões de continuidade deste trabalho:

Melhorar a localização de nós vizinhos no momento em que o domínio de suporte é gerado. Para atender a esta necessidade, faz-se necessário encontrar maneiras de localizar vizinhos ainda mais rapidamente do que a pesquisa local distribuída apresentada neste trabalho, pois a tarefa prevalece como sendo de alto custo computacional. Isto torna-se ainda mais crítico em problemas em que o pré-processamento da nuvem de nós não é possível. Existem bibliotecas que implementam algoritmos como *KD-Trees* e *BD-Trees*, resolvendo o problema de forma eficiente, como a ANN (*Approximate Nearest Neighbor*), em `C++`. Entretanto, existem pesquisas de soluções também em GPU, como os trabalhos de Garcia et al. (2008) e Mei et al. (2016);

Usar soluções consolidadas, onde esperam-se implementações otimizadas sobretudo em operações de álgebra linear em GPU. Como apresentado neste trabalho, um problema importante identificado é o mal condicionamento do sistema de equações lineares resul-

tante. Na prática, para resolver tais sistemas, muitas iterações de *solver* são necessárias e, consequentemente, muitas operações sobre matrizes e vetores. Apesar de ter sido preocupação durante o desenvolvimento realizado, ainda assim há necessidade de buscar soluções cada vez mais eficientes e que explorem as propriedades mais atuais do *hardware*. Algumas bibliotecas sugeridas são: CULA (EM Photonics, 2014), MAGMA (Chrzeszczyk & Chrzeszczyk, 2013), AmgX (Naumov et al., 2015), Thrust (Phillips & Fatica, 2015) e *Message Passing Interface* — MPI (Kang et al., 2015; Boehmer et al., 2011; Griebel & Zaspel, 2010);

Melhorar o *solver* para aumentar sua capacidade de resolver problemas mal condicionados. Neste sentido, faz-se necessário buscar *solvers* mais apropriados para sistemas de condicionamento ruim e explorar pré-condicionadores adaptados às propriedades do problema, que sobretudo permitam computação em GPU. É o que trata os trabalhos de Anzt et al. (2017), Ament et al. (2010), Richter et al. (2016) e Zhang & Zhang (2013).

Explorar um ambiente multi-GPUs. Por mais eficiente que seja a solução, existem problemas com propriedades que demandam mais memória e capacidade computacional que uma única GPU é capaz de oferecer. Uma possibilidade é, então, usar múltiplas GPUs, como nos trabalhos de Richter et al. (2016), Verschoor & Jalba (2012), Kasmi et al. (2017), Cevahir et al. (2009), Zabelok et al. (2014) e Monakov (2013);

Tratar problemas com estruturas complexas, tridimensionais e com múltiplos materiais. O problema para o qual a solução apresentada neste trabalho foi testada é didático, mas distante das necessidades do mercado. Isto não afeta qualquer conclusão acerca das decisões de projeto, mas para tornar aplicável a problemas e necessidades reais é importante que sejam adaptadas para problemas de maior complexidade, como nos tratados nos trabalhos de Shivanian (2015), Ma et al. (2015) e Guan et al. (2014);

Melhorar o condicionamento da matriz de rigidez. Por fim, como apresentado neste trabalho, o MLPG gera um sistema de equações lineares mal condicionado. Essa propriedade, compartilhada pelos demais métodos sem malha, é um ponto negativo e que precisa ser levado em consideração. Muito já se evoluiu (Chen et al., 2017), mas o condicionamento do sistema precisa ser melhorado ainda na etapa de composição da matriz de rigidez global, de forma a tornar o trabalho do *solver* computacionalmente mais barato e, consequentemente, mais rápido.

Referências Bibliográficas

Almasi, G. S. (1994). *Highly Parallel Processing - The Benjamin/Cummings series in computer science and engineering*. Benjamin-Cummings Publishing Co.,Subs. of Addison Wesley Longman,US. [citado na(s) páginas(s) 12]

Almasi, G. S. & Gottlieb, A. (1990). Review of Highly Parallel Computing. *IBM Syst. J.*, *29*(1), 165–166. [citado na(s) páginas(s) 4, 12]

Ament, M., Knittel, G., Weiskopf, D., & Straßer, W. (2010). A parallel preconditioned conjugate gradient solver for the poisson problem on a multi-GPU platform. *Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-Based Processing, PDP 2010*, 583–592. [citado na(s) páginas(s) 97]

Amorim, L. P. (2011). Geração de Nuvem de Pontos para Métodos sem Malhas. Master's thesis, Universidade Federal de Minas Gerais, Universidade Federal de Minas Gerais. [citado na(s) páginas(s) 3]

Amorim, L. P., Mesquita, R. C., Goveia, T. D. S., & Correa, B. C. (2019). Node-to-Node Realization of Meshless Local Petrov Galerkin (MLPG) Fully in GPU. *IEEE Access*, *7*, 151539–151557. [citado na(s) páginas(s) 3]

Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., & Sorensen, D. (1999). *LAPACK users' guide* (3rd ed. ed.). Philadelphia: Society for Industrial and Applied Mathematics. [citado na(s) páginas(s) 11]

Anzt, H., Gates, M., Dongarra, J., Kreutzer, M., Wellein, G., & Köhler, M. (2017). Preconditioned Krylov solvers on GPUs. *Parallel Computing*, *68*, 32–44. [citado na(s) páginas(s) 59, 60, 97]

Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., & Yelick, K. A. (2006). The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley. [citado na(s) páginas(s) 12]

Atluri, S. N. & Zhu, T. (1998). A new Meshless Local Petrov-Galerkin (MLPG) approach in computational mechanics. *Computational Mechanics*, *22*(2), 117–127. [citado na(s) páginas(s) 6, 7, 8, 34, 44, 45, 46]

Baer, J.-L. (2009). *Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors* (1st ed.). [citado na(s) páginas(s) 13]

Balanis, C. A. (1989). *Advanced Engineering Electromagnetics* (1 ed.). John Wiley & Sons Inc. [citado na(s) páginas(s) 35]

Barrett, R., Berry, M., Chan, T. F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., & DerVorst, H. V. (1987). *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods (Miscellaneous Titles in Applied Mathematics Series No 43)* (1 ed.). Society for Industrial and Applied Mathematics. [citado na(s) páginas(s) 67, 68, 74, 75]

Belytschko, T., Lu, Y. Y., & Gu, L. (1994). Element-free Galerkin methods. *Int. J. Numer. Meth. Engng.*, *37*(2), 229–256. [citado na(s) páginas(s) 6]

Benzi, M. (2002). Preconditioning techniques for large linear systems: a survey. *J. Comput. Phys.*, *182*(2), 418–477. [citado na(s) páginas(s) 4]

Bergamaschi, L., Martínez, Á., & Pini, G. (2009). An efficient parallel MLPG method for poroelastic models. *CMES - Computer Modeling in Engineering and Sciences*, *49*(3), 191–215. [citado na(s) páginas(s) 7]

Boehmer, S., Cramer, T., Hafner, M., Lange, E., Bischof, C., & Hameyer, K. (2011). Numerical simulation of electrical machines by means of a hybrid parallelisation using MPI and OpenMP for finite-element method. *IET Science, Measurement & Technology*, *6*(5), 339. [citado na(s) páginas(s) 97]

Bolz, J., Farmer, I., Grinspun, E., & Schröoder, P. (2003). Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. *ACM Trans. Graph.*, *22*(3), 917–924. [citado na(s) páginas(s) 4]

Bruaset, A. M. & Langtangen, H. P. (1997). Object-Oriented Design of Preconditioned Iterative Methods in Diffpack. *ACM Trans. on Math. Software*, *23*(1), 50–80. [citado na(s) páginas(s) 4]

Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., & Hanrahan, P. (2004). Brook for GPUs. *ACM Transactions on Graphics*, *23*(3), 777. [citado na(s) páginas(s) 16]

Carey, G. F. & Jiang, B. (1986). Element-by-element Linear and Nonlinear Solution Schemes. *Communications in Applied Numerical Methods*, *2*, 145–153. [citado na(s) páginas(s) 4]

Cecka, C., Lew, A. J., & Darve, E. (2011). Assembly of Finite Element Methods on graphics processors. *International Journal for Numerical Methods in Engineering*, *85*(5), 640–669. [citado na(s) páginas(s) 4, 27]

Cevahir, A., Nukada, A., & Matsuoka, S. (2009). Fast conjugate gradients with multiple GPUs. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. [citado na(s) páginas(s) 4, 97]

Cheik Ahamed, A. K. & Magoulès, F. (2017). Conjugate gradient method with graphics processing unit acceleration: CUDA vs OpenCL. *Advances in Engineering Software*, *111*, 32–42. [citado na(s) páginas(s) 60]

Chen, J. S., Hillman, M., & Chi, S. W. (2017). Meshfree Methods: Progress Made after 20 Years. *Journal of Engineering Mechanics*, *143*(4), 04017001. [citado na(s) páginas(s) 3, 97]

Chrzeszczyk, A. & Chrzeszczyk, J. (2013). *Matrix computations on the GPU, CUBLAS and MAGMA by example*. [citado na(s) páginas(s) 97]

Correa, B. C. (2014). Paralelização do método Meshless Local Petrov-Galerkin (MLPG) utilizando processadores gráficos (GPU) e CUDA. Master's thesis, Universidade Federal de Minas Gerais, Universidade Federal de Minas Gerais. [citado na(s) páginas(s) 6]

Correa, B. C., Mesquita, R. C., & Amorim, L. P. (2015). CUDA Approach for Meshless Local Petrov-Galerkin Method. *IEEE Transactions on Magnetics*, *51*(3), 1–4. [citado na(s) páginas(s) 4, 5, 7, 43]

Dalton, S., Bell, N., Olson, L., & Garland, M. (2014). Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations. [citado na(s) páginas(s) 14]

Davis, T. (2009). *UMFPACK Version 5.4.0 User Guide*. [citado na(s) páginas(s) 11]

De, S. & Bathe, K. J. (2001). Towards an efficient meshless computational technique: the method of finite spheres. *Engineering Computations*. [citado na(s) páginas(s) 7]

Demmel, J. W. (1997). *Applied Numerical Linear Algebra* (1 ed.). SIAM. [citado na(s) páginas(s) 49, 56]

Developers, N. (2019). CUDA C PROGRAMMING GUIDE - Design Guide. Technical Report PG-02829-001_v10.1, NVIDIA Corporation. [citado na(s) páginas(s) 22, 25, 26, 27]

Dongarra, J. J., Duff, I. S., Sorensen, D. C., & Vorst, H. V. (1990). *Solving Linear Systems on Vector and Shared Memory Computers*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics. [citado na(s) páginas(s) 60]

Dziekonski, A., Sypek, P., Lamecki, A., & Mrozowski, M. (2012). Finite Element Matrix Generation on a Gpu. *Progress In Electromagnetics Research*, *128*(May), 249–265. [citado na(s) páginas(s) 5]

EM Photonics, I. (2014). *CULA Sparse Reference Manual* (Release S5 ed.). Partnership with NVIDIA: EM Photonics, Inc. [citado na(s) páginas(s) 97]

Erhel, J., Traynard, A., & Vidrascu, M. (1991). An element-by-element preconditioned conjugate gradient method implemented on a vector computer. *Parallel Computing*, *17*(9), 1051–1065. [citado na(s) páginas(s) 4]

Fletcher, R. (1976). Conjugate gradient methods for indefinite systems. In Watson (Ed.), *Numerical Analysis*, volume 506 of *Lecture Notes in Mathematics* chapter 7, (pp. 73–89). Springer Berlin Heidelberg. [citado na(s) páginas(s) 48]

Fonseca, A. R. (2011). *Algoritmos Eficientes em Métodos sem Malha*. PhD thesis, Universidade Federal de Minas Gerais, Universidade Federal de Minas Gerais. [citado na(s) páginas(s) 4, 5, 6, 7, 33, 36]

Fonseca, A. R., Correa, B. C., Silva, E. J., & Mesquita, R. C. (2010). Improving the Mixed Formulation for Meshless Local Petrov Galerkin Method. *IEEE Transactions on Magnetics*, *46*, 2907–2910. [citado na(s) páginas(s) 35, 39, 42, 43, 48, 82]

Fonseca, A. R., Viana, S. A., Silva, E. J., & Mesquita, R. C. (2008). Imposing Boundary Conditions in the Meshless Local Petrov-Galerkin Method. *IET Science, Measurement & Technology*, *2*(6), 387–394. [citado na(s) páginas(s) 38]

Franco, N. B. (2007). *Cálculo Numérico* (1 edição ed.). Pearson Prentice Hall. [citado na(s) páginas(s) 1, 48, 56, 57]

Gallopoulos, E., Philippe, B., & Sameh, A. H. (2015). *Parallelism in matrix computations*. bookGallopoulos: Springer Netherlands. [citado na(s) páginas(s) 54]

Garcia, V., Debreuve, E., & Barlaud, M. (2008). Fast k nearest neighbor search using GPU. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on*, (pp. 1–6). IEEE. [citado na(s) páginas(s) 40, 41, 96]

Ghai, A., Lu, C., & Jiao, X. (2019). A comparison of preconditioned Krylov subspace methods for large-scale nonsymmetric linear systems. *Numerical Linear Algebra with Applications*, *26*(1), 1–35. [citado na(s) páginas(s) 48, 60, 61]

Gingold, R. A. & Monaghan, J. J. (1977). Smoothed particle hydrodynamics-theory and application to non-spherical stars. *Monthly Notices of the Royal Astronomical Society*, *181*, 375–389. [citado na(s) páginas(s) 6]

Gobbetti, E. & Marton, F. (2007). GPU-friendly accelerated mesh-based and mesh-less techniques for the output-sensitive rendering of huge complex 3D models. In *ACM SIGGRAPH 2007 courses*, SIGGRAPH '07, New York, NY, USA. ACM. [citado na(s) páginas(s) 7]

Golub, G. H., Loan, V., & Charles, F. (1996). *Matrix computations* (3rd ed.). Johns Hopkins University Press. [citado na(s) páginas(s) 48, 49, 50, 56, 57, 62]

Golub, G. H. & Ye, Q. (1999). Inexact Preconditioned Conjugate Gradient Method with Inner-Outer Iteration. *SIAM Journal on Scientific Computing*, *21*(4), 1305–1320. [citado na(s) páginas(s) 4]

Griebel, M. & Zaspel, P. (2010). A multi-GPU accelerated solver for the three-dimensional two-phase incompressible Navier-Stokes equations. *Computer Science - Research and Development*, *25*(1-2), 65–73. [citado na(s) páginas(s) 97]

Guan, J., Yan, S., & Jin, J. M. (2014). An accurate and efficient finite element-boundary integral method with GPU acceleration for 3-D electromagnetic analysis. *IEEE Transactions on Antennas and Propagation*. [citado na(s) páginas(s) 97]

Gustafsson, I. & Lindskog, G. (1986). A Preconditioning Technique Based on Element Matrix Factorizations. *Computer Methods in Applied Mechanics and Engineering*, *55*(3), 201–220. [citado na(s) páginas(s) 4]

Hughes, T. J. R. (2000). *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis (Dover Civil and Mechanical Engineering)*. Dover Publications. [citado na(s) páginas(s) 33]

Hughes, T. J. R., Ferencz, R. M., & Hallquist, J. O. (1987). Large-scale vectorized implicit calculations in solid mechanics on a Cray X-MP/48 utilizing EBE preconditioned conjugate gradients. *Computer Methods in Applied Mechanics and Engineering*, *61*(2), 215–248. [citado na(s) páginas(s) 4]

Hughes, T. J. R., Levit, I., & Winget, J. (1983). An element-by-element solution algorithm for problems of structural and solid mechanics. *Computer Methods in Applied Mechanics and Engineering*, *36*(2), 241–254. [citado na(s) páginas(s) 4]

Idagawa, H. S. (2017). *Implementação de um simulador de partículas utilizando o Método dos Elementos Discretos (DEM) em cluster de GPUs*. PhD thesis, Universidade Estadual de Campinas. [citado na(s) páginas(s) 11]

Jeffrey, A. (2002). *Applied Partial Differential Equations: An Introduction* (1 ed.). Academic Press. [citado na(s) páginas(s) 1, 48, 56]

Jiang, B. (1998). *The Least-Squares Finite Element Method: Theory and Applications in Computational Fluid Dynamics and Electromagnetics (Scientific Computation)* (1998 ed.). Springer. [citado na(s) páginas(s) 3, 4]

Kamranian, M., Dehghan, M., & Tatari, M. (2017). An adaptive meshless local Petrov–Galerkin method based on a posteriori error estimation for the boundary layer problems. *Applied Numerical Mathematics*, *111*, 181–196. [citado na(s) páginas(s) 3]

Kang, S. J., Lee, S. Y., & Lee, K. M. (2015). Performance comparison of OpenMP, MPI, and MapReduce in practical problems. *Advances in Multimedia*, *2015*(August). [citado na(s) páginas(s) 97]

Karatarakis, A., Metsis, P., & Papadrakakis, M. (2013). Stiffness Matrix Computation for Element Free. (June), 12–14. [citado na(s) páginas(s) 5]

Kasmi, N., Zbakh, M., Mahmoudi, S. A., & Manneback, P. (2017). Performance Evaluation and Analysis for Conjugate Gradient Solver on Heterogeneous (Multi-GPUs / Multi-CPUs) platforms. *3rd International Conference of Cloud Computing Technologies and Applications (CloudTech)*, *17*(8), 206–215. [citado na(s) páginas(s) 54, 97]

Khan, A. & Topping, B. H. V. (1996). Parallel finite element analysis using Jacobi-conditioned conjugate gradient algorithm. *Advances in Engineering Software*, *25*(2-3), 309–319. [citado na(s) páginas(s) 59]

Kirk, D. B. & Hwu, W.-m. W. (2013). *Programming Massively Parallel Processors: A Hands-on Approach* (2nd Editio ed.). [citado na(s) páginas(s) 14, 16]

Kiss, I., Badics, Z., & Gyimóthy, S. (2013). GPU-optimized parallel preconditioners for the element-by-element finite element method. *Transactions on Magnetics - Conferences*. [citado na(s) páginas(s) 59]

Kiss, I., Gyimóthy, S., Badics, Z., & Pavo, J. (2012). Parallel realization of the element-by-element FEM technique by CUDA. *IEEE Transactions on Magnetics*, *48*(2), 507–510. [citado na(s) páginas(s) 4, 5, 8, 59]

Kosec, G. & Zinterhof, P. (2013). Local strong form meshless method on multiple Graphics Processing Units. *CMES - Computer Modeling in Engineering and Sciences*, *91*(5), 377–396. [citado na(s) páginas(s) 7]

Law, K. H. (1986). A parallel finite element solution method. *Computers & Structures*, *23*(6), 845–858. [citado na(s) páginas(s) 4]

Lima, N. Z. (2011). *Desenvolvimento de um Framework para Métodos sem Malha*. PhD thesis, Universidade Federal de Minas Gerais, Universidade Federal de Minas Gerais. [citado na(s) páginas(s) 3]

Lindholm, E., Nickolls, J., Oberman, S., & Montrym, J. (2008). NVIDIA Tesla: A unified graphics and computing architecture. In *IEEE Micro*, volume 28, (pp. 39–55). [citado na(s) páginas(s) 14]

Liu, G. R. (2002). *Element free methods* (1 ed.). CRC. [citado na(s) páginas(s) 1, 6, 7, 38, 43]

Liu, G. R. (2009). *Meshfree Methods: Moving Beyond the Finite Element Method* (Second Edi ed.). CRC Press. [citado na(s) páginas(s) 37]

Liu, Y., Zhou, W., & Yang, Q. (2007). A distributed memory parallel element-by-element scheme based on Jacobi-conditioned conjugate gradient for 3D finite element analysis. *Finite Elements in Analysis and Design*, *43*(6-7), 494–503. [citado na(s) páginas(s) 4, 59]

Lobeiras, J., Amor, M., & Doallo, R. (2013). Influence of memory access patterns to small-scale FFT performance. In *Journal of Supercomputing*, volume 64, (pp. 120–131). [citado na(s) páginas(s) 27]

Ma, H., Tan, H., & Guo, Y. (2015). Three-Dimensional Induced Polarization Parallel Inversion Using Nonlinear Conjugate Gradients Method. *Mathematical Problems in Engineering*, *2015*. [citado na(s) páginas(s) 97]

Mackie, R. I. (2008). *Programming Distributed Finite Element Analysis: An Object Oriented Approach (Saxe-Coburg Publications on Computational Engineering)*. Saxe-Coburg Publications. [citado na(s) páginas(s) 11]

Mehmood, R. & Crowcroft, J. (2005). Parallel Iterative solution method for Large Sparse Linear Equation Systems. *Computer Laboratory: University of Cambridge*, (650), 22. [citado na(s) páginas(s) 59]

Mei, G., Xu, N., & Xu, L. (2016). Improving GPU-accelerated adaptive IDW interpolation algorithm using fast kNN search. *SpringerPlus*, *5*(1). [citado na(s) páginas(s) 96]

Monakov, A. (2013). Efficient Multi-GPU CUDA Linear Solvers for OpenFOAM. *Presentation*, 1–26. [citado na(s) páginas(s) 97]

Moscovici, N., Cohen, N., & Petrank, E. (2017). A GPU-Friendly Skiplist Algorithm. *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*, *2017-Septe*, 246–259. [citado na(s) páginas(s) 50]

Nakata, S., Takeda, Y., Fujita, N., & Ikuno, S. (2010). Parallel algorithm for meshfree radial point interpolation method on graphics hardware. *Electromagnetic Field Computation (CEFC), 2010 14th Biennial IEEE Conference on*. [citado na(s) páginas(s) 7]

Naumov, M. (2011). Incomplete-LU and Cholesky preconditioned iterative methods using CUSPARSE and CUBLAS. *Nvidia white paper*, 1–16. [citado na(s) páginas(s) 56]

Naumov, M., Arsaev, M., Castonguay, P., Cohen, J., Demouth, J., Eaton, J., Layton, S., Markovskiy, N., Reguly, I., Sakharnykh, N., Sellappan, V., & Strzodka, R. (2015). AmgX: A Library for GPU Accelerated Algebraic Multigrid and Preconditioned Iterative Methods. *SIAM Journal on Scientific Computing*, *37*(5), S602–S626. [citado na(s) páginas(s) 97]

Naumov, M., Chien, L. S., Vandermersch, P., & Kapasi, U. (2010). *NVIDIA CUDA Sparse Matrix library (CUSPARSE)*. NVIDIA. [citado na(s) páginas(s) 14, 52, 54]

Nvidia (2010). *NVIDIA CUDA Programming Guide - Version 3.2*. [citado na(s) páginas(s) 5, 14, 15, 25]

Nvidia (2015). *CUDA Toolkit Documentation v7.5: Programming Guide*. NVIDIA Corporation. [citado na(s) páginas(s) 28]

Nvidia (2016). CUBLAS Library User Guide. Technical report, nVidia. [citado na(s) páginas(s) 54]

Ortiz, M., Pinsky, P. M., & Taylor, R. L. (1983). Unconditionally stable element-by-element algorithms for dynamic problems. *Computer Methods in Applied Mechanics and Engineering*, *36*(2), 223–239. [citado na(s) páginas(s) 4]

Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., & Purcell, T. J. (2007). A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, *26*(1), 80–113. [citado na(s) páginas(s) 16]

Phillips, E. & Fatica, M. (2015). A CUDA Implementation of the High Performance Conjugate Gradient Benchmark. In S. A. Jarvis, S. A. Wright, & S. D. Hammond (Eds.), *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, volume 8966 of *Lecture Notes in Computer Science* (pp. 68–84). Springer International Publishing. [citado na(s) páginas(s) 97]

Pikle, N. K., Sathe, S. R., & Vyavhare, A. Y. (2018). GPGPU-based parallel computing applied in the FEM using the conjugate gradient algorithm: a review. *Sadhana - Academy Proceedings in Engineering Sciences*, *43*(7), 1–21. [citado na(s) páginas(s) 8]

Purcina, L. A. & Saramago, S. F. P. (2007). Comparação entre métodos iterativos não-estacionários e métodos heurísticos aplicados à solução de grandes sistemas lineares. In *CIBIM8 - 8 Congresso Iberoamericano de Engenharia Mecanica*, (pp. 1–10). Pontificia Universidad Católica del Perú. [citado na(s) páginas(s) 67, 68]

Refsnæs, R. H. (2010). Matrix-Free Conjugate Gradient Methods for Finite Element Simulations on GPUs. Master's thesis, Norwegian University of Science and Technology. [citado na(s) páginas(s) 4, 5]

Richter, C., Schöps, S., & Clemens, M. (2016). Multi-GPU Acceleration of Algebraic Multigrid Preconditioners. *Mathematics in Industry*, *23*, 83 – 90. [citado na(s) páginas(s) 97]

Rodgers, D. P. (1985). Improvements in Multiprocessor System Design. *SIGARCH Comput. Archit. News*, *13*(3), 225–231. [citado na(s) páginas(s) 13]

Rupp, K. (2016). CPU, GPU and MIC Hardware Characteristics over Time. [citado na(s) páginas(s) 17, 18, 19]

Rupp, K. (2018). 42 Years of Microprocessor Trend Data. [citado na(s) páginas(s) 12]

Saad, Y. (2003). *Iterative Methods for Sparse Linear Systems* (2 ed.). Society for Industrial and Applied Mathematics. [citado na(s) páginas(s) 48, 59]

Sanders, J. & Kandrot, E. (2011). *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley. [citado na(s) páginas(s) 14]

Sharma, B. & Han, J. (2019). *Learn CUDA Programming*. [citado na(s) páginas(s) 16, 24]

Sheu, T., Fang, C., & Tsai, S. (1999). Application of an element-by-element BiCGSTAB iterative solver to a monotonic finite element model. *Computers & Mathematics with Applications*, *37*(3), 57–70. [citado na(s) páginas(s) 4]

Shivanian, E. (2015). Meshless local Petrov–Galerkin (MLPG) method for three-dimensional nonlinear wave equations via moving least squares approximation. *Engineering Analysis with Boundary Elements*, *50*, 249–257. [citado na(s) páginas(s) 5, 97]

Storti, D. & Yurtoglu, M. (2015). *CUDA for Engineers: An Introduction to High-Performance Parallel Computing* (1st ed.). Addison-Wesley Professional. [citado na(s) páginas(s) 14, 16]

Taflove, A. & Hagness, S. C. (2005). *Computational Electrodynamics: The Finite-Difference Time-Domain Method, Third Edition* (3 ed.). Artech House. [citado na(s) páginas(s) 1, 33]

Trefethen, L. N. & Bau, D. (1997). *Numerical Linear Algebra*. SIAM: Society for Industrial and Applied Mathematics. [citado na(s) páginas(s) 49, 56]

Verschoor, M. & Jalba, A. C. (2012). Analysis and performance estimation of the Conjugate Gradient method on multiple GPUs. *Parallel Computing*, *38*(10-11), 552–575. [citado na(s) páginas(s) 97]

Watkins, D. S. (2010). *Fundamentals of Matrix Computations (Pure and Applied Mathematics: A Wiley Series of Texts, Monographs and Tracts)* (3 ed.). Wiley. [citado na(s) páginas(s) 56]

Winget, J. M. & Hughes, T. J. R. (1985). Solution algorithms for nonlinear transient heat conduction analysis employing element-by-element iterative strategies. *Computer Methods in Applied Mechanics and Engineering*, *52*(1-3), 711–815. [citado na(s) páginas(s) 4]

Yang, L. T. & Brent, R. P. (2002a). The Improved BiCG Method for Large and Sparse Linear Systems on Parallel Distributed Memory Architectures. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS '02, Washington, DC, USA. IEEE Computer Society. [citado na(s) páginas(s) 60, 61, 76]

Yang, L. T. & Brent, R. P. (2002b). The Improved BiCGStab Method for Large and Sparse Unsymmetric Linear Systems on Parallel Distributed Memory Architectures. *Proceedings of the Fifth International Conference on Algorithms and Architectures for Parallel Processing*. [citado na(s) páginas(s) 76]

Ye, X., Fan, D., Lin, W., Yuan, N., & Ienne, P. (2010). High performance comparison-based sorting algorithm on many-core GPUs. In *Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010*. [citado na(s) páginas(s) 41]

Yokota, R., Narumi, T., Sakamaki, R., Kameoka, S., Obi, S., & Yasuoka, K. (2009). Fast multipole methods on a cluster of GPUs for the meshless simulation of turbulence. *Computer Physics Communications*, *180*(11), 2066–2078. [citado na(s) páginas(s) 7]

Zabelok, S., Arslanbekov, R., & Kolobov, V. (2014). Multi-GPU kinetic solvers using MPI and CUDA. (July), 539–546. [citado na(s) páginas(s) 97]

Zhang, J. & Zhang, L. (2013). Efficient CUDA polynomial preconditioned conjugate gradient solver for finite element computation of elasticity problems. *Mathematical Problems in Engineering*, *2013*. [citado na(s) páginas(s) 97]

Zhang, J.-L., Ma, Z.-H., Chen, H.-Q., & Cao, C. (2018). A GPU-accelerated implicit meshless method for compressible flows. *Journal of Computational Physics*, *360*, 39–56. [citado na(s) páginas(s) 5, 7]

Zienkiewicz, O. C., Taylor, R. L., & Zhu, J. Z. (2013). *The Finite Element Method: Its Basis and Fundamentals* (Seventh Ed ed.). Oxford: Butterworth-Heinemann. [citado na(s) páginas(s) 1]