UNIVERSIDADE DE SÃO PAULO–USP
ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

ANA FLÁVIA PEIXOTO DE CAMARGOS

**COMPUTAÇÃO PARALELA EM GPU PARA
RESOLUÇÃO DE SISTEMAS DE EQUAÇÕES
ALGÉBRICAS RESULTANTES DA APLICAÇÃO
DO MÉTODO DE ELEMENTOS FINITOS EM
ELETROMAGNETISMO**

São Paulo
2014

Ana Flávia Peixoto de Camargos

**Computação Paralela em GPU para
Resolução de Sistemas de Equações
Algébricas Resultantes da Aplicação do
Método de Elementos Finitos em
Eletromagnetismo**

Tese de doutorado apresentada ao Programa de Pós-graduação em Engenharia Elétrica da Escola Politécnica da Universidade de São Paulo como parte dos requisitos para a obtenção do título de Doutora em Ciências.

Área de concentração: Sistemas Elétricos de Potência

Orientador: Viviane Cristine Silva

São Paulo
2014

Este exemplar foi revisado e corrigido em relação à versão original, sob responsabilidade única do autor e com a anuência de seu orientador.

São Paulo, 03 de outubro de 2014.

Assinatura do autor _____

Assinatura do orientador _____

Catálogo-na-publicação

Camargos, Ana Flávia Peixoto de

Computação paralela em GPU para resolução de sistemas de equações algébricas resultantes da aplicação do método de elementos finitos em eletromagnetismo / A.F.P. de Camargos. -- versão corr. -- São Paulo, 2014.

139 p.

Tese (Doutorado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Energia e Automação Elétricas.

1.Método dos elementos finitos 2.Métodos iterativos 3.Unidade de processamento gráfico 4.Paralelismo 5.Resolução de sistemas de equações algébricas I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Energia e Automação Elétricas II.t.

Agradecimentos

Os agradecimentos são muito importantes, pois várias pessoas me apoiaram de alguma forma. No entanto, sinto dificuldades em expressar corretamente ou expor as melhores palavras que contextualizam as diversas situações.

Primeiramente gostaria de agradecer à Deus, por me permitir vivenciar grandes oportunidades e conhecer pessoas maravilhosas ao longo da minha jornada profissional.

Ao meus pais, pelo incentivo, carinho e paciência. Apesar da ausência, sempre me apoiaram em qualquer decisão profissional. O apoio emocional sempre foi primordial nesta fase de tanta determinação e perseverança.

Ao Fernando, pelo apoio, incentivo e dedicação. Foram vários momentos ausentes e de muito nervosismo devido à distância, aos estudos e ao cansaço. Mesmo assim, sempre me incentivou para que eu finalizasse essa etapa importante na minha vida e que eu tanto almejava. A compreensão, amizade e o carinho foram imprescindíveis neste momento.

À minha orientadora Dra. Viviane Cristine Silva, pela fantástica orientação e pelo direcionamento pontual da pesquisa nos momentos mais delicados e importantes. Agradeço-lhe também pela confiança, amizade e compreensão naqueles momentos que mais precisei.

Aos Dr. José Roberto Cardoso, pela oportunidade profissional ao me convidar a fazer o doutorado no Laboratório de Eletromagnetismo Aplicado da Escola Politécnica.

Aos amigos do IFMG, os professores Me. André Roger Rodrigues e Dr. Gláucio Ribeiro Silva, pelo apoio e articulação para liberação das minhas atividades acadêmicas. Além disso, os conselhos e as palavras de amigos foram muito importantes nas minhas decisões.

Aos amigos Paloma, Marilene, Francisco e Fábio, pelas palavras de incentivo e conforto.

Aos professores Dr. Luiz Lebensztajn e Dr. Maurício Salles, pelo convívio amigável e cordial diariamente no LMAG. Agradeço também pelas palavras de conforto, incentivo e amizade. Também aos professores Dr. Carlos Sartori e Dr. Luiz Natal Rossi.

Aos colegas pesquisadores Ademir, Lucas e Daniela, pela boa convivência no laboratório e pela troca de experiências.

Ao IFMG, pela concessão do afastamento para capacitação.

Ao PEA e ao CNPQ, pelo apoio financeiro.

Resumo

Camargos, Ana Flávia Peixoto de **Computação Paralela em GPU para Resolução de Sistemas de Equações Algébricas Resultantes da Aplicação do Método de Elementos Finitos em Eletromagnetismo**. 139 p. Tese de doutorado – Escola Politécnica da Universidade de São Paulo, Universidade de São Paulo, 2014.

Este trabalho apresenta a aplicação de técnicas de processamento paralelo na resolução de equações algébricas oriundas do Método de Elementos Finitos aplicado ao Eletromagnetismo, nos regimes estático e harmônico. As técnicas de programação paralelas utilizadas foram OpenMP, CUDA e *GPUDirect*, sendo esta última para as plataformas do tipo *Multi-GPU*. Os métodos iterativos abordados incluem aqueles do subespaço Krylov: Gradientes Conjugados, Gradientes Biconjugados, Conjugado Residual, Gradientes Biconjugados Estabilizados, Gradientes Conjugados para equações normais (CGNE e CGNR) e Gradientes Conjugados ao Quadrado. Todas as implementações fizeram uso das bibliotecas CUSP, CUSPARSE e CUBLAS. Para problemas estáticos, os seguintes pré-condicionadores foram adotados, todos eles com implementações paralelizadas e executadas na GPU: Decomposições Incompletas LU e de Cholesky, *Multigrid* Algébrico, Diagonal e Inversa Aproximada. Para os problemas harmônicos, apenas os dois primeiros pré-condicionadores foram utilizados, porém na sua versão sequencial, com execução na CPU, resultando em uma implementação híbrida CPU-GPU. As ferramentas computacionais desenvolvidas foram testadas na simulação de problemas de aterramento elétrico. No caso do regime harmônico, em que o fenômeno é regido pela Equação de Onda completa com perdas e não homogênea, a formulação adotada foi aquela em dois potenciais, *A-V* aresta-nodal. Em todas as situações, os aplicativos desenvolvidos para GPU apresentaram *speedups* apreciáveis, demonstrando a potencialidade dessa tecnologia para a simulação de problemas de larga escala na Engenharia Elétrica, com excelente relação custo-benefício.

Palavras-chave: Método de Elementos Finitos. Métodos Iterativos. Unidade de Processamento Gráfico. Paralelismo. Resolução de Sistemas de Equações Algébricas..

Abstract

Camargos, Ana Flávia Peixoto de **Parallel Computing on GPU for Solving Systems of Algebraic Equations Resulting from Application of Finite Element Method in Electromagnetism**. 139 p. Ph.D. Thesis – Politécnica School at University of São Paulo, University of São Paulo, 2014.

This work presents the use of parallel processing techniques in Graphics Processing Units (GPU) for the solution of algebraic equations arising from the Finite Element modeling of electromagnetic phenomena, both in steadystate and time-harmonic regime. The techniques used were parallel programming OpenMP, CUDA and GPUDirect, the latter for those platforms of type Multi-GPU. The iterative methods discussed include those of the Krylov subspace: Conjugate Gradients, Bi-conjugate Gradients, Conjugate Residual, Bi-conjugate Gradients Stabilized, Conjugate Gradients for Normal Equations (CGNE and CGNR) and Conjugate Gradients Squared. All implementations have made use of CUSP, CUSPARSE and CUBLAS libraries. For the static problems, the following pre-conditioners were adopted, all with parallelized implementations and executed on the GPU: Incomplete decompositions, both LU and Cholesky, Algebraic Multigrid, Diagonal and Approximate Inverse. For the time-harmonic varying problems, only the first two pre-conditioners were used, but in their sequential version and running in the CPU, which yielded a hybrid CPU-GPU implementation. The developed computational tools were tested in the simulation of electrical grounding systems. In the case of the harmonic regime, in which the phenomenon is governed by the driven, lossy wave equation, the formulation adopted was that in two potential, the ungauged edge A - V formulation. In all cases, the developed GPU-based tools showed considerable speedups, showing that this is a promising technology for the simulation of large-scale Electrical Engineering problems, with excellent cost-benefit.

Keywords: Finte Element Method. Iterative Solver. Graphic Processing Unit. Parallelism. Solving Systems of Algebraic Equations..

Lista de ilustrações

Figura 1.1	Histórico da evolução dos desempenhos computacionais da CPU e GPU em GFlops/s	21
Figura 3.1	Placa de vídeo Geforce GTX da NVIDIA	36
Figura 3.2	Performance entre CPU e GPU	37
Figura 3.3	Diferença na filosofia do <i>design</i> entre CPU e GPU	38
Figura 3.4	Blocos de <i>threads</i>	39
Figura 3.5	Execução de um algoritmo CUDA	42
Figura 3.6	Organização das <i>threads</i>	43
Figura 3.7	Modelo de memória do <i>device</i> CUDA	44
Figura 3.8	Tecnologia <i>GPUDirect</i> que permite comunicação entre GPUs	46
Figura 3.9	Desempenho das arquiteturas Geforce GTX 680 e Tesla M 2090	47
Figura 3.10	Largura de banda de memória para a CPU e GPU (Tesla e Geforce)	47
Figura 5.1	Número de iterações em função dos valores do <i>shift</i> — α	68
Figura 5.2	Perfis das matrizes antes (A) e após as decomposições incompletas ILU (L e U) e IC (M).	68
Figura 6.1	Fluxograma da resolução paralela do sistema $A.x=b$, com as etapas realizadas na CPU e na GPU	77
Figura 6.2	Exemplo de código fonte para a resolução paralela em GPU de sistema de equações $A.x=b$ utilizando o método CG da biblioteca CUSP	78
Figura 6.3	Fluxograma do algoritmo paralelo para solução pelo método CG do sistema $A.x=b$ quando se utilizam as bibliotecas CUBLAS e CUSPARSE	79
Figura 6.4	Efeito da aplicação da reordenação RCM (a) gerada pela aplicação da formulação em dois potenciais do MEF de aresta (b) sobre a matriz A	80
Figura 6.5	Ilustração esquemática do modelo usado na simulação do sistema de aterramento dos casos-teste	82

Figura 6.6	Modelo 3D do eletrodo enterrado e representado por apenas um quarto do domínio, o qual foi malhado em hexaedros	84
Figura 7.1	Tempos de execução totais (s) das implementações do método CG (Seq. - CSR, CUSP e CUBLAS + CUSPARSE) - Estudo de Caso 1	89
Figura 7.2	Tempos de execução totais (s) de vários métodos - Estudo de Caso2	90
Figura 7.3	<i>Speedup</i> do método CG - Estudo de Caso 2	91
Figura 7.4	Comparação entre os tempos de execução totais (s) em dois hardwares diferentes (H1 e H2) - Estudo de Caso 2	92
Figura 7.5	Tempos de execução totais (s) para três formatos de armazenamentos das matrizes - Estudo de Caso 2	92
Figura 7.6	<i>Speedup</i> dos métodos CG e PCG - Estudo de Caso 3	93
Figura 7.7	Fluxogramas das operações para os casos C1A, C2A e C2B - Estudo de Caso 4	94
Figura 7.8	<i>Speedup</i> do caso C1A dos métodos CG e PCG - Estudo de Caso 4	95
Figura 7.9	Percentual das principais operações do IC + PCG - Estudo de Caso 4	95
Figura 7.10	Percentual das principais operações do ILU + PCG - Estudo de Caso 4	96
Figura 7.11	<i>Speedup</i> do PCG (MTH1 e MTH2) dos casos C2A e C2B - Estudo de Caso 4	98
Figura 7.12	<i>Speedup</i> do PCG (MTH1 e MTH2) - Estudo de Caso 5	101
Figura 7.13	<i>Speedup</i> do PBiCG (MTH1 e MTH2) - Estudo de Caso 5	102
Figura 7.14	<i>Speedup</i> do PCG (MTH3 e MTH4) - Estudo de Caso 5	103
Figura 7.15	Principais operações do PCG (MS6) em C2A e C2B - Estudo de Caso 5 . . .	104
Figura 7.16	Histórico da evolução da norma residual do PCG para os vários métodos (MTH1 a MTH4) - Estudo de Caso 5	104
Figura 7.17	Tempos de execução totais (s) do caso C2A - Estudo de Caso 6	106
Figura 7.18	Tempos de execução totais (s) dos métodos MTH1 até MTH3 - Estudo de Caso 7	108
Figura 7.19	<i>Speedup</i> dos métodos CG e PCG - Estudo de Caso 7	109
Figura 7.20	Distribuição das operações num sistema <i>Multi-GPU</i> dos casos C2A e C2B - Estudo de Caso 7	110
Figura 7.21	Tempos de execução totais (s) dos métodos MTH17 e MTH18 - Estudo de Caso 7	111
Figura 7.22	<i>Speedup</i> do PCG e PBiCGStab (MTH1 e MTH2) - Estudo de Caso 8	113
Figura 7.23	Distribuição das operações do PCG na <i>Multi-GPU</i> dos casos C2A e C2B - Estudo de Caso 9	114
Figura 7.24	<i>Speedup</i> do PCG e PBiCG (MTH15, MTH16 e MTH17) - Estudo de Caso 9	115

Lista de tabelas

Tabela 6.1	Características das configurações de hardware utilizadas	80
Tabela 6.2	Parâmetros geométricos e físicos dos modelo usados nos experimentos no Regime Estacionário	85
Tabela 6.3	Parâmetros geométricos e físicos dos modelo usados nos experimentos no Regime Harmônico	85
Tabela 7.1	Detalhamento dos métodos e pré-condicionador dos estudos de caso - configurações C1A e C1B	87
Tabela 7.2	Detalhamento dos formatos das matrizes, técnicas paralelas, bibliotecas e referências dos estudos de caso - configurações C1A e C1B	88
Tabela 7.3	Detalhamento dos métodos e pré-condicionador dos estudos de caso - configurações C2A e C2B	88
Tabela 7.4	Detalhamento dos formatos das matrizes, técnicas paralelas, bibliotecas e referências dos estudos de caso - configurações C2A e C2B	88
Tabela 7.5	Tempos de execução totais (s) - Estudo de Caso 1	89
Tabela 7.6	Tempos de execução totais (s) - Estudo de Caso 2	90
Tabela 7.7	Tempos de execução totais (s) - Estudo de Caso 3	93
Tabela 7.8	Tempos de execução totais (s) do caso C1A - Estudo de Caso 4	94
Tabela 7.9	Diferentes discretizações e truncamentos para os casos C2A e C2B - Estudo de caso 4	96
Tabela 7.10	Descrição dos métodos implementados dos casos C2A e C2B - Estudo de Caso 4	97
Tabela 7.11	Tempos de execução totais (s) dos casos C2A e C2B - Estudo de Caso 4 . . .	97
Tabela 7.12	Principais operações do PCG dos casos C2A e C2B - Estudo de Caso 4 . . .	98
Tabela 7.13	Porcentagem das operações na GPU do caso C2B - ii - Estudo de Caso 4 . . .	99
Tabela 7.14	Níveis de discretização das malhas (MS1 a MS5 com elementos de 1ª ordem e MS6 com elementos de 2ª ordem) - Estudo de Caso 5	99
Tabela 7.15	Descrição dos métodos implementados - Estudo de Caso 5	100

Tabela 7.16	Tempos de execução totais (s) dos métodos MTH1 e MTH2 - Estudo de Caso 5	100
Tabela 7.17	Tempos de execução totais (s) dos métodos MTH3 e MTH4 - Estudo de Caso 5	102
Tabela 7.18	Porcentagens (%) em relação ao tempo total das principais operações do PCG (MS6) - Estudo de Caso 5	103
Tabela 7.19	Níveis de discretização das malhas (MS1 a 5 com elementos de 1 ^a ordem e MS6, 2 ^a ordem) - Estudo de Caso 6	105
Tabela 7.20	Descrição dos métodos implementados - Estudo de Caso 6	105
Tabela 7.21	Tempos de execução totais (s) - Estudo de Caso 6	106
Tabela 7.22	Descrição dos métodos MTH1 até MTH8 do caso C1A - Estudo de Caso 7 .	107
Tabela 7.23	Tempos de execução totais (s) dos métodos MTH1 até MTH8 - Estudo de Caso 7	107
Tabela 7.24	Descrição dos métodos MTH9 até MTH14 do caso C1A - Estudo de Caso 7	108
Tabela 7.25	Tempos de execução totais (s) dos métodos MTH9 até MTH14 do caso C1A - Estudo de Caso 7	109
Tabela 7.26	Descrição dos métodos implementados dos casos C2A e C2B - Estudo de Caso 7	109
Tabela 7.27	Tempos de execução totais (s) dos casos C2A e C2B - Estudo de Caso 7 . .	110
Tabela 7.28	Descrição dos métodos implementados - Estudo de Caso 8	112
Tabela 7.29	Tempos de execução totais (s) dos métodos MTH1 até o MTH6 - Estudo de Caso 8	112
Tabela 7.30	Descrição dos métodos implementados dos casos C2A e C2B - Estudo de Caso 7	113
Tabela 7.31	Tempos de execução totais (s) dos casos C2A e C2B - Estudo de Caso 9 . .	114

Lista de siglas

AMG *Multigrid Algébrico*

BiCG *Biconjugate Gradient*

BiCGStab *Biconjugate Gradient Stabilized*

CG *Conjugate Gradient*

CGNE *Conjugate Gradient on the Normal Equations*

CGNR *Conjugate Gradient on the Normal Equations*

CGS *Conjugate Gradient Squared*

CNPq Conselho Nacional de Desenvolvimento Científico e Tecnológico

COO *Coordinate Format*

CR *Conjugate Residual*

CPU Unidade Central de Processamento

CSR *Compressed Sparse Row Format*

CUDA *Compute Unified Device Architecture*

DIA *Diagonal*

DLP *Data-Level Parallelism*

DRAM *Dynamic Random Access Memory*

GMRES *Generalized Minimum Residual*

GPU Unidade de Processamento Gráfico

HYB *Hybrid Format*

IC *Incomplete Cholesky Factorization*

ILU *Incomplete LU Factorization*

LMAG Laboratório de Eletromagnetismo Aplicado da USP

MEF Método de Elementos Finitos

MIMD *Multiple Instruction Multiple Data*

MISD *Multiple Instruction Single Data*

MPI *Message Passing Interface*

MPP *Massively Parallel Processing*

OpenCL *Open Computing Language*

OpenMP *Open Multi-Processing*

PML *Perfectly Matched Layer*

RCM *Reverse Cuthill-McKee Ordering*

RISC *Reduced Instruction Set Computer*

SIMD *Single Instruction Multiple Data*

SIMT *Single Instruction Multiple Thread*

SISD *Single Instruction Single Data*

SPMD *Single Program Multiple Data*

TLP *Thread-Level Parallelism*

Sumário

1	Introdução	19
1.1	Motivação	19
1.2	Objetivos e Metas	22
1.2.1	Objetivos Específicos	22
1.2.2	Metas	23
1.3	Organização do Documento	23
2	Estado da Arte	25
2.1	Introdução	25
2.2	Revisão Bibliográfica	25
2.3	Originalidade do Trabalho	30
2.4	Síntese do Capítulo	31
3	Processamento Paralelo	33
3.1	Introdução	33
3.2	Tecnologia de Computação	33
3.3	Classes de Paralelismo e Arquiteturas Paralelas	34
3.4	Unidade de Processamento Gráfico	36
3.4.1	Performance da Tecnologia GPU, <i>Multi-core</i> e CPU	36
3.4.2	Arquitetura da GPU	38
3.4.3	Linguagens e Modelos de Programação Paralela	39
3.4.4	Estrutura de um Programa CUDA	41
3.4.5	Memória do <i>Device</i> e Transferência de Dados	43
3.4.6	Tecnologia NVIDIA <i>GPUDirect</i>	45
3.5	Tecnologias e Relação Custo-Benefício	46
3.6	Síntese do Capítulo	48

4	Método de Elementos Finitos no Eletromagnetismo	49
4.1	Introdução	49
4.2	Formulação para o Regime Estático	49
4.3	Formulação para o Regime Harmônico	52
4.4	Síntese do Capítulo	55
5	Métodos Iterativos de Solução de Sistemas Lineares	57
5.1	Introdução	57
5.2	Métodos de Solução do Sistema de Equações Algébricas	57
5.2.1	Método do Gradiente Conjugado	59
5.2.2	Método Conjugado Residual	60
5.2.3	Método Gradiente Biconjugado	61
5.2.4	Método Gradiente Conjugado ao Quadrado	63
5.2.5	Método Gradiente Biconjugado Estabilizado	65
5.3	Pré-condicionadores	66
5.3.1	Decomposição Incompleta de Cholesky	69
5.3.2	Decomposição Incompleta LU	70
5.3.3	Método do Gradiente Conjugado Pré-condicionado	71
5.4	Método de Reordenação Cuthill-McKee Reverso	72
5.5	Formato de Acesso e Manipulação das Matrizes	72
5.6	Bibliotecas de Álgebra Linear	74
5.7	Síntese do Capítulo	74
6	Metodologias e Definição dos Casos-teste	75
6.1	Introdução	75
6.2	Metodologia dos Casos-teste	75
6.2.1	Principais Etapas Computacionais	75
6.2.2	Estrutura do Algoritmo Paralelo	77
6.2.3	Reordenação Cuthill-McKee Reverso	79
6.3	Configurações de Hardware	80
6.4	Detalhes da Implementação em CUDA	80
6.5	Configurações Utilizadas nos Casos-teste	82
6.5.1	Configuração 1 - Regime Estacionário	84
6.5.2	Configuração 2 - Regime Harmônico	85
6.6	Métrica de Desempenho	86
6.6.1	<i>Speedup</i>	86
6.7	Síntese do Capítulo	86

7	Resultados	87
7.1	Introdução	87
7.2	Detalhamento dos Métodos Utilizados nos Casos-teste	87
7.3	Estudo de Caso 1	88
7.4	Estudo de Caso 2	90
7.5	Estudo de Caso 3	92
7.6	Estudo de Caso 4	94
7.7	Estudo de Caso 5	99
7.8	Estudo de Caso 6	105
7.9	Estudo de Caso 7	107
7.10	Estudo de Caso 8	111
7.11	Estudo de Caso 9	113
7.12	Estudo de Caso 10	115
7.13	Outros Resultados	115
7.14	Síntese do Capítulo	116
8	Conclusões	117
8.1	Síntese do Trabalho	117
8.2	Principais Dificuldades	118
8.3	Principais Contribuições	119
8.4	Perspectivas e Trabalhos Futuros	120
	Referências Bibliográficas	121
	Apêndice A - Artigo Submetido e em Análise	127

Introdução

1.1 Motivação

Até meados da década de 40, problemas de Eletromagnetismo eram resolvidos pela solução analítica aproximada das equações a derivadas parciais que governam os fenômenos envolvidos. Esse procedimento só permitia o tratamento de problemas homogêneos, com geometria simples e meios contínuos. A partir dessa década, vários métodos numéricos foram propostos para tentar amenizar essas dificuldades, dentre eles pode-se destacar o método de Elementos Finitos (MEF). Esse método foi proposto em 1940, porém sua utilização ocorreu somente a partir de 1950 (JIN, 1993). Inicialmente, ele foi aplicado em problemas de análises estruturais e após sua generalização e aceitação ele foi muito utilizado nos mais diversos estudos, sendo reconhecido como um método geral e amplamente aplicado a problemas da engenharia e da matemática. Na área de cálculo de campos eletromagnéticos o MEF somente foi utilizado a partir de 1968 (SADIKU, 2000). Devido à sua flexibilidade e estabilidade numérica ele pode ser facilmente implementado em um sistema computacional, fato que explica sua grande aplicação em várias áreas atualmente. Assim, para utilizar esse método é preciso discretizar o domínio do problema, selecionar a função de interpolação, encontrar o sistema de equações algébricas o qual pode ser obtido através do método Variacional ou do método de Galerkin, e finalmente resolver esse sistema de equações resultantes (JIN, 1993).

Com a grande aplicação do método, vários problemas puderam ser solucionados computacionalmente, principalmente devido à proliferação dos microcomputadores ocorrida no final da década de 80. Entretanto, ao se resolver problemas mais realísticos com geometrias complexas, múltiplos meios, domínios 3D, materiais não lineares e variação no tempo, esse método pode gerar um sistema matricial mal condicionado e com ordens muito elevadas. Dependendo do problema físico e desses fatores limitadores a resolução do sistema de equações algébricas pode demorar até alguns dias de execução. Assim, de uma lado tem-se uma ótima ferramenta computacional de resolução de problemas realísticos, porém de outro lado, tem-se o limite computacional para realizar diversos cálculos matemáticos com um grande volume de tarefas e

dados.

Desde o início, a indústria de computadores tem realizado uma pesquisa contínua em busca de crescente poder computacional. Nos anos 50 e 60, as tecnologias de fabricação das máquinas monoprocessadas pareciam ser suficientes para atender à demanda daquela época. Os microprocessadores baseados em um única unidade central de processamento (CPU) impulsionaram aumentos de desempenho e reduções de custo nas aplicações de computador por mais de duas décadas (KIRK; HWU, 2010). Com a possibilidade de trabalhar com giga operações de ponto flutuante por segundo (GFLOP/s) e com o contínuo impulso de melhoria de desempenho, os *softwares* de aplicação passaram a oferecer mais funcionalidade e melhores interfaces com o usuário, além de gerar resultados mais precisos. Porém, a partir dos anos 70, as necessidades crescentes de capacidade de processamento não eram mais atendidas pelas evoluções na tecnologia dos monoprocessadores. Assim, com usuários cada vez mais exigentes, uma demanda crescente de melhorias e mais capacidade computacional foram exigidas, o que possibilitou um ciclo virtuoso para a indústria de computadores.

Esse processo de busca de melhorias está relacionado aos avanços no *hardware* o qual permite aumentar a velocidade das aplicações, ou seja, o *software* é executado mais rápido à medida que nova geração de processadores é lançada. Entretanto, existe uma limitação nestas melhorias, porque desde 2003 as questões do consumo de energia e dissipação de calor tem sido um grande limitador no aumento da frequência de *clock* e no nível de tarefas realizadas em cada período, em um única CPU. Assim, ocorreu um esforço ainda maior da comunidade de desenvolvimento de *software* para melhorar o poder de processamento. Foi nesta época que surgiram as várias unidades de processamento, conhecidas como núcleos de processamento (KIRK; HWU, 2010).

Tradicionalmente, a grande maioria das aplicações de *software* são escritas de forma sequencial e os desenvolvedores de aplicação tiveram que introduzir novos recursos e capacidades em seus *softwares* à medida que novos microprocessadores foram desenvolvidos. Assim, outras possibilidades foram surgindo para os desenvolvedores, como por exemplo, trabalhar com programas paralelos, onde várias *threads* de execução cooperam para completar o trabalho total em um tempo menor. Com a introdução de novos microprocessadores naquela época, o número de aplicações desenvolvidas utilizando programas paralelos cresceu rapidamente. Assim, duas trajetórias na indústria de semicondutores foram estabelecidas desde 2003. A primeira utiliza os processadores com múltiplos núcleos (*multicore*) que foram inicialmente criados com dois núcleos, sendo que esse número tem praticamente dobrado a cada nova geração. A segunda trajetória baseou-se em vários núcleos (*many-core*) que permitiu a criação da unidade de processamento gráfico (GPU), sendo que cada núcleo é maciçamente *multithreaded*. Esses processadores têm liderado a corrida do desempenho em ponto flutuante desde 2003, em termos de melhoria de desempenho. Em 2009, a razão entre GPUs e CPUs era cerca de 10 para 1, em termos de cálculo de ponto flutuante. A filosofia das trajetórias foram bastante diferentes. Na primeira, a arquitetura da CPU foi otimizada para o desenvolvimento de código sequencial com

grandes memórias cache, enquanto que na segunda utiliza-se modelos de memória mais simples com uma largura de banda de memória alta e código maciçamente paralelo. Neste segundo caso, os fabricantes procuraram maneiras de maximizar a área do chip e o consumo de energia dedicados aos cálculos de ponto flutuante (KIRK; HWU, 2010).

Essa grande diferença em desempenho e os preços mais acessíveis dos processadores com vários núcleos motivaram muitos desenvolvedores a mudarem suas trajetórias computacionais, optando pela segunda. Assim, cabe ao desenvolvedor/pesquisador definir quais partes do programa deve ser executado de forma sequencial e quais de forma paralela. A maioria das aplicações utiliza tanto a CPU quanto a GPU. A Figura 1.1 ilustra a evolução do desempenho computacional da CPU versus GPU, em termos de GFLOP/s.

Figura 1.1 – Histórico da evolução dos desempenhos computacionais da CPU e GPU em GFlops/s

Fonte: NVIDIA, 2014

Embora o ritmo de melhoria de desempenho dos microprocessadores da CPU tenha exibido poucas evoluções nestes últimos anos, os da GPUs têm melhorado intensamente. Analisando a Figura 1.1 pode-se perceber que de 2011 até o momento o número de GFLOP/s nas GPUs praticamente triplicou. Por um lado a facilidade de obter processadores cada vez melhores permite desenvolvimentos cada vez mais rápidos, entretanto, por outro lado essas tecnologias novas obrigam os desenvolvedores/pesquisadores a inovarem em seus programas e a atualizarem seus recursos com as possibilidades que as novas tecnologias oferecem.

A NVIDIA lançou em 2007 a plataforma de computação paralela e o modelo de programação conhecido como CUDA (*Compute Unified Device Architecture*) (NVIDIA, 2014). Essa arquitetura facilita a programação paralela e permite que as ferramentas de programação C/C++ sejam utilizadas, além de possibilitar que os usuários utilizem a GPU com mais facilidade, contribuindo para uma diminuição da complexidade de manipulação dos dados na placa de vídeo. Todas essas vantagens possibilita que os desenvolvedores/pesquisadores aumentem ainda mais o poder de processamento de suas aplicações, sem a necessidade de conhecer profundamente os detalhes do *hardware*. A NVIDIA também desenvolveu o compilador CUDA C/C++ e Fortran com bibliotecas e *softwares* de *runtime* para possibilitar que os programadores acessem rapidamente o novo modelo de programação paralela de dados (KIRK; HWU, 2010).

Portanto, a crescente evolução na área de computação proporcionou um avanço tecnológico em várias áreas do conhecimento, como por exemplo a Engenharia. Isso possibilitou que os métodos numéricos ganhassem cada vez mais aplicação, principalmente em modelagem de problemas reais e tridimensionais. O avanço somente foi possível devido às pesquisas por melhores resultados, por processadores cada vez mais velozes, por um conjunto de inovações que puderam fornecer produtos cada vez melhores e com um alto desempenho computacional. Nesse contexto, insere-se então a motivação principal deste trabalho, que é a de incorporar esses recentes avanços tecnológicos no desenvolvimento de ferramentas computacionais que tragam todas as potencialidades da computação paralela, aliadas a uma plataforma acessível, a pequenos grupos de pesquisa ou a pesquisadores individuais envolvidos com o cálculo computacional de campos eletromagnéticos. A proposta será a de solucionar problemas de larga escala, oriundos da modelagem por elementos finitos dos fenômenos envolvendo a difusão tridimensional de correntes elétricas em meios abertos e dissipativos, mais especificamente, eletrocinética e propagação de ondas em meios muito dissipativos. Uma aplicação prática imediata dessa modelagem é a análise de problemas de aterramento elétrico, seja no regime estático, seja no dinâmico, sendo este último em particular, aquele que tem oferecido os maiores desafios em termos de complexidade e exigência computacional.

1.2 Objetivos e Metas

Este trabalho tem como objetivo principal apresentar um conjunto de ferramentas de resolução do sistema de equações algébricas lineares, oriundas da simulação de fenômenos eletromagnéticos pelo MEF, que utiliza recursos da computação paralela maciça em computadores pessoais, especialmente GPU e programação CUDA. Os sistemas lineares são aqueles procedentes da aplicação do método de Elementos Finitos na modelagem do fenômeno da difusão de correntes elétricas em meios abertos e dissipativos nos regimes estacionário e harmônico.

1.2.1 Objetivos Específicos

Os objetivos específicos são os seguintes:

- ❑ implementação em OpenMP e CUDA das versões paralelas dos métodos de solução do sistema linear, como por exemplo, Gradiente Conjugado, Gradiente Biconjugado, Gradiente Biconjugado Estabilizado, Conjugado Residual, Gradiente Conjugado Quadrado, dentre outros;
- ❑ implementação das técnicas de programação paralela, OpenMP e CUDA, nos computadores equipados com processadores *multi-core* e *many-core*, repectivamente;
- ❑ utilização e testes das bibliotecas de álgebra linear disponibilizadas pela NVIDIA: CUSP, CUSPARSE e CUBLAS (NVIDIA, 2014);
- ❑ aplicação dos algoritmos desenvolvidos na modelagem pelo MEF de um exemplo prático de problema da Engenharia Elétrica, caso da análise de sistemas de aterramento elétrico;
- ❑ utilização do modelo de programação paralela CUDA em processadores *many-core* para plataformas do tipo *Single-GPU* e *Multi-GPU*;
- ❑ implementação em CUDA de alguns pré-condicionadores;
- ❑ validação de todos os algoritmos paralelos desenvolvidos na pesquisa.

1.2.2 Metas

Destaca-se como meta principal solucionar problemas de aterramento elétrico em larga escala e analisar os pontos positivos e negativos da implementação paralela para avaliar as métricas de desempenho computacional. As metas específicas são as listadas a seguir:

- ❑ implementação das ferramentas computacionais para solucionar os sistemas a coeficientes reais e complexos, oriundos do Método de Elementos Finitos;
- ❑ desenvolvimento dos casos-teste e análise qualitativa dos resultados;
- ❑ obtenção de ganho computacional.

1.3 Organização do Documento

O trabalho foi dividido em oito capítulos além das Referências Bibliográficas e Apêndices.

O Capítulo 2 apresenta o estado da arte da Computação Paralela, da resolução de sistemas de equações algébricas lineares por métodos iterativos e seu uso no contexto do MEF.

O Capítulo 3 introduz os principais conceitos de computação paralela para desenvolver um programa paralelo utilizando OpenMP, CUDA e *GPUDirect*.

O Capítulo 4 aborda sucintamente as principais formulações do MEF, nos regimes estático e harmônico, e os seus respectivos sistemas de equações resultantes, cujas soluções representam uma das principais motivações do presente trabalho.

O Capítulo 5 introduz os principais métodos iterativos de solução de sistemas lineares, aqueles mais adequados e utilizados no contexto do MEF em geral e das aplicações particulares que serão abordadas. Também serão discutidas as técnicas de pré-condicionamento aplicáveis.

O Capítulo 6 detalha os problemas que serão usados como casos-teste das rotinas desenvolvidas e os métodos de solução e algoritmos que foram implementados.

O Capítulo 7 mostra os resultados obtidos, bem como análise dos mesmos.

O Capítulo 8 introduz uma síntese do trabalho com as conclusões finais, dificuldades enfrentadas, desafios e perspectivas e encaminhamento para trabalhos futuros.

No Apêndice A é apresentado um artigo que contém resultados adicionais o qual foi submetido em periódico indexado e ainda está sob análise no momento da finalização deste documento.

Estado da Arte

2.1 Introdução

Neste capítulo são abordadas as principais publicações sobre processamento paralelo em GPU aplicado na resolução de sistemas lineares oriundos do método de Elementos Finitos. Ao final, será feito um breve paralelo entre esses trabalhos e o que se pretende propor.

2.2 Revisão Bibliográfica

O método do Gradiente Conjugado (CG) foi proposto por Hestenes e Stiefel (1952) e tem uma história longa e bem sucedida na matemática computacional, assim, como outros métodos do subespaço Krylov (SHEWCHUK, 1994), (VAN DER SLUIS; VAN DER VORST, 1986). Saad (2003) escreveu sobre a importância dos pré-condicionadores e abordou vários métodos iterativos que podem ser implementados de forma paralela ao utilizar memória compartilhada e memória distribuída. Ele enfatizou que a computação paralela está proporcionando um desempenho satisfatório em várias áreas científicas.

Ida e Wang (1988), Barton e Rattner (1992) e Vollaire et al. (1998) escreveram sobre a importância do uso da computação paralela na implementação do método de Elementos Finitos na área de cálculo de campos eletromagnéticos. Eles afirmam que o uso da computação paralela é muito vantajoso para resolver problemas práticos e reais em tempos menores de execução. Especialmente, Barton e Rattner (1992) disseram da revolução que a computação, naquela época, proporcionou para a ciência e engenharia. Descreveu que antigamente o computador era muito utilizado como auxiliar da experimentação no desenvolvimento e verificação da teoria científica e análise da engenharia. Também relatou que as simulações de computadores irão substituir grande parte da experimentação em um amplo espectro de aplicações, e que as razões para isso é o simultâneo aumento no desempenho e na redução do custo para realizar grandes cálculos computacionalmente. Afirmou ainda que a computação paralela permite a solução de vários problemas mais ambiciosos. Segundo Saad (2003) as primeiras considerações de alto desem-

penho para implementações de métodos iterativos foram obtidas em implementações ao usar computadores vetoriais. Estes esforços foram iniciados em meados dos anos 1970, quando os computadores vetoriais apareceram pela primeira vez.

Ainda segundo esses autores, a computação paralela está sendo utilizada há muitos anos na área de Eletromagnetismo, porém nos últimos dez anos é que está sendo introduzida efetivamente para resolver diversas classes de problemas. Essa interdisciplinaridade (computação, matemática e engenharia elétrica) é muito importante porque são exigidos cada vez mais avanços tecnológicos e maior desempenho computacional.

De acordo com Saad (2003), a computação paralela está se tornando rapidamente uma alternativa barata para resolver grandes problemas que surgem em pesquisas científicas e nas aplicações da engenharia. Para ele existem duas alternativas tradicionais para desenvolver técnicas iterativas paralelas. A primeira consiste em analisar o algoritmo e ver a possibilidade de aplicar paralelismo em alguma parte. A segunda é desenvolver algoritmos alternativos com paralelismo. Esse autor começou a aplicar técnicas paralelas naquela época e continuou nesse caminho para descobrir e investir em novas tecnologias oriundas da computação. Recentemente, Li e Saad (2013) publicaram um trabalho sobre computação paralela em GPU para resolução das equações algébricas e solução de sistemas lineares esparsos. Eles descrevem as vantagens de trabalhar com GPU, especialmente com placas da NVIDIA e com o modelo de programação CUDA.

Pelas publicações em revistas indexadas, pode-se inferir que nos últimos 10 anos a técnica mais utilizada pelos pesquisadores, na área de Eletromagnetismo e Elementos Finitos, é a de decomposição de domínio com uso de *cluster* e troca de mensagens (MPI). Porém, desde 2008, existe uma grande tendência para o uso de outras técnicas novas ao utilizar computadores *multi-core* e GPU (CUDA). À medida que a área da computação avança em termos de desenvolvimento/processador e desempenho computacional, as outras áreas científicas tendem a acompanhar esse evolução. Owens et al. (2008) descreveram que a computação em GPU tornou-se uma tendência importante na computação paralela devido às vantagens em relação aos computadores paralelos tradicionais (*clusters*), tais como: maior desempenho computacional, menor consumo e custo. Neste sentido, essas três importantes características motivaram as comunidades científicas a investigarem essa nova tecnologia, utilizando GPUs e CUDA.

Assim, alguns trabalhos publicados na área foram relacionados e seus resultados abordados com mais detalhe. Por exemplo, Komatitsch et al. (2009) utilizaram o método de Elementos Finitos em uma simulação numérica de propagação de ondas causada por terremoto na Terra, utilizando placas gráficas NVIDIA GeForce 8800 GTX e GTX 280. Os autores utilizaram três configurações de malhas: uma de baixa resolução de 65 MB de memória, uma de média resolução de 405 MB, e uma terceira de alta resolução com 633 MB. As três malhas contêm, respectivamente, em termos de elementos e nós: (7424, 496525), (46592, 3061529) e (72800, 4769577). Fizeram a medição de velocidade para duas versões de código: (i) inteiramente na GPU; e (ii) compartilhado CPU/ GPU. Concluíram que os valores variam muito pouco com o

tamanho da malha. O *speedup* máximo obtido foi de 25 vezes na GPU GTX 280 e 15 vezes na 8800 GTX para a primeira versão do programa, e 5,3 vezes na GPU 8800 GTX e 7,1 vezes na GTX 280 para a segunda versão. Constataram o poder computacional da GPU, concluindo que a última versão é significativamente mais lenta que a primeira.

Dehnavi et al. (2010) afirmam que o método do Gradiente Conjugado implementado em um único núcleo de processamento proporciona um grande tempo de execução, e que o uso de técnicas eficientes é desejável para melhorar o desempenho da utilização dos métodos iterativos. Concluíram que a multiplicação matriz-vetor consome muito tempo computacional no método do Gradiente Conjugado e que as unidades de processamento gráfico têm sido muito atrativas para processamento de dados em paralelo. Afirmaram ainda que as interfaces de programação CUDA permitem a utilização de vários núcleos na GPU, proporcionando um custo-benefício bastante atrativo.

Li et al. (2010) explicam que os métodos iterativos são limitados em desempenho quando simulados em um único núcleo de processamento e que técnicas paralelas tornaram-se fundamentais para obter aumento contínuo no desempenho computacional. Explicaram que *clusters* de computadores oferecem significativo poder computacional, entretanto, são caros, consomem energia em excesso e podem se tornar limitados em sua expansão. Além disso, disseram que novas arquiteturas de hardware, como GPU, são promissoras. Disseram que cada sistema é composto de centenas e até milhares de processadores de alto desempenho, e que uma GPU moderna pode ser equivalente a um *cluster* de médio alcance, em termos de desempenho computacional. Afirmam também, que a agregação de *cluster* e GPU pode ter um desempenho ainda maior. Como resultados, eles demonstraram que o tempo computacional com GPU é 100 vezes menor em relação aos códigos desenvolvidos com CPU sequencial.

Godel et al. (2010) desenvolveram um trabalho com a técnica de decomposição de domínio e múltiplas GPUs. O domínio foi particionado em 8 CPUs (*quad-core*) e então, cada bloco foi enviado a uma determinada GPU (2 placas de vídeo Tesla S 1070 com 4 GPUs cada). Esses autores dizem que a largura de banda de memória em GPUs é muito maior que em sistemas de CPU. Eles fizeram uma simulação com 8 GPUs e uma simulação com uma CPU contendo 28 núcleos e constataram que a primeira simulação é bem mais rápida que a segunda. Concluíram que os computadores convencionais utilizados (CPUs) são mais caros e consomem mais energia quando comparados com as GPUs.

Nesta linha de implementação, Kakay et al. (2010) fizeram uma comparação dos códigos CPU e GPU. Utilizaram uma máquina com 8 núcleos e uma GPU (NVIDIA GTX 285, com 240 núcleos de processadores) e constaram o desempenho da GPU em relação a máquina *multi-core*. Para estudos micromagnéticos, esses autores concluíram que a arquitetura paralela do sistema de GPU é muito importante porque: (i) possibilita a resolução deste tipo de problema; (ii) a velocidade total de cálculo da GPU pode exceder um fator de quatro quando comparada à paralelização utilizando 8 núcleos e; (iii) a redução no tempo computacional possibilita estudos da dinâmica da magnetização em amostras de grandes dimensões, com alta resolução espacial.

Estes autores concluíram que o tempo computacional do código na GPU aumenta quase que linearmente.

Godel et al. (2010) realizaram um trabalho e utilizaram a arquitetura GPU NVIDIA GT 200, fornecendo 30 multiprocessadores por GPU. Implementaram e compararam uma arquitetura de 4 GPUs com uma de 4 CPUs (*quad-core*), bem como com uma agregando CPU e GPU. Para múltiplas GPUs os autores utilizaram o software METIS para fazer a decomposição de domínio (KARYPIS; KUMAR, 1998). Desenvolveram três implementações de arquitetura (CPU, GPU e CPU/GPU). Foram considerados dois exemplos para visualização do desempenho computacional. Esses autores concluíram, para o primeiro exemplo, que a simulação com GPU é 8,02 vezes mais rápida que com a arquitetura CPU, e que por sua vez, é superada em 1,22 vezes quando se aplica CPU e GPU concomitantemente. Já para o segundo exemplo, a arquitetura CPU/GPU é 33,48 vezes mais rápida que a arquitetura CPU. Mais uma vez foi demonstrado a importância e a grande aplicabilidade do uso da arquitetura GPU na solução de problemas da área de cálculo de campos eletromagnéticos.

Ren et al. (2010) e Xu et al. (2010) desenvolveram suas pesquisas utilizando MPI e GPU e concluíram que é bastante viável a utilização dessas arquiteturas e que o tempo de processamento da GPU em relação ao cluster é bem menor. Já os autores Ikuno et al. (2010) concluíram o quanto a GPU é mais rápida quando comparada com a implementação CPU, a uma taxa de 22,53 vezes. Nessa mesma linha de pesquisa, porém com uma única CPU, Fernandez et al. (2011) aplicaram a técnica em um problema eletrostático clássico, em duas dimensões, para validar o método proposto. Implementaram um código utilizando GPU NVIDIA GeForce 8800 GT. Obtiveram um aumento de velocidade de até 13 vezes em relação a uma versão sequencial executada em uma CPU, demonstrando o potencial do método proposto quando se utiliza vários núcleos paralelos. Dziekonski et al. (2011) fizeram uma implementação paralela do método do Gradiente Conjugado pré-condicionado utilizando o método de Jacobi para problemas de microondas. Fizeram uma comparação da arquitetura paralela utilizando GPU com uma unidade de processamento (*quad-core*) e também concluíram a eficiência da GPU em relação a um computador.

Dehnavi et al. (2011) implementaram o PCG utilizando GPU. Propuseram técnicas para melhorar o desempenho em relação a outros trabalhos, independente da geração GPU e do padrão de esparsidade da matriz. Eles utilizaram sete matrizes oriundas da Universidade da Flórida (*Sparse Matrix Collection*), com diferentes padrões de esparsidade e áreas de aplicação. As otimizações propostas foram desenvolvidas com as placas de vídeo da NVIDIA (G80 e GT 200). Os autores obtiveram um aumento de desempenho de até 3,4 e 2,5 vezes, respectivamente, em comparação com métodos anteriores propostos. Também obtiveram para o PCG um aumento em desempenho de até 23 vezes em relação à versão otimizada em CPUs modernas.

Fernandez et al. (2012) apresentaram uma nova formulação para o Método de Elementos Finitos utilizando processamento paralelo, baseado na solução de elementos de malha (FEM-SES). A ideia principal foi desacoplar a solução, realizando o paralelismo no nível do elemento.

O problema solucionado foi eletrostático, clássico 2D (cabo coaxial), que foi utilizado para validar o novo método proposto e para obter resultados e estudar o comportamento da convergência. Duas gerações de placas de vídeo da NVIDIA, habilitadas com CUDA, foram utilizadas para implementar o método FEM-SES. A primeira foi a GPU 8800 GT e a segunda foi a GTX 480. Os tempos de execução foram comparados com o MEF clássico. Todos os cálculos foram realizados em precisão simples, sendo que as GPUs utilizadas tiveram certa limitação ao trabalhar com dupla precisão. Como resultado, os autores obtiveram um *speedup* de 14 vezes na GPU 8800 GT e 111 vezes na GTX 480 ao comparar os resultados com a CPU (MEF clássico).

Rodrigues et al. (2012) explicaram que a programação paralela exige uma distribuição não trivial de tarefas e dados e que os desenvolvedores têm dificuldade para implementar seus aplicativos de forma eficaz. Assim, a fim de reduzir a complexidade do projeto, eles propuseram uma abordagem para gerar o código em arquiteturas híbridas (por exemplo, CPU x GPU) usando OpenCL (padrão aberto de programação paralela de sistemas heterogêneos). O objetivo deles foi fornecer recursos para os não especialistas em programação paralela ao implementar suas aplicações. Além do desenvolvimento de software em abstrações de alto nível, eles propuseram melhorar o desempenho usando ambientes com *Multi-GPU*. Eles implementaram quatro versões do método do Gradiente Conjugado, em precisão dupla. A primeira (utilizada como referência), é sequencial e desenvolvida em MATLAB. As outras são implementações em paralelo as quais utilizam OpenCL e GPUs: 1, 2 e 4, respectivamente. Eles obtiveram um *speedup* de 4,81 para 1 GPU, de 6,87 para 2 GPUs e de 8,34 para 4 GPUs. Concluíram que o ganho não é linear (embora significativo), devido à transferência de dados entre CPU e dispositivos. Alguns autores desse último trabalho também publicaram em 2013 e utilizaram GPU. Eles também queriam se beneficiar do excelente desempenho que podem obter com a programação paralela. Eles concluíram que a engenharia de software leva a um desenvolvimento de aplicativos mais rápido e mais seguros, além de permitir uma programação de alto nível para o programador. Eles utilizaram GPU para aumentar o desempenho computacional daqueles problemas maiores de simulação de máquinas elétricas. Eles obtiveram um *speedup* na ordem de 10 comparado à CPU (RODRIGUES et al., 2013).

Ikuno et al. (2012) implementaram um pré-condicionar variável (VP) utilizando o subespaço Krylov, com precisão mista. Eles utilizaram 1 GPU e *Multi-GPU* (GTX 480 da NVIDIA) e o sistema operacional Linux. Eles implementaram os métodos do Gradiente Conjugado, Conjugado Residual (CR) e o Método do Resíduo Mínimo Generalizado (GMRES), utilizando 1 GPU, *Multi-GPU* com MPI e *Multi-GPU* com a comunicação *peer-to-peer*. Concluíram que os tempos de execução da aplicação *Multi-GPU* são mais rápidos que os obtidos com 1 GPU. Além disso, afirmaram que o tempo de execução da comunicação *peer-to-peer* melhorou comparado ao MPI. No entanto, a melhoria significativa não foi obtida por meio da técnica *Multi-GPU*. Isto aconteceu devido ao grande volume de dados que são copiados ou devido à grande comunicação de dados que ocorre ao utilizar *Multi-GPU*. A maior parte do tempo de execução foi utilizado para copiar dados ou na própria comunicação. Concluíram também que a taxa de transmissão

entre as GPUs é o gargalo do desempenho. Para o método VP-CG, o tempo em 1 GPU foi de 23,65 segundos contra 19,56 segundos na *Multi-GPU*. Para o método VP-CR o tempo em 1 GPU foi de 125,94 segundos contra 97,63 segundos na *Multi-GPU*. Eles determinaram que o VP-CG-JOR (Jacobi) em GPU com precisão mista é 41,85 vezes mais rápido que o VP-CG-CG na CPU.

Seguindo essa mesma linha de implementação, Okimura et al. (2013) abordaram um problema cuja análise de campo magnético é não linear. Utilizaram o MEF e GPU NVIDIA. Os métodos implementados foram o JOR, CR e CG os quais foram aplicados junto ao pré-condicionador variável (VP). Os resultados dos cálculos demonstraram que o VP-CG na GPU melhorou significativamente o desempenho computacional. Concluíram que o método CG foi 39 vezes mais rápido na GPU que o método CG junto com a decomposição incompleta de Cholesky (ICCG), em uma CPU. Concluíram que a convergência se torna pior quando se tem matrizes com ordem elevadas. O *speedup* foi de 39, 19, e 14, para os métodos VP-CG-CG, VP-CG-CR e VP-CG-JOR, respectivamente. Também concluíram que o *speedup* foi elevado considerando o critério de convergência inferior a 0,01. O método VP-CG-JOR mostrou uma lenta convergência na análise dos campos magnéticos do material ferromagnético, quando a matriz dos coeficientes não era diagonalmente dominante.

Por fim, Richter et al. (2014) fizeram uma simulação de problemas eletromagnéticos/térmicos. Enfatizaram que CUDA auxilia na utilização de recursos paralelos das GPUs para acelerar os cálculos numéricos. Isso reduz o tempo necessário para resolver problemas reais. Neste trabalho, os autores implementaram um problema regido pela equação de Poisson e utilizaram o pré-condicionador *Multigrid* acelerado por GPU, disponível na biblioteca CUSP da NVIDIA (CUSP, 2014). Eles compararam a programação OpenMP com a CUDA e utilizaram a arquitetura *Kepler*, da NVIDIA. Eles utilizaram 1, 4 e 8 núcleos de processadores, além da GPU. Neste caso, o melhor tempo de processamento foi obtido para 4 núcleos. Eles obtiveram uma aceleração de 5,2 para o *Multigrid* na GPU, comparado com o mesmo código executado na CPU.

2.3 Originalidade do Trabalho

Dos trabalhos citados acima, todos eles tiveram como foco a utilização de processamento paralelo em GPU na resolução de sistemas do MEF. Pode-se observar a atualidade do tema, bem como a sua relevância. Em praticamente todas as publicações, utilizou-se as placas de vídeo da NVIDIA e a linguagem de programação CUDA. Entretanto, a maioria deles aborda problemas no regime estático e no regime quase-estático e, em praticamente todos os trabalhos, utilizou-se o método CG. Para aqueles trabalhos cujos problemas estão no domínio do tempo, Dziekonski et al. (2011) resolveram um sistema complexo, oriundo da aplicação do MEF de aresta, porém não a formulação em dois potenciais, e numa aplicação de microondas. Ikuno et al. (2012) usaram a formulação *A-V* de aresta, porém resolveram um problema quasi-estático

no domínio do tempo, simulando um dispositivo eletromagnético não linear.

Até o momento não foi possível identificar um trabalho que apresentasse uma solução em GPU para o sistema de equações derivado da aplicação da formulação A - V , aresta-nodal, do Método de Elementos Finitos, aplicado à solução da Equação de Onda completa não homogênea; nem um método adequado de resolução desse sistema. A maioria dos métodos citados por aqueles autores não foram suficientes para resolver o sistema que desejamos, em particular os pré-condicionadores, e sobretudo aqueles que mais se prestam ao paralelismo.

2.4 Síntese do Capítulo

Os principais trabalhos publicados utilizam processamento paralelo em GPU na resolução de equações algébricas. Demonstrou-se que em quase todos os casos, os problemas solucionados foram regidos no regime estacionário e que a maioria dos trabalhos foram desenvolvidos utilizando o método do Gradiente Conjugado. Também em todos os casos, observou-se que o uso de GPU na resolução de métodos iterativos tem ganhado muito destaque principalmente devido ao custo-benefício e ao desempenho computacional comparado às CPUs. Por fim, foi apresentada uma contextualização deste trabalho e as diferenças entre ele e os demais, apontando a sua contribuição. No próximo capítulo as principais características do processamento paralelo em GPU serão apresentadas.

Processamento Paralelo

3.1 Introdução

Neste capítulo são abordados os principais aspectos sobre Processamento Paralelo. Serão introduzidas as classes de paralelismo, as arquiteturas paralelas, as linguagens de programação, a hierarquia de memória, dentre outras abordagens. Além disso, o enfoque será maior para as unidades de processamento gráfico, que é o foco deste trabalho.

3.2 Tecnologia de Computação

A tecnologia de computação teve um progresso muito grande nos últimos 65 anos, desde que foi criado o primeiro computador eletrônico de uso geral (PATTERSON; HENNESSY, 2014). Essa rápida evolução deveu-se aos avanços na tecnologia usada para montar os computadores e na inovação dos projeto (PATTERSON; HENNESSY, 2014). Assim, durante os 25 anos de existência dos computadores eletrônicos, houve uma melhoria no desempenho em cerca de 25% por ano. Neste contexto, no final da década de 1970, ocorreu o surgimento do microprocessador que teve a capacidade de acompanhar as melhorias na tecnologia do circuito integrado e que permitiu que a taxa de desempenho aumentasse ainda mais, aproximadamente 35% por ano. No mercado atual estão disponíveis computadores pessoais a preços acessíveis, de maior desempenho e maior quantidade de memória e armazenamento em disco, comparado àqueles computadores adquiridos na década de 80.

A taxa de crescimento combinada com as vantagens do custo de um microprocessador facilitaram uma grande demanda por computadores com uma nova arquitetura. No início de 1980, houve um desenvolvimento bem sucedido de um novo conjunto de arquiteturas com instruções mais simples, conhecidas como RISC (*Reduced Instruction Set Computer*) que são os computadores com conjunto de instruções reduzidas. As máquinas baseadas nessa arquitetura tiveram grande aceitação principalmente devido ao paralelismo em nível de instrução e uso de *caches*. Assim, houve uma maximização do desempenho e as arquiteturas anteriores tiveram

que acompanhar esse padrão. A combinação de melhorias na organização e na arquitetura dos computadores fez com que o crescimento fosse constante e bem alto durante 17 anos, com uma taxa anual de mais de 50%. De acordo, com Patterson e Hennessy (2014), essa notável taxa de crescimento impactou principalmente na melhoria:

- ❑ considerável na capacidade disponível aos usuários de computador;
- ❑ do custo-desempenho;
- ❑ da contínua fabricação de semicondutores que levou à dominância de computadores baseados em microprocessadores e;
- ❑ no desenvolvimento de software.

Além disso, a natureza das aplicações também mudou. Esse crescimento de 50%, no desempenho do processador, alterou desde 2003. A partir desse ano, a melhoria de desempenho dos monoprocessadores caiu para cerca de 22% por ano, devido à dissipação máxima de potência do *chip*, à falta de maior paralelismo em nível de instrução, bem como devido à latência longa da memória. Em 2004, a empresa Intel juntou-se a outras empresas com objetivo de demonstrar que o caminho para um desempenho mais alto seria através de vários processadores por *chip* e não de uniprocessadores mais rápidos (PATTERSON; HENNESSY, 2014). Assim, houve um marco histórico nesta época onde o paralelismo em nível de instruções passou a concorrer com o paralelismo em nível de *thread* (*Thread-Level Parallelism - TLP*) e com o paralelismo em nível de dados (*Data-Level Parallelism - DLP*).

3.3 Classes de Paralelismo e Arquiteturas Paralelas

Existem basicamente dois tipos de paralelismo em aplicações, quais sejam:

1. Paralelismo em nível de dados (*Data-Level Parallelism*): grande volume de dados que podem ser operados ao mesmo tempo.
2. Paralelismo em nível de tarefas (*Task-Level Parallelism - TLP*): as tarefas podem operar de modo independente e, principalmente, de forma paralela.

Além disso, o hardware do computador pode explorar esses dois tipos de paralelismo de aplicações de quatro modos principais, tais como:

- ❑ paralelismo em nível de instruções: usa paralelismo em nível de dados com auxílio do compilador e utiliza *pipelining* em níveis médios, com execução especulativa;
- ❑ arquiteturas vetoriais e as unidades de processador gráfico (GPU): utilizam o paralelismo em nível de dados ao aplicar uma única instrução em um conjunto grande de dados, de forma paralela;

- ❑ paralelismo em nível de *threads*: trabalha com paralelismo em nível de dados ou paralelismo em nível de tarefas em um modo de hardware acoplado (interação entre *threads* paralelas);
- ❑ paralelismo em nível de requisição: explora o paralelismo entre tarefas muito desacopladas as quais são especificadas pelo programador ou pelo sistema operacional.

Esses quatro modos do hardware suportar o paralelismo têm sido motivo de estudo há 50 anos. Em 1966, Michael Flynn, estudou os esforços da computação paralela e propôs quatro classificações do sistema de computação baseado em fluxos de instruções (única ou múltipla) e em fluxos de dados (única ou múltipla) (PATTERSON; HENNESSY, 2014). As quatro categorias são:

1. Fluxo simples de instrução e fluxo simples de dados (*Single Instruction Stream, Single Data Stream* - SISD). Aqui se enquadra o uniprocessador, onde o programador trabalha com computador sequencial padrão. Também pode ser explorado o paralelismo em nível de instrução.
2. Fluxo simples de instrução e fluxo múltiplo de dados (*Single Instruction Stream, Multiple Data Streams* - SIMD). A mesma instrução é executada por múltiplos processadores usando diferentes fluxos de dados. Computadores nesta categoria exploram o paralelismo em nível de dados ao utilizar as mesmas operações a múltiplos itens em paralelo. Cada processador tem sua própria memória de dados, mas existe uma única memória de instruções e um único processador de controle que busca e envia instruções. Essa categoria engloba as arquiteturas vetoriais, extensões multimídia a conjuntos de instruções-padrão e as GPUs.
3. Múltiplas instruções e fluxo simples de dados (*Multiple Instruction Stream, Single Data Stream* - MISD). Nenhum microprocessador comercial foi construído até hoje com essa categoria.
4. Múltiplas instruções e fluxo múltiplo de dados (*Multiple Instruction Stream, Multiple Data Streams* - MIMD). Cada processador busca suas próprias instruções e opera seus próprios dados. O paralelismo ocorre em nível de tarefas, mas pode ocorrer também em nível de dados. Essa arquitetura engloba o paralelismo em nível de *thread* (múltiplas *thread* cooperam e operam em paralelo), e em nível de requisição (computadores em escala, onde muitas tarefas independentes podem ocorrer naturalmente em paralelo com pouca necessidade de comunicação e sincronismo).

3.4 Unidade de Processamento Gráfico

Desde 2003, a indústria de semicondutores tem estabelecido duas principais trajetórias de pesquisa e desenvolvimento dos microprocessadores. A primeira trajetória é a *multi-core* que trabalha com vários núcleos de processamento e tenta manter a velocidade de execução dos programas sequenciais. Inicialmente criou-se processadores com dois núcleos, sendo duplicado a cada geração. A segunda trajetória é a *many-core* que trabalha com centenas de núcleos de processamento para obter um desempenho computacional ainda mais alto, onde destaca-se a GPU (KIRK; HWU, 2010). Um exemplo de uma placa gráfica é ilustrado na Fig. 3.1.

Figura 3.1 – Placa de vídeo Geforce GTX da NVIDIA

Fonte: NVIDIA, 2014

3.4.1 Performance da Tecnologia GPU, *Multi-core* e CPU

A NVIDIA desenvolveu uma linguagem semelhante a C e um ambiente de programação que melhora a produtividade dos programadores de GPU. O nome desse sistema é conhecido como Arquitetura de Computação de Dispositivo Unificado (CUDA) que suporta as linguagens C/C++ para o processador do sistema (*host*) e para a GPU (*device*). A NVIDIA definiu que o unificador de todas essas formas de paralelismo seria o *Thread* CUDA, que estabelece que o compilador e o hardware possam reunir milhares de *threads* para utilizar os diversos estilos de paralelismo dentro de uma GPU. Neste sentido, a NVIDIA classificou o modelo de programação CUDA como instrução única e múltiplas *threads* (*Single Instruction, Multiple Thread - SIMT*) (PATTERSON; HENNESSY, 2014).

Nesse contexto, o modelo CUDA permite a plena utilização da capacidade dos núcleos de processadores da GPU para acelerar a execução das tarefas mais exigentes do sistema. Isso gera um grande aumento do desempenho quando comparado com as CPUs tradicionais (unidade de processamento central). A GPU permite que mais ações possam ser executadas com

menos tráfego de informações nos barramentos ao usar a área de *cache* comum e acesso direto à memória. Porém, a memória *cache* é utilizada em pequenas quantidades.

Além disso, a NVIDIA também desenvolveu o compilador CUDA C/C++, bibliotecas e software de *runtime* para permitir que os programadores rapidamente acessem o modelo de computação paralela de dados. Os programadores não precisam mais usar a API gráfica para acessar a capacidade de computação paralela da GPU.

De acordo com a Fig. 3.2, a melhoria do desempenho dos microprocessadores, de uma forma geral, avançou significativamente, ganhando um grande destaque as tecnologias GPUs, as quais estão evoluindo continuamente (KIRK; HWU, 2010).

Figura 3.2 – Performance entre CPU e GPU

Fonte: KIRK; HWU, 2010

Em 2009, pode-se perceber o grande avanço na velocidade de execução entre GPUs com múltiplos núcleos em relação aos processadores *multi-core*: 1 teraflops (1000 gigaflops) contra 100 gigaflops. Esta grande diferença de desempenho motivou muitos desenvolvedores a adaptar partes de seus softwares para uso em GPU (KIRK; HWU, 2010).

A diferença de desempenho entre GPUs e, por exemplo, processadores *multi-core* com 4 núcleos está na filosofia de *design* entre os dois tipos de processadores, como mostra a Fig. 3.3. Segundo Kirk e Hwu (2010) o projeto de uma CPU é otimizado para o desempenho do código sequencial e faz uso de lógica de controle para permitir que as instruções de um único segmento de execução possa ser executado em paralelo ou mesmo fora de sua ordem sequencial. Grandes memórias *cache* são fornecidas para reduzir a instrução e as latências de acesso aos dados de grandes aplicações complexas. Outra questão importante é a largura de banda da memória. As GPUs operam com cerca de 10 vezes a largura de banda das CPUs. No final de 2006, o projeto GeForce 8800 GTX, foi desenvolvido para mover dados em cerca de 85 gigabytes por segundo dentro e fora de sua memória principal (Memória de Acesso Aleatório Dinâmica - DRAM). As

Figura 3.3 – Diferença na filosofia do *design* entre CPU e GPU

Fonte: NVIDIA, 2014

GPUs possuem maior área de *chip* voltada para o processamento de instruções que as CPUs, que têm maior parte da área do chip dedicada a memória *cache* e controle. As CPUs têm grandes áreas de *chip* dedicadas às unidades lógicas e aritméticas (ULA) que são encarregados de executar as operações lógicas e aritméticas do sistema.

3.4.2 Arquitetura da GPU

Uma GPU preparada para CUDA é organizada em uma matriz de multiprocessadores de *streaming* (SMs) altamente encadeados (KIRK; HWU, 2010). As *threads* são reunidas em blocos, onde o hardware que executa um bloco inteiro de *threads* é conhecido como processador SIMD *multitreaded*. É necessário identificar as *threads* e os blocos através das variáveis: *threadIdx* e *blockIdx*, respectivamente. Além disso, CUDA fornece um identificador para o número de *threads* por bloco, conhecido como *blockDim*, de tal forma que o conjunto desses blocos formam um *grid*. Assim, o *grid* e bloco de *threads* são abstrações de programação implementadas no hardware da GPU para auxiliar os programadores a organizarem seus códigos CUDA (PATTERSON; HENNESSY, 2014). Uma coleção de processadores SIMD *multitreaded* executa um *grid* de blocos de *threads*, tal como ilustrado na Fig. 3.4. Cada processador SIMD pode executar um número variável de *threads* e os recursos locais são divididos entre elas, conforme especificado pelo programador. Essa flexibilidade permite otimizar ainda mais as aplicações.

As GPUs atuais podem disponibilizar acima de 6 gigabytes de DRAM, chamada de memória global. Entretanto, dependendo do volume de dados pode haver limitação da largura de banda da memória, sendo necessário utilizar as memórias especializadas no chip da GPU para reduzir o número de acessos à DRAM. É necessário que o programador analise a melhor forma de projetar seu algoritmo.

Figura 3.4 – Blocos de *threads*

Fonte: SANDERS; KANDROT, 2011

Além disso, a maioria das aplicações possui partes que podem ser executadas muito melhor pela CPU. Assim, é preciso utilizar a computação heterogênea do sistema, mesclando CPU/GPU, de tal modo que um complete a execução do outro. Isso pode ser obtido através da programação CUDA. Entretanto, é necessário estudar o código de uma aplicação para saber qual parte é sequencial e qual parte é paralela. Nos estudos de caso deste trabalho, essa programação heterogênea foi utilizada e permitiu grande redução no tempo de processamento.

3.4.3 Linguagens e Modelos de Programação Paralela

Muitas linguagens e modelos de programação paralela têm sido propostos nas últimas décadas. Os mais utilizados são o *Message Passing Interface* (MPI) para computação de *cluster* escalável e o OpenMP para sistemas multiprocessados, com memória compartilhada. O MPI é um modelo de programação no qual os nós de computadores, em um *cluster*, não compartilham a memória, sendo todo o compartilhamento e interação de dados feitos através de passagem explícita de mensagens, através de uma rede. Esse modelo funciona muito bem em várias aplica-

ções, entretanto o esforço exigido para transpor uma aplicação para MPI pode ser extremamente alto, devido à falta de memória compartilhada entre os nós (PATTERSON; HENNESSY, 2014). Se houver grande comunicação entre os nós, isso pode se tornar também um limitador neste modelo de programação. Além disso, soma-se a questão do custo computacional, pois muitos nós devem estar disponíveis para se montar um *cluster*. O modelo CUDA, por outro lado, oferece memória compartilhada para a execução paralela na GPU e resolve essas limitações, além de ser mais barato que a montagem de um *cluster*. Ao utilizar computação heterogênea em CUDA, os programadores precisam gerenciar a transferência de dados entre CPU e GPU e vice-versa, de maneira semelhante à passagem de mensagem "unilateral".

De acordo com Patterson e Hennessy (2014), OpenMP tem suporte para memória compartilhada, de modo a oferecer a mesma vantagem do modelo CUDA, em relação aos esforços de programação. Entretanto, não foi capaz de expandir os nós de computação devido aos esforços adicionais de gerenciamento de *thread* e requisitos de hardware, para coerência de memória *cache*. Assim, CUDA atinge uma escalabilidade muito maior e possui um gerenciamento de *threads* simples, com pouco esforço adicional, e nenhum requisito de hardware para coerência de *cache*.

OpenMP permite maior automação no gerenciamento da execução paralela, entretanto, muitas técnicas de otimização de desempenho são comuns entre os modelos MPI, OpenMP e CUDA. Assim, isso torna o modelo CUDA fácil de se assimilar.

Além dos citados, existe ainda o modelo de programação semelhante ao CUDA, conhecido como OpenCL, que define extensões da linguagem e APIs de *runtime* para permitir que os programadores gerenciem o paralelismo e o envio de dados, de forma paralela. Entretanto, o nível das construções de programação ainda é mais baixo que CUDA, sendo muito mais difícil de ser utilizado. Além disso, de acordo com Patterson e Hennessy (2014), a velocidade alcançada em uma aplicação OpenCL é muito menor que aquelas obtidas em CUDA, considerando as plataformas que admitem ambas. Esse foi um dos motivos no direcionamento deste trabalho, além da NVIDIA CUDA ser a classe de dispositivos GPUs mais comum em uso atualmente. Duas desvantagens que merecem destaque são: (i) ao se usar CUDA, tem-se um aplicativo dependente do hardware (NVIDIA) o que prejudica a sua portabilidade e (ii) a largura de banda da memória pode ser um limitador.

Resumindo, as principais características dos modelos de programação, de acordo com Steinberg et al. (2010), são:

- ❑ OpenCL: é um padrão aberto de computação heterogênea, desenvolvida pelo grupo Khronos OpenCL. Implementações estão disponíveis pelos fornecedores de hardware e software, incluindo a AMD, a Apple, NVIDIA e IBM. É um modelo de programação de baixo nível e utiliza filas de comando assíncrono, além de apresentar uma estrutura de memória complexa que requer gerenciamento. Suporta as linguagens de programação C e C++ e fornece um modelo de programação muito poderoso para acessar todo o potencial das arquiteturas de múltiplos núcleos;

- ❑ CUDA: permite a aceleração do desempenho computacional, aproveitando o poder da GPU. Também permite acessar as funções das placas usando a linguagem C e C++, além de conter recursos para a linguagem Java. O nível da construção da programação é mais alto que OpenCL, o que o torna uma ferramenta um pouco mais simples para os desenvolvedores;
- ❑ arquitetura híbrida: pode combinar duas ou mais bibliotecas em um único programa, com intuito de aproveitar mais recursos computacionais. As formas mais comuns de modelos híbridos combinam MPI com OpenMP e MPI com CUDA. O primeiro é adequado para uso em *clusters*, onde os nós são máquinas *multi-core*. O segundo é adequado para uso também em *clusters*, entretanto os nós são máquinas *many-core* (GPU). Assim, MPI é utilizado para trocar dados entre nós, OpenMP para aumentar o poder computacional dos núcleos em nós individuais e CUDA para aumentar a performance de cada placa.

Neste trabalho foram utilizados os seguintes modelos de programação na solução do sistema de equações lineares:

1. OpenMP no estudo de caso que implementa o método de Elementos Finitos no regime estático.
2. CUDA nos estudos de caso que implementam tanto o método de Elementos Finitos no regime estático quanto no regime harmônico.

3.4.4 Estrutura de um Programa CUDA

Um programa CUDA consiste em uma ou mais fases que são executadas ou no *host* (CPU) ou no *device* (GPU). As partes que exigem pouco ou nenhum paralelismo são implementadas no código *host*, enquanto que as partes que envolvem grande paralelismo são realizadas no código do *device*, tal como ilustrado na Fig. 3.5.

Um programa CUDA é um código fonte unificado, tanto para CPU quanto para GPU. Para utilizar o modelo CUDA, todo um ambiente deve ser criado para executar um código em paralelo (SANDERS; KANDROT, 2011). Neste sentido, os pré-requisitos para desenvolver o código CUDA C são:

- ❑ um processador gráfico compatível com CUDA;
- ❑ um *driver* de dispositivo NVIDIA;
- ❑ um conjunto de ferramentas de desenvolvimento CUDA;
- ❑ um compilador C padrão.

Figura 3.5 – Execução de um algoritmo CUDA

Fonte: NVIDIA, 2014

Primeiramente, o computador deve ter um processador gráfico que utilize CUDA. A NVIDIA fornece software que permite que o programa consiga se comunicar com o hardware compatível com CUDA, sendo necessário instalar a GPU NVIDIA corretamente. É importante manter os *drivers* mais recentes compatíveis com a placa gráfica e com o sistema operacional.

Além disso, é imprescindível a instalação de um software adicional, conhecido como *CUDA Toolkit* para desenvolver código GPU NVIDIA CUDA com linguagem C, por exemplo. Todos estão disponíveis em (NVIDIA, 2014). Ao instalar esses softwares é necessário selecionar a plataforma 32-bit ou 64-bit cujas versões estão disponíveis em Windows XP, Windows Vista, Windows 7, Linux e Mac OS.

Para compilar o código é necessário um compilador C da NVIDIA (*nvcc*) que separa o código do *host* e o código do *device*. O código do *host* é escrito usando ANSI C simples, que é compilado com os compiladores C padrão do *host*, o qual roda como um processo comum da CPU. O código do *device* é escrito usando ANSI C estendido, com palavras-chave para rotular as funções com dados paralelos, chamadas *Kernels*, e suas estruturas de dados associadas. Esse código normalmente é compilado ainda pelo *nvcc* e executado em um *device* (SANDERS; KANDROT, 2011), (KIRK; HWU, 2010). Neste caso, para plataforma Windows o ambiente de desenvolvimento e compilação mais recomendado é o Microsoft Visual Studio (SANDERS; KANDROT, 2011).

As funções *Kernels* normalmente geram um grande número de *threads* para explorar o paralelismo de dados. Assim, todas elas são geradas por um *Kernel*, as quais são conhecidas coletivamente como *grid*, como já exposto anteriormente e ilustrado na Fig. 3.6. Essa figura tem dois *kernels*, sendo que cada um é formado por um *grid* de blocos de *threads*, ou seja, cada *grid* consiste em um ou mais blocos de *threads* e cada bloco é organizado como uma matriz tridimensional.

Figura 3.6 – Organização das *threads*

Fonte: NVIDIA, 2014

3.4.5 Memória do *Device* e Transferência de Dados

Em CUDA, *host* e *device* possuem espaços de memória separados. As placas gráficas vêm com sua própria memória DRAM. Para executar um *Kernel*, o programador precisa alocar memória no *device* e transferir dados da memória do *host* para a memória da GPU, ao utilizar o processamento paralelo. De modo semelhante, após execução no *device*, o programador precisa transferir os dados da memória da GPU para a CPU e liberar aquela memória do *device* que não for mais utilizada. A Fig. 3.7 ilustra a visão geral do modelo de memória da GPU NVIDIA, que inclui seus diversos componentes, além dos registradores, as memórias compartilhada, local, global e constante.

Figura 3.7 – Modelo de memória do *device* CUDA

Fonte: NVIDIA, 2014

Ao analisar essa figura, pode-se concluir que a memória global e a memória constante são utilizadas para que o código *host* transfira dados para o *device*, conforme ilustrado pelas setas bidirecionais entre elas e o *host*. A memória constante permite o acesso apenas de leitura pelo código do *device*, com baixa latência e alta largura de banda. A função da API para alocar e desalocar a memória global do *device* é chamada de `cudaMalloc()`. Essa função pode ser chamada pelo código do *host* para alocar parte da memória global. Quando um programa já tiver alocado a memória global no *device* para os objetos de dados, ele pode solicitar que os dados sejam transferidos do *host* para o *device* através da função da API CUDA, a qual é conhecido como `cudaMemcpy()` (KIRK; HWU, 2010).

Cada *grid* consiste em um ou mais blocos de *threads* e um bloco *threads* pode cooperar entre si, sincronizando sua execução (para acesso à memória compartilhada) e compartilhando dados de modo eficiente. Assim, duas *threads* de dois blocos diferentes não podem cooperar entre si.

Os registradores e a memória compartilhada são memórias no chip. As variáveis que residem nesses tipos de memória podem ser acessadas em velocidade muito alta, de uma maneira altamente paralela. Os registradores alocam as *threads* individuais, sendo que cada uma só pode acessar seus próprios registradores. Uma função do *Kernel* normalmente usa registradores para

manter variáveis acessadas frequentemente, que são privadas a cada *thread*. A memória compartilhada é alocada em blocos de *threads* e, todas elas em um bloco, podem acessar variáveis nos locais de memória compartilhada, alocados nesse bloco. Um mecanismo eficiente para as *threads* cooperarem compartilhando seus dados de entrada e seus resultados intermediários é através da memória compartilhada. Os programadores normalmente usam a memória compartilhada para manter a parte dos dados da memória global que são muito usados em uma fase de execução do *Kernel*. Os acessos a uma variável global são lentos, entretanto as variáveis globais são visíveis a todas as *threads* de todos os *Kernels* (KIRK; HWU, 2010).

Assim, um código do *device* tem as seguintes funções:

- ☐ ler e escrever registradores por *thread*;
- ☐ ler e escrever memória local por *thread*;
- ☐ ler e escrever memória compartilhada por *thread*;
- ☐ ler e escrever memória global por *thread*;
- ☐ ler apenas memória constante por *grid*.

O código *host* pode transferir dados de/para memória global e constante, por *grid*.

3.4.6 Tecnologia NVIDIA *GPUDirect*

A computação paralela maciça nas GPUs nem sempre soluciona problemas em grande escala. Uma desvantagem que merece destaque é o fato das GPUs terem baixa capacidade de memória. Isso se torna um gargalo quando se trabalha com grandes escalas de problemas, especialmente com um volume grande de dados. De fato, muitos computadores pessoais atuais contêm geralmente quatro a oito vezes mais memória na placa-mãe (CPU) que nas GPUs. Entretanto, a NVIDIA desenvolveu a tecnologia *GPUDirect* que auxilia a transferência de dados entre GPUs, que contêm duas placas integradas em um mesmo computador. Essa configuração é conhecida como *Multi-GPU* que tornou-se uma solução viável para muitas aplicações em grande escala.

A tecnologia NVIDIA *GPUDirect* pode fornecer uma boa solução na comunicação, além de permitir a comunicação entre GPUs no mesmo servidor. A *GPUDirect* permite também o uso em alta velocidade do acesso direto à memória (DMA) para copiar dados entre as memórias de duas GPUs, no mesmo barramento do sistema (padrão PCIe). Os dados podem se comunicar diretamente e pode ocorrer a redução do tempo de comunicação ao utilizar a função *peer-to-peer* para (NVIDIA, 2014):

- ☐ permitir o acesso a memória entre duas GPUs;
- ☐ permitir a cópia de dados entre duas GPUs.

Essa tecnologia suporta o modelo de programação CUDA, entretanto precisa-se ter instalado no computador o pacote *CUDA Toolkit* e um sistema com duas ou mais GPUS que têm a arquitetura Fermi ou Kepler, no mesmo barramento PCIe. A Fig. 3.8 ilustra essa tecnologia, sendo que a primeira se refere ao acesso direto à memória e a segunda à transferência entre as GPUs.

Figura 3.8 – Tecnologia *GPUDirect* que permite comunicação entre GPUs

Fonte: NVIDIA, 2014

3.5 Tecnologias e Relação Custo-Benefício

Existem atualmente no mercado diversas placas de vídeo da NVIDIA, cada uma com determinada característica e objetivo. Por exemplo, em termos de desempenho computacional (dupla precisão) a GPU GTX 680 (arquitetura *Kepler*) apresentou 197,80 GFlops contra 616,30 GFlops para a GPU Tesla M 2090 (arquitetura *Fermi*) (NVIDIA, 2014). Isso pode ser visualizado através da Fig. 3.9.

As arquiteturas GeForce, Quadro e Tesla estão habilitadas para trabalhar com a plataforma de computação paralela CUDA. Os produtos da GeForce e Quadro são desenvolvidos especialmente para o consumidor comum e visualização profissional, enquanto que os produtos Tesla foram projetados com recursos exclusivos para maximizar performance dos trabalhos dos profissionais que utilizam a supercomputação. A Fig. 3.10 ilustra um comparativo em termos de largura de banda da CPU e GPU, sendo essa última para as arquiteturas Tesla e Geforce.

A relação custo-benefício da arquitetura Tesla deixa a desejar devido ao seu alto custo, comparada aos produtos GeForce. Por exemplo, o preço da GPU GTX 680 é em torno de 3,5 vezes mais barato que a GPU Tesla M 2090. Isso foi um dos fatores decisivos na escolha da

Figura 3.9 – Desempenho das arquiteturas Geforce GTX 680 e Tesla M 2090

Fonte: NVIDIA, 2014

arquitetura. Desde o início do trabalho focou-se em utilizar os recursos da computação paralela em computador pessoal de custo relativamente baixo.

Figura 3.10 – Largura de banda de memória para a CPU e GPU (Tesla e Geforce)

Fonte: NVIDIA, 2014

3.6 Síntese do Capítulo

Foram abordadas neste capítulo as características principais e intrínsecas do processamento paralelo em CPU e GPU. Alguns modelos de programação foram apresentados, além de sua arquitetura, memória e estrutura de dados. No próximo capítulo serão apresentadas as formulações do Método de Elementos Finitos nos regimes estático e harmônico.

Método de Elementos Finitos no Eletromagnetismo

4.1 Introdução

Neste capítulo serão apresentadas sucintamente as formulações do Método de Elementos Finitos nos regimes estático e harmônico em Eletromagnetismo. No primeiro caso, regido pela equação de Laplace, a formulação nodal será apresentada e o potencial elétrico será determinado. Já no segundo caso, regido pela equação de onda completa, não homogênea, em meios com perdas e no domínio da frequência, a formulação nodal-aresta é introduzida e os potenciais escalar elétrico e vetor magnético serão determinados a partir de um mesmo sistema de equações algébricas. As principais abordagens serão introduzidas neste capítulo e informações mais detalhadas e específicas podem ser encontradas em (SILVA, 2006). Características específicas do problema de aterramento serão abordados no Capítulo 6.

4.2 Formulação para o Regime Estático

As relações dos campos elétricos e magnéticos, cargas, correntes e ondas eletromagnéticas são governadas pelas equações de Maxwell (JIN, 2002). O problema de distribuição de correntes estacionárias, em meios condutores, a Eletrocinética, é modelado pelas seguintes equações:

$$\nabla \times \mathbf{E} = 0, \quad (4.1)$$

$$\nabla \cdot \mathbf{J} = 0, \quad (4.2)$$

$$\mathbf{J} = [\sigma] \mathbf{E}, \quad (4.3)$$

sendo $[\sigma]$ um tensor de condutividade elétrica, $\mathbf{J}$ o campo vetorial densidade de corrente e $\mathbf{E}$ o campo elétrico. Da Eq. (4.1) deriva-se a expressão que introduz o potencial escalar elétrico, V , tal como:

$$\mathbf{E} = -\nabla V. \quad (4.4)$$

Ao rearranjar as equações descritas acima e substituir a Eq. (4.4) na (4.2) e na (4.3), uma equação diferencial a derivadas parciais pode ser obtida, denominada equação de Laplace, como segue:

$$\nabla \cdot (-[\sigma] \nabla V) = 0 \quad \text{em } \Omega. \quad (4.5)$$

sendo Ω o domínio no qual a equação se aplica.

A essa equação deve-se acrescentar as condições de contorno de Dirichet e de Neumann (JIN, 2002) dadas a seguir:

$$V = V_0 \quad \text{em } \Gamma_D, \quad (4.6)$$

$$J \cdot \hat{n} = 0, \text{ ou } \nabla V \cdot \hat{n} = 0 \quad \text{em } \Gamma_N. \quad (4.7)$$

Assim, pode-se encontrar o potencial elétrico em qualquer ponto do domínio Ω utilizando os valores pré-estabelecidos na fronteira, onde $\Gamma_D \cup \Gamma_N = \Gamma$ corresponde à fronteira que delimita Ω . A partir desse desenvolvimento pode-se aplicar o Método de Elementos Finitos para obter a solução do problema.

De uma forma geral, alguns passos devem ser estabelecidos e seguidos para a aplicação do MEF, quais sejam:

- ☐ modelagem geométrica do problema e propriedades físicas (pré-processamento);
- ☐ aplicação do método residual de Galerkin ou Método Variacional nas Eqs. (4.5), (4.6) e (4.7) (JIN, 2002);
- ☐ discretização do domínio Ω , ou geração da malha de elementos finitos;
- ☐ definição da função V de aproximação para um elemento típico;
- ☐ efetivação das etapas de integração no elemento e montagem do sistema de equações algébricas;
- ☐ resolução do sistema de equações algébricas;
- ☐ exploração dos resultados e cálculo de grandezas de interesse derivadas (pós-processamento).

No segundo item introduzido acima, vê-se que o MEF pode ser aplicado via dois métodos distintos, Residual de Galerkin ou Variacional. Ambos conduzirão no entanto ao mesmo sistema de equações, que resultará simétrico (JIN, 2002).

Os elementos da discretização podem ter uma, duas ou três dimensões os quais podem assumir as seguintes formas: linhas, triângulos, quadriláteros, tetraedros, hexaedros, prismas e pirâmides. Para o caso específico da Eletrocinética neste trabalho, o elemento utilizado para discretizar o domínio foi o tetraedro.

Após discretizar o domínio, a função incógnita no elemento (V) pode ser aproximada pela função $\tilde{V}$ tal como:

$$V^e(x, y, z) \approx \tilde{V}^e(x, y, z) = \sum_{j=1}^{nn} V_j^e N_j^e(x, y, z), \quad (4.8)$$

sendo nn o número de nós do elemento, $N_j^e(x, y, z)$ a função de forma do elemento e associada ao nó j e V_j^e a função no nó j . Ao aplicar o Método Residual de Galerkin chega-se à seguinte equação residual (SILVA, 2006):

$$R_i^e = \sum_{j=1}^{nn} V_j^e \int_{\Omega^e} \nabla N_i^e [\sigma] \nabla N_j^e d\Omega - \int \int_{\Gamma^e} N_i^e [\sigma] \nabla V^e \cdot \hat{n}^e d\Gamma, \quad (4.9)$$

que na forma matricial se escreve:

$$\{R^e\} = [K^e] \{V^e\} - \{g^e\}. \quad (4.10)$$

Igualando a zero as contribuições de todos os resíduos dos elementos contidos no domínio discretizado, com o intuito de minimizá-lo globalmente, chega-se ao seguinte sistema de equações algébricas:

$$[K] \{V\} = \{g\}, \quad (4.11)$$

em que os termos genéricos são dados a seguir:

$$K_j^i = \int_{\Omega^e} \nabla N_i^e [\sigma] \nabla N_j^e d\Omega \quad (4.12)$$

e

$$g_i = \int \int_{\Gamma^e} N_i^e [\sigma] \nabla V^e \cdot \hat{n}^e d\Gamma. \quad (4.13)$$

em que V_i é o valor nodal do potencial escalar elétrico no nó i do elemento.

A partir da solução desse sistema, o potencial elétrico pode ser obtido em todos os pontos do domínio da malha de elementos finitos. Esse sistema de equações tem as seguintes particularidades:

- ❑ é simétrico e esparso. A esparsidade é devida à discretização da geometria e à montagem do sistema global;
- ❑ a resolução das equações algébricas depende da corrente inserida no sistema de aterramento elétrico e das condições de contorno;
- ❑ sua ordem é elevada e igual ao número de nós da malha de elementos finitos.

4.3 Formulação para o Regime Harmônico

Os fenômenos eletromagnéticos com variação temporal senoidal são governados pelas Equações de Maxwell na sua forma completa escritas com notação fasorial (JIN, 2002), quais sejam:

$$\nabla \times \mathbf{H} = \mathbf{J}, \quad (4.14)$$

$$\nabla \times \mathbf{E} = -j\omega \mathbf{B}, \quad (4.15)$$

$$\nabla \cdot \mathbf{B} = 0, \quad (4.16)$$

$$\nabla \cdot \mathbf{J} = 0, \quad (4.17)$$

$$\nabla \cdot \mathbf{D} = \rho, \quad (4.18)$$

em que ω é a frequência angular da variação senoidal, j é a raiz de (-1) , $\mathbf{H}$ corresponde ao campo magnético, $\mathbf{E}$ o campo elétrico, $\mathbf{B}$ e $\mathbf{D}$ as densidades de fluxo magnético e elétrico, respectivamente e $\mathbf{J}$ a densidade de corrente elétrica total, englobando as correntes de condução e de deslocamento. Neste contexto, as grandezas vetoriais são também fasores, mas não será utilizada notação distinta aqui, por simplicidade.

Aplicando-se as relações constitutivas e rearranjando-se essas equações, chega-se a uma equação diferencial a derivadas parciais que correspondente à equação de onda vetorial não homogênea na forma fasorial, dada por:

$$\nabla \times [\mathbf{v}] \nabla \times \mathbf{E} + j\omega [\dot{\sigma}] \mathbf{E} = j\omega \mathbf{J}_0. \quad (4.19)$$

sendo $[\mathbf{v}]$ um tensor de relutividade magnética, $\mathbf{J}_0$ uma densidade de corrente impressa devido à fonte de correntes externas, e o $[\dot{\sigma}]$ um tensor, tal que:

$$[\dot{\sigma}] = [\sigma] + j\omega \epsilon_0 [\epsilon_r]. \quad (4.20)$$

Nessa expressão, $[\sigma]$ é um tensor de condutividade elétrica dos meios envolvidos, ϵ_0 é a permissividade do vácuo e $[\epsilon_r]$, o tensor de permissividade elétrica relativa desses meios. A essa equação deve-se acrescentar as condições de contorno e de interface para obter a solução do problema.

A Eq. (4.19) pode ser escrita em termos do potencial elétrico e magnético, fazendo uso das seguintes relações:

$$\mathbf{B} = \nabla \times \mathbf{A}, \quad (4.21)$$

$$\mathbf{E} = -j\omega\mathbf{A} - \nabla V, \quad (4.22)$$

em que $\mathbf{A}$ corresponde ao fasor potencial vetor magnético e V o fasor do potencial escalar elétrico. Efetuando-se as substituições necessárias chega-se a:

$$\nabla \times [\mathbf{v}] \nabla \times \mathbf{A} + [\dot{\sigma}] (j\omega\mathbf{A} + \nabla V) = \mathbf{J}_0. \quad (4.23)$$

Essa equação não é suficiente para se determinar os potenciais $\mathbf{A}$ e V , mesmo com as condições de contorno e interface. A equação faltante é dada pela Equação da Continuidade, (4.17), escrita em termos de $\mathbf{A}$ e V , ou seja:

$$\nabla \cdot (-j\omega[\dot{\sigma}]\mathbf{A} - [\dot{\sigma}]\nabla V) = 0. \quad (4.24)$$

A essas equações aplica-se então o método residual de Galerkin, e após alguma manipulação matemática, chega-se a:

$$\int_{\Omega} \nabla \times \mathbf{\varpi}_i \cdot [\mathbf{v}] \nabla \times \mathbf{A} \, d\Omega + \int_{\Omega} \mathbf{\varpi}_i \cdot j\omega[\dot{\sigma}](\mathbf{A} + \nabla \mathbf{v}) \, d\Omega = \int \int_{\Gamma_H} \mathbf{\varpi}_i \cdot (\bar{\mathbf{H}}_t \times \hat{n}) \, d\Gamma, \quad (4.25)$$

$$\int_{\Omega} \nabla N_i \cdot j\omega[\dot{\sigma}](\mathbf{A} + \nabla \mathbf{v}) \, d\Omega = I + \int \int_{\Gamma_H - \Gamma_{PD}} \nabla N_i \cdot (\bar{\mathbf{H}}_t \times \hat{n}) \, d\Gamma. \quad (4.26)$$

em que $\mathbf{\varpi}_i$ representa a função peso vetorial do método de resíduos ponderados que corresponde à função de forma vetorial do MEF de aresta (SILVA, 2006). Nessas expressões fez-se a mudança de variável, $V = j\omega\mathbf{v}$, a fim de tornar o sistema de equações simétricos.

Assim, após discretizar o domínio, as funções incógnitas $\mathbf{v}$ e $\mathbf{A}$ podem ser aproximadas como se segue:

$$\mathbf{A}^e(x, y, z) \cong \sum_{i=1}^{na} a_i \mathbf{\varpi}_i^e(x, y, z), \quad (4.27)$$

$$\mathbf{v}^e(x, y, z) \cong \sum_{i=1}^{nm} v_i N_i^e(x, y, z). \quad (4.28)$$

em que na é o número total de arestas de um elemento e a_i é a integral de linha de $\mathbf{A}$ sobre a aresta i do elemento.

Substituindo as Eqs. (4.27) e (4.28) nas Eqs. (4.25) e (4.26) e aplicando-se as condições de contorno, chega-se ao seguinte sistema matricial:

$$\begin{pmatrix} M_A & M_{Av} \\ M_{Av}^T & M_v \end{pmatrix} \begin{bmatrix} a \\ v \end{bmatrix} = \begin{bmatrix} 0 \\ I \end{bmatrix} \quad (4.29)$$

Os termos genéricos de cada submatriz são dados por:

$$M_{A_{mn}} = \int_{\Omega} [\nabla \times \mathfrak{w}_m]^T \cdot [v] [\nabla \times \mathfrak{w}_n] + [\mathfrak{w}_m]^T \cdot j\omega [\dot{\sigma}] [\mathfrak{w}_n] d\Omega, \quad (4.30)$$

$$M_{Av_{mn}} = j\omega \int_{\Omega} [\mathfrak{w}_m]^t [\dot{\sigma}] [\nabla N_n] d\Omega, \quad (4.31)$$

$$M_{v_{mn}} = j\omega \int_{\Omega} [\nabla N_m]^t [\dot{\sigma}] [\nabla N_n] d\Omega, \quad (4.32)$$

em que a_n é a integral de linha de A sobre a aresta n , v_n é o valor da variável v no nó n e $I = I$ no ponto de injeção de corrente e nulo nos restantes dos nós.

Esse sistema de equações apresenta as seguintes características:

- ❑ é simétrico e esparso. A esparsidade é devida à discretização da geometria e à assemblagem do sistema global;
- ❑ a resolução das equações algébricas depende estritamente da imposição da corrente no sistema de aterramento elétrico, a qual é imposta pelas condições de contorno;
- ❑ a largura de banda é influenciada pelo algoritmo de geração da malha. A condensação dos elementos na matriz global depende da numeração dos nós e das arestas. A largura de banda será menor quando os nós e as arestas contiverem uma numeração mais próxima possível. A largura da banda influenciará o modo de armazenamento da matriz na memória;
- ❑ a matriz M_A é oriunda do método de Elementos Finitos de aresta e é utilizada para calcular a distribuição do potencial vetor magnético no domínio, enquanto que a matriz M_v , da integral no tempo do potencial elétrico $v = V/j\omega$, é oriunda do método de Elementos Finitos nodal e se refere ao cálculo dos potenciais elétricos;
- ❑ as matrizes M_{Av} e sua transposta estabelecem a conexão daqueles elementos adjacentes que contribuem tanto para o cálculo do potencial elétrico quanto do potencial vetor magnético;
- ❑ pode ser muito mal-condicionado dependendo da combinação de propriedades físicas dos meios, da frequência de variação temporal, da qualidade dos elementos da malha de elementos finitos, do contraste entre diferentes propriedades físicas e dimensões dentro do mesmo domínio, etc.

4.4 Síntese do Capítulo

Neste capítulo foram abordados os equacionamentos baseados na aplicação do MEF que originam os sistemas de equações algébricas, real e complexo, que deverão ser resolvidos por algum método de solução iterativo. Os métodos de interesse para este trabalho serão apresentados no próximo capítulo.

Métodos Iterativos de Solução de Sistemas Lineares

5.1 Introdução

*E*ste capítulo se destina a introduzir os principais métodos de solução do sistema linear oriundo do Método de Elementos Finitos. Além dos métodos iterativos, algumas características particulares da matriz obtida dos problemas de aterramento elétrico no regime estático e harmônico serão apresentadas. Todos os métodos foram implementados de forma paralela utilizando CUDA. A sua forma sequencial está disponível na biblioteca LMAGLIB, do laboratório LMAG-EPUSP, exceto o método do Gradiente Conjugado para matrizes reais. Para acelerar a convergência dos métodos iterativos e para resolver o sistema de equações algébricas alguns pré-condicionadores foram utilizados, os quais serão apresentados ao longo deste capítulo.

5.2 Métodos de Solução do Sistema de Equações Algébricas

De uma forma geral o conjunto de equações lineares oriundos do método de Elementos Finitos pode ser representado matricialmente por (JIN, 2002):

$$[A] [x] = [b], \quad (5.1)$$

de tal forma que $[x]$ representa o vetor incógnita do sistema. Esse sistema linear pode ter milhares de incógnitas as quais dependem do problema físico. Para um sistema com n variáveis, as equações algébricas podem ser representadas da seguinte forma:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \quad (5.2)$$

onde, $[A]$ é a matriz de coeficientes e $[b]$ é a matrix coluna.

Os métodos iterativos utilizam aproximações sucessivas para encontrar as incógnitas do sistema de equações lineares. O número de iterações necessárias para obter a solução depende do processo inerente de cada método e do problema físico. O sistema demora a obter a convergência quando este número de iterações é muito grande e, como consequência, o custo computacional se torna muito elevado. A convergência não é garantida para todos os métodos e alguns convergem mais rapidamente do que outros.

A escolha do melhor método requer um pouco de cuidado e estudo (SAAD, 2003). Se a matriz A é simétrica e definida positiva a escolha fica mais fácil porque o método do Gradiente Conjugado é bem adequado. Para outros tipos de matrizes a situação é um pouco mais delicada. O método do Resíduo Mínimo Generalizado (GMRES), proposto por Saad e Schultz (1986), é o mais robusto, mas não se torna tão atrativo em termos de armazenamento por iteração. O método Gradiente Biconjugado (BiCG), sugerido por Fletcher (1976), é uma alternativa satisfatória que necessita relativamente menos armazenamento de memória que o GMRES, entretanto, contém problemas de convergência, usualmente denominado de *breakdown*. Para contorná-lo foram, posteriormente, desenvolvidas outras técnicas como *look-ahead*. O desenvolvimento de métodos híbridos começou em 1989, proposto por Sonneveld (1989) e conhecido como Gradiente Conjugado ao Quadrado (CGS) e seguido pelo Gradiente Biconjugado Estabilizado (BiCGStab), proposto por Van der Vorst (1992), dentre outros.

Alguns desses métodos encontram-se disponíveis na forma sequencial na biblioteca LMA-GLIB, do laboratório LMAG. Para os estudos de caso deste trabalho que contam com certas particularidades (dimensão do problema e propriedades físicas) e matrizes muito mal-condicionadas, esses métodos se aplicam bem. O método BiCG requer algum armazenamento extra de memória comparado ao CG, mas responde bem às características intrínsecas das matrizes.

Os principais métodos analisados foram (JIN, 2002), (SAAD, 2003):

- ❑ Gradiente Conjugado (CG);
- ❑ Conjugado Residual (CR);
- ❑ Gradiente Biconjugado (BiCG);
- ❑ Gradiente Biconjugado Estabilizado (BiCGStab);
- ❑ Gradiente Conjugado para equações normais (CGNE e CGNR);

❑ Gradiente Conjugado ao Quadrado (CGS).

Os métodos Gradiente Conjugado para equações normais CGNE e CGNR foram implementados para um caso específico, mas não é foco de discussão nesta pesquisa. Em função disso, eles não serão detalhados neste capítulo.

Além desses métodos iterativos é necessário pré-condicionar as matrizes de tal forma a acelerar a convergência (SAAD, 2003). Foram analisados os seguintes métodos:

- ❑ Decomposição Incompleta LU (ILU);
- ❑ Decomposição Incompleta de Cholesky (IC);
- ❑ *Multigrid* Algébrico (AMG);
- ❑ Diagonal (DIA);
- ❑ Inversa Aproximada (AINV).

Dependendo da matriz, esses algoritmos podem falhar (pivôs nulos ou negativos) e algum processo adicional de compensação precisa ser aplicado de modo a se conseguir a dominância da diagonal. O mais usado e o qual será adotado é aquele detalhado em Fujiwara et al. (1993). É necessário adicionar à diagonal um determinado valor para que a diagonal se transforme em dominante. Esse método será melhor detalhado adiante.

5.2.1 Método do Gradiente Conjugado

No início da década de 50, o método do Gradiente Conjugado foi proposto por Hestenes e Stiefel (1952). Esse método é muito utilizado na resolução de várias classes de problemas e tem a grande vantagem de ser rápido na sua convergência e na solução de sistemas de equações de ordem elevada.

O método pertence à família do subespaço de Krylov que são muito poderosos na resolução de sistemas lineares esparsos, especificamente com matriz simétrica definida positiva. O método funciona gerando sequências de vetores de soluções aproximadas, de vetores de resíduo e de vetores de direções de procura que são utilizados para atualizar as soluções aproximadas e os resíduos (SAAD, 2003). Mesmo que o método produza uma sequência de iterações grande, poucos vetores precisam ser armazenados.

O CG atinge a solução depois de n iterações. De acordo com Saad (2003) e Jin (2002) um critério de parada é estabelecido e a solução é encontrada quando o resíduo for suficientemente pequeno e quando a convergência é obtida. O critério analisado é conhecido como norma do resíduo. O algoritmo para quando o seguinte critério é obtido:

$$\frac{\|r_i\|_2}{\|b\|_2} \leq \epsilon \quad (5.3)$$

O pseudo-código, como encontrado em Saad (2003), pode ser visualizado na Eq. (5.4).

Gradiente Conjugado

1. *Calcular* $r_0 = b - Ax_0$,
 $p_0 = r_0$.
2. *Para* $j = 0, 1, \dots$, *até convergir* :
3. $\alpha_j = \frac{(r_j, r_j)}{(Ap_j, p_j)}$
4. $x_{j+1} = x_j + \alpha_j p_j$
5. $r_{j+1} = r_j - \alpha_j Ap_j$
6. $\beta_j = \frac{(r_{j+1}, r_{j+1})}{(r_j, r_j)}$
7. $p_{j+1} = r_{j+1} + \beta_j p_j$
8. *Fim* (5.4)

As variáveis α_j e β_j são escalares e dependem das características intrínsecas do problema e, posteriormente, da matriz A . O vetor r_j é conhecido como resíduo. Já o vetor com a solução aproximada, x_j , é atualizado em cada iteração por um múltiplo α_j daquele vetor que guarda a direção de procura p_i .

Em cada iteração, os produtos internos são calculados a fim de satisfazer certas condições de ortogonalidade. O algoritmo CG armazena um total de quatro vetores (x , p , Ap e r), além da matriz A de coeficientes. As operações realizadas a cada iteração são: dois produtos internos, o produto matriz-vetor (Ap_j) e três fórmulas de recorrência (atualizações de vetores).

5.2.2 Método Conjugado Residual

O CG é utilizado para matriz simétrica, definida positiva. Similarmente, um novo algoritmo foi desenvolvido, mas desta vez para trabalhar com matrizes Hermitianas (SAAD, 2003) e denomina-se método Conjugado Residual (CR). Trata-se de um exemplo, onde o vetor resíduo agora é A -ortogonal, isto é, conjugado. Nesse caso, o vetor Ap_j também é ortogonal. O algoritmo é detalhado no pseudo-código a seguir.

Conjugado Residual

1. *Calcular* $r_0 = b - Ax_0$,
 $p_0 = r_0$.
2. *Para* $j = 0, 1, \dots$, *até convergir* :
3.
$$\alpha_j = \frac{(r_j, Ar_j)}{(Ap_j, Ap_j)}$$
4.
$$x_{j+1} = x_j + \alpha_j p_j$$
5.
$$r_{j+1} = r_j - \alpha_j Ap_j$$
6.
$$\beta_j = \frac{(r_{j+1}, Ar_{j+1})}{(r_j, Ar_j)}$$
7.
$$p_{j+1} = r_{j+1} + \beta_j p_j$$
8.
$$Ap_{j+1} = Ar_{j+1} + \beta_j Ap_j$$
9. *Fim* (5.5)

Na última linha do algoritmo, o vetor Ap_j é obtido diretamente do produto Ar_j , sem a necessidade de calcular um produto matriz-vetor adicional. Além da matriz de coeficientes A , o algoritmo CR armazena um total de cinco vetores (x , p , Ap , Ar e r). As operações realizadas a cada iteração são: dois produtos internos, o produto matriz-vetor (Ar_j) e quatro fórmulas de recorrência (atualizações de vetores). De acordo com Saad (2003), o CR tem um comportamento similar ao CG em termos de convergência, entretanto, o CG é geralmente preferido ao algoritmo CR.

5.2.3 Método Gradiente Biconjugado

O método Gradiente Biconjugado é uma variação do método do Gradiente Conjugado, sendo que o CG não é utilizado para sistemas não simétricos porque os vetores residuais não podem ser feitos ortogonais em poucas iterações (SAAD, 2003). O BiCG é necessário substituir a sequência ortogonal do resíduo por duas sequências mutuamente ortogonais e não realizar mais

a minimização. A atualização do resíduo aumenta neste método quando comparado ao CG, sendo necessário duas sequências de resíduos, tal como demonstrado no pseudo-código abaixo.

Gradiente Biconjugado

1. *Calcular* $r_0 = b - Ax_0$,
2. *Escolher* r_0^* desde que $(r_0, r_0^*) \neq 0$,
 $p_0 = r_0$,
 $p_0^* = r_0^*$.
3. *Para* $j = 0, 1, \dots$, *até convergir* :
4.
$$\alpha_j = \frac{(r_j, r_j^*)}{(Ap_j, p_j^*)}$$
5.
$$x_{j+1} = x_j + \alpha_j p_j$$
6.
$$r_{j+1} = r_j - \alpha_j Ap_j$$
7.
$$r_{j+1}^* = r_j^* - \alpha_j A^T p_j^*$$
8.
$$\beta_j = \frac{(r_{j+1}, r_{j+1}^*)}{(r_j, r_j^*)}$$
9.
$$p_{j+1} = r_{j+1} + \beta_j p_j$$
10.
$$p_{j+1}^* = r_{j+1}^* + \beta_j p_j^*$$
11. *Fim* (5.6)

As variáveis α_j e β_j também são escalares. Esse método requer o produto matriz-vetor Ap_j e também o produto $A^T p_j^*$, onde A^T é a matriz transposta. Para matriz simétrica definida positiva o método tem o mesmo número de iteração que o Gradiente Conjugado, entretanto o custo computacional é duas vezes maior devido aos armazenamentos extras que são necessários.

O algoritmo BiCG armazena um total de seis vetores (x , r , r^* , Ap , p e p^*), além das matrizes de coeficientes A e A^T . As operações realizadas a cada iteração são: dois produtos internos, os produtos matriz-vetor (Ap_j) e $(A^T p_j^*)$, além de cinco fórmulas de recorrência (atualizações de vetores).

De acordo com Jin (2002) o método Gradiente Biconjugado pode ser implementado sem a necessidade de calcular a matriz transposta de A . Nesse caso, isso só é possível quando o sistema linear é simétrico. O pseudo-código que implementa esse método está disponível a seguir na Eq. (5.7).

Gradiente Biconjugado (Matriz Simétrica)

1. *Calcular* $r_0 = b - Ax_0$,
 $p_0 = r_0$.
 $R_{norm} = (r_0, r_0)$.
2. *Para* $j = 0, 1, \dots$, *ate' convergir* :
3. $\alpha_j = \frac{R_{norm}}{(Ap_j, p_j)}$
4. $x_{j+1} = x_j + \alpha_j p_j$
5. $r_{j+1} = r_j - \alpha_j Ap_j$
6. $\gamma_j = \frac{(r_{j+1}, r_{j+1})}{R_{norm}}$
7. $p_{j+1} = r_{j+1} + \gamma_j p_j$
8. *Fim* (5.7)

Esse método converge muito mais rápido que o BiCG tradicional o qual necessita da transposta da matriz A . A maior vantagem sobre o método tradicional é que esse realiza somente um produto matriz-vetor.

5.2.4 Método Gradiente Conjugado ao Quadrado

O algoritmo CGS foi desenvolvido por Sonneveld em 1989, para evitar o cálculo da transposta da matriz A que é realizado no BiCG. Pode-se conseguir uma convergência mais rá-

pida neste método, considerando aproximadamente o mesmo custo computacional que o BiCG (SAAD, 2003). O pseudo-código está disponível na Eq. (5.8).

Gradiente Conjugado ao Quadrado

1. *Calcular* $r_0 = b - Ax_0$,
2. *Escolher* r_0^* *arbitrário*,
 $p_0 = u_0 = r_0$.
3. *Para* $j = 0, 1, \dots$, *até convergir* :
4. $\alpha_j = \frac{(r_j, r_0^*)}{(Ap_j, r_0^*)}$
5. $q_j = u_j - \alpha_j Ap_j$
6. $x_{j+1} = x_j + \alpha_j (u_j + q_j)$
7. $r_{j+1} = r_j - \alpha_j A (u_j + q_j)$
8. $\beta_j = \frac{(r_{j+1}, r_0^*)}{(r_j, r_0^*)}$
9. $u_{j+1} = r_{j+1} + \beta_j q_j$
10. $p_{j+1} = u_{j+1} + \beta_j (q_j + \beta_j p_j)$
11. *Fim* (5.8)

O CGS funciona muito bem em muitos casos, no entanto, existe uma dificuldade em trabalhar com a relação de contração ao quadrado. Neste caso, os erros de arredondamento tendem a ser mais prejudiciais que o algoritmo padrão, BiCG.

Neste algoritmo, além dos vetores e escalares comuns a todos os métodos anteriores, também são necessários calcular o vetor auxiliar de direção u_j e os vetores de direção. Além disso, são necessárias duas multiplicações matriz-vetor, dois produtos internos e seis fórmulas de recorrência, entretanto não é necessário utilizar a matriz transposta de A .

5.2.5 Método Gradiente Biconjugado Estabilizado

O método Gradiente Biconjugado Estabilizado é uma variação do método CGS (BARRETT et al., 2006) que foi desenvolvido para resolver sistema linear não simétrico e para evitar a convergência irregular que pode ocorrer no CGS. O pseudo-código está disponível na Eq. (5.9).

Gradiente Biconjugado Estabilizado

1. *Calcular* $r_0 = b - Ax_0$,
2. *Escolher* r_0^* *arbitrário*,
 $p_0 = r_0$.
3. *Para* $j = 0, 1, \dots$, *até* *convergir* :
4. $\alpha_j = \frac{(r_j, r_0^*)}{(Ap_j, r_0^*)}$
5. $s_j = r_j - \alpha_j Ap_j$
6. $\omega_j = \frac{(As_j, s_j)}{(As_j, s_j)}$
7. $x_{j+1} = x_j + \alpha_j p_j + \omega_j s_j$
8. $r_{j+1} = s_j - \omega_j As_j$
9. $\beta_j = \frac{(r_{j+1}, r_0^*)}{(r_j, r_0^*)} \times \frac{\alpha_j}{\omega_j}$
10. $p_{j+1} = r_{j+1} + \beta_j (p_j - \omega_j Ap_j)$
11. *Fim* (5.9)

Quando a matriz transposta de A não estiver disponível, esse método pode ser uma boa opção. Ele contém a atualização da variável ω_j que tem como objetivo minimizar o resíduo r_j , similar ao método CG. O vetor residual é minimizado e leva a uma convergência mais suave. Em alguns casos, a convergência pode ser estagnada, o que não permite a resolução do sistema linear.

Este método requer os produtos matriz-vetor Ap_j e As_j , além de quatro produtos internos (dois a mais que o CG e BiCG) e de cinco fórmulas de recorrência. De um lado, o BiCGSTAB requer o cálculo de quatro produtos internos, entretanto, por outro lado, não precisa armazenar a transposta da matriz A , tal como o BiCG. Esse método, contém dois critérios de parada, sendo o primeiro relacionado a norma de s_j e o segundo relacionado a norma de r_j .

5.3 Pré-condicionadores

O objetivo do pré-condicionador é reduzir o número de condição do sistema, produzindo uma convergência mais rápida no processo iterativo (BARRETT et al., 2006). Uma grande quantidade de pré-condicionadores é baseada em decomposições incompletas da matriz de coeficientes. Uma decomposição é conhecida como incompleta quando certos elementos de preenchimento (diferentes de zero na decomposição) são ignorados em algumas posições onde a matriz original tem um valor zero correspondente. Os pré-condicionadores IC e ILU entram nesta classificação. A eficiência do pré-condicionador (M) depende de quão bem M^{-1} se aproxima de A^{-1} .

Nas decomposições incompletas podem ocorrer alguns erros durante o processo os quais são conhecidos como:

- ❑ *break-down*, que é ocasionado pela divisão por um pivô nulo, ou
- ❑ resultar em matrizes indefinidas que têm pivô negativo.

Os pré-condicionadores, como o ILU e o IC, têm um custo no processo de decomposição e exigem um armazenamento adicional para a nova matriz pré-condicionada.

O sistema linear oriundo do Método de Elementos Finitos pode ser representado pela seguinte equação:

$$[A] \cdot [x] = [b] \quad (5.10)$$

Esse sistema linear, para o problema governado pela equação de onda e discretizado através da formulação $A - V$, apresenta a característica de ser muito mal-condicionado, especialmente para excitação de correntes em altas frequências (MHz). Isto ocorre principalmente por:

- ❑ grande contraste de dimensões em diferentes partes do domínio de estudo;
- ❑ grande contraste entre propriedades físicas dos meios constituintes;
- ❑ dificuldade em se obter malha de boa qualidade (elementos deformados, "pontudos", muito alongados, etc);

- técnicas de truncamento do domínio aberto, como por exemplo, o uso de absorvedores anisotrópicos utilizados para truncar o domínio (PML).

A fim de se contornar esses problemas é necessário aplicar o pré-condicionador M , de tal forma que o novo sistema da Eq. (5.10) possa ser reescrito, tal como:

$$[M^{-1}] [A] [x] = [M^{-1}] [b] \quad (5.11)$$

Para as formulações introduzidas no capítulo anterior, os pré-condicionadores mais eficientes são os que utilizam a decomposição incompleta LU e a de Cholesky (BARRETT et al., 2006). No primeiro caso, a matriz M é obtida por:

$$[M] = [L] [U] \quad (5.12)$$

sendo L , a matriz triangular inferior e U , a matriz triangular superior. Mais detalhes do desenvolvimento do método podem ser encontrados em Saad (2003) e em Golub e Van Loan (1989). Já no segundo caso, a matriz M é determinada da seguinte forma:

$$[M] = [L] [L^T] \quad (5.13)$$

sendo L , a matriz triangular inferior e L^T , a sua transposta. Essas matrizes têm a mesma esparsidade da matriz A .

Entretanto, as vezes a decomposição incompleta não garante que a matriz simétrica, definida positiva possa ser decomposta. Todos os resultados que garantem a existência dependem de alguma forma de dominância da diagonal (SAAD, 2003). Nesse caso, em alguns problemas é necessário adicionar um fator na diagonal de tal forma que ela se transforme em dominante e a decomposição seja possível. Para minimizar essa dificuldade utiliza-se uma compensação em $[M]$ por meio de um fator, α , real e positivo (SAAD, 2003), tal que:

$$[M(\alpha)] = M + \alpha [I] \quad (5.14)$$

A abordagem dos pré-condicionadores pode parecer atraente por causa do sucesso das fatorações incompletas IC e ILU. Entretanto, nem sempre essa decomposição é encontrada. Uma questão frequentemente discutida é a forma de encontrar bons valores para o α . Estudo mais detalhado sobre isso pode ser encontrado em Saad (2003) e em Fujiwara et al. (1993). Uma sugestão é selecionar o valor menor possível que permita que a diagonal da matriz compensada seja dominante. Infelizmente, encontrar o valor adequado de α não é uma tarefa fácil. Saad (2003) propôs alguns valores dessa variável em função do número de iterações, tal como ilustrado na Fig. 5.1.

Figura 5.1 – Número de iterações em função dos valores do *shift* – α

O que se pode observar na Fig. (5.1) é que o número de passos necessários para a convergência começa a decrescer com o aumento do α , e então, logo em seguida, aumenta novamente. Neste trabalho, o valor adotado desse fator *shift* foi de 0,1 (FUJIWARA et al., 1993). Mesmo assim, o número de iterações ainda segue elevado, porém substancialmente menor.

Exemplos dos perfis das matrizes após a aplicação das duas decomposições incompletas podem ser visualizados na Fig. 5.2.

Figura 5.2 – Perfis das matrizes antes (A) e após as decomposições incompletas ILU (L e U) e IC (M).

Por exemplo, para a decomposição IC, a matriz A está identificada na parte inferior da Fig. 5.2. As matrizes triangulares inferior e superior, L e U repectivamete, estão ilustradas na parte superior dessa figura e contêm a mesma estrutura da matriz A . Se a decomposição existir, então a matriz resultante M terá o mesmo padrão de esparsidade de A , tal como ilustrado na parte inferior direita da figura e identificada como M .

5.3.1 Decomposição Incompleta de Cholesky

Um dos mais importantes pré-condicionadores é conhecido como decomposição incompleta de Cholesky (GOLUB; VAN LOAN, 1989). De uma forma geral, quando a matriz de coeficiente A for simétrica e definida positiva, ou seja, com forma quadrática $v^T A v > 0, \forall v \neq 0$, então A pode ser decomposta tal como:

$$[M] = [L] [L^T] \quad (5.15)$$

em que L é a matriz triangular inferior e L^T a matriz triangular transposta. É necessário armazenar a matriz A e a matriz M , além da esparsidade da matriz decomposta ser mantida semelhante da matriz A . Assim, de acordo com Golub e Van Loan (1989) pode-se obter a decomposição através do pseudo-código descrito na Eq. (5.16).

1. Para $k = 1 : n$
2. $A(k, k) = \sqrt{A(k, k)}$
3. Para $i = k + 1 : n$
4. Se $A(i, k) \neq 0$
5. $A(i, k) = \frac{A(i, k)}{A(k, k)}$
6. Fim
7. Fim
8. Para $j = k + 1 : n$
9. Para $i = j : n$
10. Se $A(i, j) \neq 0$
11. $A(i, j) = A(i, j) - A(i, k)A(j, k)$
12. Fim
13. Fim
14. Fim
15. Fim

(5.16)

Dentro da resolução do sistema de equações lineares é necessário realizar as substituições progressivas e regressivas as quais estão relacionadas com a matriz M , tal como:

$$LL^T x = b \quad (5.17)$$

$$L^T y = b \quad (5.18)$$

$$Lx = y \quad (5.19)$$

A decomposição incompleta de Cholesky pode ser determinada sem extrair a raiz quadrada. Detalhes dessa implementação podem ser encontrados em Saad (2003).

5.3.2 Decomposição Incompleta LU

Ao utilizar a decomposição incompleta LU sem preenchimento, ILU(0) (GOLUB; VAN LOAN, 1989), a matriz M pode ser decomposta da seguinte forma:

$$[M] = [L][U] \quad (5.20)$$

em que L é a matriz triangular inferior e U é a matriz triangular superior. A diagonal de L é unitária e não precisa de armazenamento extra. Para resolver o sistema contido na Eq. (5.10) é necessário armazenar as matrizes A e M . Assim, de acordo com Saad (2003), pode-se obter a decomposição através do pseudo-código descrito na Eq. (5.21).

$$\begin{aligned}
 &1. \quad \text{Para } i = 2 : n \\
 &2. \quad \quad \text{Para } k = 1 : i - 1 \\
 &3. \quad \quad \quad \text{Se } A(i, k) \neq 0 \\
 &4. \quad \quad \quad \quad A(i, k) = \frac{A(i, k)}{A(k, k)} \\
 &5. \quad \quad \quad \text{Para } j = k + 1 : n \\
 &6. \quad \quad \quad \quad \text{Se } A(i, j) \neq 0 \\
 &7. \quad \quad \quad \quad \quad A(i, j) = A(i, j) - A(i, k)A(k, j) \\
 &8. \quad \quad \quad \quad \text{Fim} \\
 &9. \quad \quad \text{Fim} \\
 &10. \quad \text{Fim}
 \end{aligned} \quad (5.21)$$

Dentro da resolução do sistema de equações lineares é necessário realizar as substituições progressivas e regressivas, as quais estão relacionadas com a matriz M , tal como:

$$LUx = b \quad (5.22)$$

$$Uy = b \quad (5.23)$$

$$Lx = y \quad (5.24)$$

5.3.3 Método do Gradiente Conjugado Pré-condicionado

Para resolver o método do Gradiente Conjugado pré-condicionado é necessário que a matriz A seja simétrica e definida positiva (GOLUB; VAN LOAN, 1989). Ele é similar ao método contido na Eq. (5.4), no entanto, neste caso é necessário introduzir a matriz pré-condicionada M . Assim, o algoritmo da Eq. (5.4) deve ser modificado de tal forma que o valor adicional de z seja determinado, tal como no pseudo-código a seguir.

Gradiente Conjugado Pré-condicionado

1. *Calcular* $r_0 = b - Ax_0$,
 $z_0 = M^{-1}r_0$,
 $p_0 = z_0$.
2. *Para* $j = 0, 1, \dots$, *até convergir* :
3. $\alpha_j = \frac{(r_j, z_j)}{(Ap_j, p_j)}$
4. $x_{j+1} = x_j + \alpha_j p_j$
5. $r_{j+1} = r_j - \alpha_j Ap_j$
6. $z_{j+1} = M^{-1}r_{j+1}$
7. $\beta_j = \frac{(r_{j+1}, z_{j+1})}{(r_j, z_j)}$
8. $p_{j+1} = z_{j+1} + \beta_j p_j$
9. *Fim* (5.25)

Para que esse método seja eficiente, $M = z r$ deve ser facilmente resolvido e a convergência rápida. Nesse caso, é necessário também calcular dois produtos internos, o produto matriz-vetor (Ap_j) e uma fórmula de recorrência adicional que no algoritmo CG. Logo, quatro atualizações de vetores são necessárias. Outros métodos, como por exemplo, o Gradiente Biconjugado e o Gradiente Biconjugado Estabilizado, ambos pré-condicionados, podem se encontrados com mais detalhe em Barrett et al. (2006). Os pré-condicionadores mais eficientes para os problemas deste trabalho foram o ILU e o IC.

5.4 Método de Reordenação Cuthill-McKee Reverso

O método Cuthill-McKee Reverso (RCM) é eficiente para permutar-limitar o perfil da matriz. Ele é usado com frequência para reduzir a largura de banda da matriz. A banda das linhas da matriz é praticamente igual ao número da entrada da matriz, considerando a primeira e a última entrada diferente de zero na linha, com a condição de que a diagonal é sempre tratada como diferente de zero (GEORGE; LIU, 1981), (GILBERT; MOLER; SCHREIBER, 1992). Outros detalhes desse método serão apresentados nos próximos capítulos.

5.5 Formato de Acesso e Manipulação das Matrizes

Para utilizar CUDA em sistemas lineares esparsos é necessário que as matrizes sejam representadas em determinados formatos para tratar a esparsidade. Para tanto, deve-se utilizar alguns métodos de acesso e manipulação dos elementos não nulos das matrizes. Existem vários métodos específicos, sendo que eles se distinguem nos quesitos de armazenamento dos vetores, nas características computacionais e no método de acesso e manipulação das entradas das matrizes (BELL; GARLAND, 2008). Esses métodos armazenam basicamente os números de elementos não nulos da matriz e seus índices. De acordo com Bell e Garland (2008) os métodos mais utilizados são: o *Coordinate* (COO), o *Compressed Sparse Row* (CSR) e o *Hybrid* (HYB).

5.5.0.1 Formato *Coordinate*

Assume-se que a matriz esparsa A , do sistema linear $Ax = b$, seja armazenada em *row-major*, ou seja, o índice do *array* é primeiramente ordenado por índice de linha e, em seguida, dentro da mesma linha, por índice de coluna. Assim, antes de utilizar esse formato é necessário reorganizar a matriz de tal forma que ela esteja armazenada na CPU por linha, e não por coluna.

Por exemplo, seja o sistema matricial de ordem 4×4 :

$$\begin{bmatrix} 1 & 4 & 0 & 0 \\ 0 & 2 & 3 & 0 \\ 5 & 0 & 0 & 7 \\ 0 & 0 & 9 & 0 \end{bmatrix}, \quad (5.26)$$

Caso os índices das linhas e colunas sejam base zero, o formato COO adquire 3 *arrays*, tais como:

$$cooValA = \begin{bmatrix} 1 & 4 & 2 & 3 & 5 & 7 & 9 \end{bmatrix}, \quad (5.27)$$

$$cooRowA = \begin{bmatrix} 0 & 0 & 1 & 1 & 2 & 2 & 3 \end{bmatrix} e \quad (5.28)$$

$$cooColA = \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 3 & 2 \end{bmatrix}. \quad (5.29)$$

O primeiro vetor se refere a todos os elementos não nulos da matriz, sendo que o segundo e o terceiro se referem aos índices das linhas e colunas onde esses elementos se posicionam na matriz original.

5.5.0.2 Formato *Compressed Sparse Row*

A única diferença desse formato para o anterior, COO, é o segundo vetor. Neste caso, o vetor contém todos os primeiros elementos não nulos de todas as linhas, em ordem crescente de contagem. O último elemento do vetor é sempre o último valor da contagem dos elementos, acrescido de 1. Por exemplo, para a mesma matriz esparsa anterior, definida como:

$$\begin{bmatrix} 1 & 4 & 0 & 0 \\ 0 & 2 & 3 & 0 \\ 5 & 0 & 0 & 7 \\ 0 & 0 & 9 & 0 \end{bmatrix}, \quad (5.30)$$

considerando índice com base zero, o formato CSR adquire 3 *arrays*, dados por:

$$cooValA = \begin{bmatrix} 1 & 4 & 2 & 3 & 5 & 7 & 9 \end{bmatrix}, \quad (5.31)$$

$$csrRowA = \begin{bmatrix} 0 & 2 & 4 & 6 & 7 \end{bmatrix} e \quad (5.32)$$

$$csrColA = \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 3 & 2 \end{bmatrix}. \quad (5.33)$$

O vetor *cooValA* inicia com o elemento 1 e ele, na contagem inicializa com valor 0. O segundo elemento é o 4 e ele, na contagem, é considerado 1. O seguinte é o elemento 2 e na contagem ele é o 2, e assim por diante. O vetor *csrRowA* terá a contagem correspondente dos elementos não nulos iniciais de cada linha, ou seja, o elemento 1, 2, 5 e 9.

O formato HYB é composto pelos formatos *Ellpack-Itpack* e COO. Esse método foi utilizado em um caso específico. Detalhes adicionais desse método podem ser encontrados em Bell e Garland (2008).

5.6 Bibliotecas de Álgebra Linear

A NVIDIA desenvolveu bibliotecas para manipulação matemática de matrizes, conhecidas como (CUBLAS, 2014), (CUSPARSE, 2014), (CUSP, 2014):

- ❑ CUBLAS;
- ❑ CUSPARSE;
- ❑ CUSP.

Cada biblioteca tem suas funcionalidades e particularidades, como por exemplo, a CUBLAS tem sub-rotinas básicas de álgebra linear (produto interno, multiplicação, soma, etc.). A CUSPARSE e a CUSP também têm uma coleção de sub-rotinas básicas, de Álgebra Linear, mas também podem ser usadas para matrizes esparsas.

A CUSP possui código aberto e fornece de forma flexível uma interface de alto nível para o programador, permitindo ainda resolução de sistemas lineares esparsos. Entretanto, essa última biblioteca é bem limitada quando se trabalha com números complexos e com matrizes mal condicionadas. Como existem vários algoritmos disponíveis, é mais complicado alterar qualquer configuração que se queira para se adequar a alguma particularidade específica das matrizes. Para matrizes com números reais, essa biblioteca pode ajudar em alguns casos. Por exemplo, ela contém rotinas dos seguintes métodos de solução do sistema linear: CG, BiCG, BiCGStab, dentre outros. Também tem disponível os pré-condicionadores: *Multigrid* Algébrico, Diagonal e Inversa Aproximada.

Todas essas bibliotecas foram utilizadas neste trabalho. A CUBLAS e a CUSPARSE foram utilizadas para implementar todos os sistemas iterativos, sendo que cada passo desses métodos requer uma determinada função e uma chamada de *Kernel*, ou seja, uma execução na GPU. A CUSP foi utilizada para resolver matrizes que contém números reais e para testar um ou outro pré-condicionador específico. Com essa última biblioteca não se conseguiu resolver nenhum sistema linear a valores complexos.

5.7 Síntese do Capítulo

Neste capítulo foram apresentados os principais métodos para resolução do sistema de equações lineares. Esses métodos apresentam algumas sequências de atualização de vetores que são similares, entretanto cada um tem suas particularidades. Para matrizes simétricas o CG é o mais adequado, entretanto os outros métodos podem ser utilizados também. Quando não se dispõe da matriz transposta de A , então o BiCG não se torna tão atrativo quanto os outros. Além desses, foram apresentadas as técnicas de decomposição incompleta LU e Cholesky, os métodos de armazenamento das matrizes e as bibliotecas da NVIDIA. O capítulo seguinte irá abordar as características dos casos-teste.

Metodologias e Definição dos Casos-teste

6.1 Introdução

Neste capítulo serão apresentados os casos-teste implementados no trabalho. Como já visto anteriormente, dois estudos de caso foram propostos, sendo o primeiro relacionado à um problema de aterramento elétrico no regime estático, o qual foi modelado utilizando o Método de Elementos Finitos nodal. O sistema de equações resultante é a coeficientes reais. Já o segundo, se refere à um problema de aterramento elétrico com injeção de corrente harmônica, sendo modelado pelo Método de Elementos Finitos aresta-nodal. Neste último caso, o potencial vetor magnético e o potencial elétrico foram determinados a partir de um único sistema de equações algébricas, a coeficientes complexos.

6.2 Metodologia dos Casos-teste

Essa seção se destina a apresentar toda a metodologia utilizada para implementar computacionalmente, e de forma sequencial e paralela, os métodos iterativos para solução do sistema linear oriundo do método de Elementos Finitos.

6.2.1 Principais Etapas Computacionais

O desenvolvimento da ferramenta computacional proposta envolve diversas etapas, sendo uma parte considerável na preparação dos dados, que correspondem aos sistemas matriciais de equações oriundos do MEF. Essas etapas são descritas a seguir e ilustradas na Fig. (6.1).

- ❑ **Passo 1:** Gerar a malha em elementos finitos. Foi utilizado o software GMSH (GEUZAINÉ; REMACLE, 2014), disponível gratuitamente para fins acadêmicos, para criar a

geometria, a malha e analisar o pós-processamento. Qualquer especificação dos módulos de entrada é feito de forma interativa com o usuário e em arquivos ASCII ou texto, utilizando linguagem *script* própria.

- ❑ **Passo 2:** Após utilizar o software para gerar a malha, um arquivo de texto é gerado contendo as coordenadas dos nós dos elementos e a tabela de conectividade. Esse arquivo alimenta um outro programa, desenvolvido no LMAG-EPUSP (SILVA et al., 2007), (SILVA et al., 2011) responsável pela simulação por elementos finitos dos problemas modelados e discretizados com GMSH. Esse programa é utilizado para gerar as matrizes locais e realizar a montagem do sistema global. É neste passo que o sistema de equações algébricas é obtido. Até esse momento, todo o processo é montado de forma sequencial utilizando a CPU do computador.
- ❑ **Passo 3:** Essa etapa envolve a resolução do sistema linear. Os sistemas de equações foram resolvidos utilizando os métodos iterativos: Gradiente Conjugado, Gradiente Biconjugado, Conjugado Residual, Gradiente Biconjugado Estabilizado, Gradiente Conjugado para equações normais e Gradiente Conjugado ao Quadrado (JIN, 1993), (SAAD, 2003). Agregados a esses métodos, foram utilizados os pré-condicionadores conhecidos como: decomposição incompleta LU, decomposição incompleta de Cholesky, *Multigrid* Algébrico, Diagonal e Inversa Aproximada (SAAD, 2003). Para as matrizes envolvendo número reais, o sistema linear foi paralelizado utilizando CUDA, inclusive os pré-condicionadores. Também foi utilizada a técnica OpenMP, mas somente na multiplicação matriz-vetor. Para as matrizes envolvendo números complexos, o sistema linear foi solucionado utilizando CUDA, entretanto os pré-condicionadores utilizados foram aqueles disponíveis na biblioteca LMAGLIB (IC) e no software MATLAB (ILU), de forma sequencial.
- ❑ **Passo 4:** Análise da performance computacional utilizando várias métricas, como por exemplo, tempo de processamento sequencial e paralelo, speedup, escalabilidade, dentre outras.
- ❑ **Passo 5:** Análise dos resultados do potencial vetor magnético e do potencial elétrico e validação mediante comparação com uma referência.

A geração da malha, assim como a montagem do sistema global esparso, foram executados no *host* (CPU), enquanto que a resolução do sistema linear foi solucionado ora no *host* (sequencial) ora no *device* (GPU), tal como ilustrado na Fig. (6.1). No caso do *device*, a GPU foi alocada e os dados foram copiados da memória da CPU para a memória da GPU e vários *kernels* foram invocados para calcular cada passo do sistema iterativo. Após finalizar a resolução do sistema linear na GPU, os resultados foram então copiados de volta para a memória da CPU e analisados.

Figura 6.1 – Fluxograma da resolução paralela do sistema $Ax=b$, com as etapas realizadas na CPU e na GPU

Para a solução do sistema iterativo, foi adotado em todos os casos uma tolerância residual igual a 10^{-6} e a diagonal da matriz foi compensada através da aplicação de um fator *shift* igual a 0,1.

Além da biblioteca LMAGLIB, foi utilizada a biblioteca *Matrix Template Library* (MTL) e *Iterative Template Library* (ITL) (LUMSDANIE; SIEK, 1998).

6.2.2 Estrutura do Algoritmo Paralelo

A Fig. (6.2) ilustra um pseudo-código que pode ser facilmente implementado ao utilizar a biblioteca CUSP e CUDA. Esse algoritmo ilustra as etapas e as sequências necessárias para resolver um problema, por exemplo para o regime estático, como segue:

1. Criar a estrutura da matriz esparsa (formato HYB).
2. Carregar a matriz.
3. Alocar a memória da GPU com o vetor de solução x e o vetor b .
4. Resolver o sistema linear $A*x = b$ ao utilizar o método Gradiente Conjugado.

A saída desse código é o vetor x de solução do sistema linear. Entretanto, quando se utiliza as bibliotecas CUBLAS e CUSPARSE a complexidade do algoritmo aumenta bastante. Nesse caso, é necessário definir os passos como são apresentados na Fig. (6.3).

```
#include <cusp/hyb_matrix.h>
#include <cusp/io/matrix_market.h>
#include <cusp/krylov/cg.h>

int main(void)
{
    // Cria a estrutura da matriz esparsa (Formato HYB)
    cusp::hyb_matrix<int, float, cusp::device_memory> A;

    // Carrega a matriz armazenada no formato MatrixMarket
    cusp::io::read_matrix_market_file(A, "teste.mtx");

    // Aloca memória para o vetor de solução (x) e o vetor (b)
    cusp::array1d<float, cusp::device_memory> x(A.num_rows, 0);
    cusp::array1d<float, cusp::device_memory> b(A.num_rows, 1);

    // Resolve o sistema linear  $A * x = b$  utilizando o método Gradiente Conjugado
    cusp::krylov::cg(A, x, b);

    return 0;
}
```

Figura 6.2 – Exemplo de código fonte para a resolução paralela em GPU de sistema de equações $A.x=b$ utilizando o método CG da biblioteca CUSP

De uma forma geral, a Fig. (6.3) contém a estrutura para desenvolver uma algoritmo de forma paralela ao utilizar as bibliotecas CUBLAS e CUSPARSE. Todas as etapas são importantes para finalizar e obter a solução do problema de aterramento elétrico, tanto no regime estático quanto no harmônico.

Ao declarar as variáveis é necessário conhecer a estrutura de cada matriz para facilitar o seu armazenamento. Deve-se conhecer previamente se a matriz é cheia ou triangular (superior ou inferior), se a diagonal é unitária ou não, além de outras características. As quatro primeiras etapas da Fig. (6.3) são similares as duas primeiras enumeradas acima.

Os *Kernels* se referem ao processo iterativo que é somente realizado após as etapas anteriores estiverem finalizadas. Esse processo inclui todos os passos de cada pseudo-código ilustrado no capítulo anterior.

Figura 6.3 – Fluxograma do algoritmo paralelo para solução pelo método CG do sistema $A \cdot x = b$ quando se utilizam as bibliotecas CUBLAS e CUSPARSE

6.2.3 Reordenação Cuthill-McKee Reverso

Para a solução do sistema linear esparso, duas situações foram consideradas: (i) sem reordenação e, (ii) com reordenação. O método de reordenação utilizado foi o Cuthill-McKee Reverso aplicado a fim de se reduzir a banda da matriz (GEORGE; LIU, 1981), (GILBERT et al., 1992).

A Fig. (6.4) ilustra o efeito da reordenação da matriz de um dos problemas regidos pelo regime harmônico. Ao analisá-la pode-se perceber que a primeira matriz representa o sistema original oriundo do Método de Elementos Finitos aresta-nodal, e a segunda, ilustra a matriz com redução de banda ao utilizar o RCM. É importante salientar que os elementos não nulos da matriz original A , são reestruturados e recebem novas numerações, tal como o vetor b e as incógnitas do sistema. Entretanto, a quantidade de elementos diferente de zero continua sendo a mesma. Esse método é muito importante para a solução do sistema linear em questão porque acelera a convergência do sistema iterativo.

Figura 6.4 – Efeito da aplicação da reordenação RCM (a) gerada pela aplicação da formulação em dois potenciais do MEF de aresta (b) sobre a matriz A

6.3 Configurações de Hardware

Todos os testes deste trabalho foram desenvolvidos em três diferentes configurações de máquinas, detalhadas na Tab. 6.1.

Tabela 6.1 – Características das configurações de hardware utilizadas

Máquina	Processador	CPU (GB)	Modelo GPU	Núcleos GPU	GB/s GPU	GPU (GB)
H1	i5 - 3,2	16	GT 240	96	54,4	0,512
H2	i7 - 3,4	32	GTX 690	3072	384	4 (2+2)
H3	i7 - 3,4	32	GTX TITAN	2688	288,4	6

Essas três máquinas têm características bem distintas, como por exemplo, a configuração da placa de vídeo. A primeira tem 512 MB de memória na GPU, a segunda tem 4 GB de memória, sendo que esta contém duas placas de vídeo separadas e integradas em uma única máquina (cada uma com 2 GB), e a terceira contém 6 GB de memória em uma placa única.

Todos os algoritmos foram compilados utilizando o sistema operacional MS-Windows 7 e o compilador MS-Visual Studio 2010, da MICROSOFT. Para os problemas regidos pelo regime estático, utilizou-se precisão aritmética simples e, nos problemas regidos pelo regime harmônico, precisão dupla. Nesse último caso, a precisão simples não é suficiente para resolver o sistema linear oriundo do MEF, para esse problema de aterramento específico. Para compilar o algoritmo de forma paralela utilizou-se a versão 5.0 CUDA, da NVIDIA (NVIDIA, 2014).

6.4 Detalhes da Implementação em CUDA

Esta seção destina-se a apresentar de forma simples como um algoritmo é gerado e compilado ao utilizar CUDA e GPU, pois aí reside toda a complexidade da migração para essa nova

arquitetura. Assim, de forma geral, para criar um primeiro algoritmo que utiliza CUDA os seguintes passos são necessários:

1. Identificar se o computador já possui uma placa de vídeo. Para isso é necessário ir em "Painel de Controle" => "Gerenciador de Dispositivo" => "Adaptadores de Vídeo". Caso tenha alguma placa de vídeo instalada é necessário que se verifique se essa placa é da NVIDIA. No site da NVIDIA (NVIDIA, 2014) é possível verificar todos os modelos, inclusive os lançamentos e as configurações individuais de cada uma. É importante verificar também a memória da GPU porque dependendo do volume de dados, isso pode ser tornar um limitante;
2. Definir qual sistema operacional irá utilizar. O MS-Windows 7 foi escolhido pois adapta-se muito bem a CUDA.
3. Definir o compilador. O Visual Studio 2010 da Microsoft, pacote C++, já vem com a opção de integrar CUDA ao iniciar um programa. Assim, é importante criar um algoritmo com extensão em CUDA para que todas as funcionalidades estejam disponíveis.
4. Instalar o pacote NVIDIA *Computing Toolkit* o qual está disponível no site da NVIDIA (<https://developer.nvidia.com/cuda-toolkit>). Esse pacote fornece um ambiente de desenvolvimento abrangente em C e C++ para os desenvolvedores construírem seus aplicativos acelerados por GPU. Esse pacote inclui o compilador para GPU, bibliotecas matemáticas e ferramentas para *debugging* e otimização do desempenho dos aplicativos. Além disso, é possível obter guias de programação, manuais de usuário e outras documentações para iniciantes que queiram começar a desenvolver suas aplicações utilizando GPU.
5. Conferir se a integração do MS-Visual Studio 2010 com CUDA está disponível. Criar um projeto e verificar se a extensão CUDA está disponível. Se estiver, o desenvolvedor já pode começar a trabalhar com CUDA e GPU.
6. Incluir todas as demais bibliotecas importantes para a aplicação (CUSP, CUBLAS, CUSPARSE).

Após todos esses passos, um programador já está apto a desenvolver um algoritmo paralelo e analisar o desempenho computacional. Para trabalhar com matrizes esparsas ainda é necessário definir alguns passos adicionais. Assim, antes de utilizar as bibliotecas CUBLAS e CUSPARSE é necessário identificar as características das matrizes e definir se:

- ☐ são esparsas;
- ☐ são simétricas;
- ☐ o algoritmo exige a transposta ou não;

- ☐ a matriz de entrada será armazenada inteira ou somente a parte inferior L ou a superior U ;
- ☐ a matriz de entrada está gravada por linha ou coluna;
- ☐ a matriz é real ou complexa;
- ☐ a solução do sistema linear exige que os dados estejam disponíveis em precisão simples ou dupla;
- ☐ o armazenamento das posições dos elementos na matriz estão na base zero ou base um.

Vale lembrar que para utilizar as funções disponíveis nas bibliotecas CUBLAS e CUSPARSE é necessário se trabalhar somente com vetores. Assim, todas as matrizes e dados são definidos como ponteiro. Caso se queira utilizar os pré-condicionadores ILU e IC, é necessário utilizar variáveis extra, tanto nas declarações de variáveis quanto nos armazenamentos de dados. As funções dessas bibliotecas exigem alguns formatos específicos de armazenamento das matrizes, tal como citados anteriormente (COO e CSR).

6.5 Configurações Utilizadas nos Casos-teste

Esta seção se destina a apresentar uma análise mais detalhada dos estudos de caso. Todos eles envolvem a modelagem de um sistema de aterramento canônico, composto de uma haste vertical isolada, enterrada verticalmente em solo homogêneo. Na Fig. 6.5 tem-se uma visão esquemática do domínio tridimensional, objeto dos estudos.

Figura 6.5 – Ilustração esquemática do modelo usado na simulação do sistema de aterramento dos casos-teste

Fonte: SILVA, 2006

É importante observar que essa configuração de aterramento, em particular, poderia ter sido modelada em duas dimensões, com uma formulação axissimétrica, no entanto, trata-se de um caso isolado; a grande maioria dos sistemas de aterramento exige tratamento 3D.

Apesar da aparente simplicidade da configuração, a modelagem desse problema, tanto no regime estático, como no harmônico, guarda todas as dificuldades numéricas (exceto a etapa de definição e modelagem geométrica) presentes nos sistemas de aterramento mais complexos, devido principalmente aos fatores enumerados a seguir (SILVA et al., 2011):

1. Contrastes nas dimensões constituintes do domínio - eletrodos de seção transversal desprezível face ao comprimento, colocando problemas na geração de uma malha adequada, mesmo quando se utilizam formulações que usam elementos filiformes (1D).
2. Contrastes nas propriedades físicas - interface entre condutores de valores de condutividade elétrica muito díspares (cobre e solo, que pode ter várias camadas, também contrastantes entre si). Esta condição conduz a gradientes de potencial muito elevados na interface, exigindo grande refinamento na malha de elementos finitos nessa região.
3. Truncamento do domínio - Tanto a utilização de truncamento simples, como a utilização de alguma técnica de tratamento de domínios abertos, geram dificuldade na modelagem e alterações na matriz gerada. No primeiro caso, é necessário aumentar ainda mais a dimensão das fronteiras externas do domínio, piorando ainda mais o problema do contraste de dimensões e, em consequência, a qualidade da malha. No segundo caso, como por exemplo na técnica PML (SILVA et al., 2007), consegue-se reduzir o tamanho do domínio, porém às custas da introdução de mudanças nas propriedades físicas do problema (materiais absorvedores anisotrópicos fictícios).

Devido à simetria da configuração da Fig. 6.5, o modelo 3D usado para as simulações por elementos finitos limita-se a um quarto do domínio total, conforme ilustrado na Fig. 6.6. Maiores detalhes sobre as condições de contorno usadas podem ser obtidos em (SILVA, 2006).

A configuração da Fig. 6.5 foi usada nos estudos de caso em duas situações. A primeira envolve a simulação do problema mediante uma excitação DC, que gera sistemas de equações a coeficientes reais. A segunda situação é aquela em que a excitação tem variação temporal senoidal (harmônica). Esta condição é importante em estudos de compatibilidade eletromagnética e em situações de surto nos sistemas de aterramento. Dependendo das frequências envolvidas e das propriedades físicas dos meios, é necessário considerar o efeito da corrente de deslocamento, o que exige a solução da Equação de Onda completa, com perdas, não homogênea. O sistema de equações algébricas gerado pela aplicação do MEF é a variáveis complexas, e a matriz resultante é mal-condicionada, exigindo cuidado na seleção do pré-condicionador, que deve ser muito robusto.

A resolução deste sistema consumiu uma parcela considerável do desenvolvimento deste trabalho, e acreditamos que uma das principais, senão a maior, contribuição dele reside na sua solução, que não foi observada, ou não recebeu o devido destaque, em nenhum dos trabalhos pesquisados até o momento na redação deste manuscrito.

Figura 6.6 – Modelo 3D do eletrodo enterrado e representado por apenas um quarto do domínio, o qual foi malhado em hexaedros

Fonte: SILVA, 2006

Os métodos de truncamento usados foram de dois tipos: truncamento simples (sem uso de qualquer técnica de modelagem do infinito) e truncamento por PML, com parâmetros a_{max} e m iguais a 30 e 2, respectivamente (SILVA, 2006). Esses parâmetros são definidos ao modelar a geometria do problema, especificamente no gerador de malhas.

A seguinte simbologia será adotada para os parâmetros geométricos e físicos do sistema ilustrado na Fig. 6.6:

- L para o comprimento da haste (m);
- r para o seu raio (mm);
- ρ para a resistividade do solo (ohm.m);
- ϵ_r para a permissividade relativa do solo.

Todos os problemas têm dimensão externa máxima de 192 m e são excitados em corrente de módulo igual a 1000 A (valor máximo no caso senoidal).

6.5.1 Configuração 1 - Regime Estacionário

O fenômeno de repartição de correntes DC num sistema de aterramento como o ilustrado Na Fig. 6.5 é regido pela equação de Laplace, Eq. (4.5). O sistema de equações gerado pela aplicação do MEF nodal e a coeficientes reais é obtido pela Eq. (6.1). A excitação é em corrente,

dada na forma de condição de Neumann não homogênea. Na fronteira do domínio é imposta condição de Dirichlet homogênea (SILVA, 2006).

$$[K] \{V\} = \{I\}. \quad (6.1)$$

As configurações detalhadas na Tab. 6.2, com seus parâmetros geométricos e físicos, foram as selecionadas para os experimentos computacionais com os métodos de solução de sistemas de equações que serão indicados mais adiante.

Tabela 6.2 – Parâmetros geométricos e físicos dos modelo usados nos experimentos no Regime Estacionário

Configuração	L (m)	r (mm)	ρ (ohm.m)	Elemento	Ordem	Truncamento
C1A	12	4	450	tetraedro	1	—
C1B	2	10	100	hexaedro	1 e 2	—

6.5.2 Configuração 2 - Regime Harmônico

O fenômeno do escoamento de correntes de variação temporal senoidal num sistema de aterramento é governado, em sua forma mais geral, pela Equação de Onda completa, não homogênea, na forma fasorial, Eq. (4.19). A formulação do MEF escolhida para a solução desse problema foi aquela em dois potenciais: potencial vetor magnético A e potencial escalar elétrico V , sendo que a aproximação de A foi feita através de elementos de aresta e a de V , nodais. O sistema matricial oriundo dessa formulação é aquele dado pela Eq. (4.29) (Capítulo 4), sendo um sistema a coeficientes complexos.

A técnica de reodernação Cuthill-McKee Reverso foi utilizada nestes problemas, tal como ilustrada na Fig. 6.4.

As configurações detalhadas na Tab. 6.3, com seus parâmetros geométricos e físicos, foram as selecionadas para os experimentos computacionais com os métodos iterativos de solução que serão indicados mais adiante.

Tabela 6.3 – Parâmetros geométricos e físicos dos modelo usados nos experimentos no Regime Harmônico

Configuração	L (m)	r (mm)	ρ (ohm.m)	ϵ_r	Frequência de Excitação Senoidal (MHz)
C2A	32	4	450	10	1
C2B	2	10	100	3	5

Foram utilizados nos casos-teste os truncamentos simples e com uso de PML. A malha foi discretizada com elementos do tipo hexaedro e de segunda ordem (MARTINHO, 2009).

6.6 Métrica de Desempenho

O principal objetivo de uma implementação paralela é melhorar o desempenho computacional. Para verificar se uma implementação paralela obteve um bom desempenho computacional, em relação ao sequencial, é necessário utilizar a métrica conhecida como *speedup*.

6.6.1 *Speedup*

Speedup é a relação entre o tempo de execução de um programa sequencial sobre o tempo de uma execução paralela. A Lei de Amdahl nos diz que quando um problema é dividido em n partes, isso não significa que o tempo total de execução também o seja (PATTERSON; HENNESSY, 2014). Isto implica que sempre haverá uma parte do programa que não poderá ser paralelizada.

O *speedup* (S_p) pode ser representado, de uma forma geral, através da seguinte equação:

$$S_p = \frac{T_s}{T_p}, \quad (6.2)$$

onde T_s é o tempo sequencial e T_p o tempo de execução paralelo.

Existem três classificações importantes que devem ser aplicadas para o *speedup*, tais como:

- ☐ sublinear;
- ☐ superlinear;
- ☐ linear.

O *speedup* superlinear ocorre quando a relação existente entre o tempo sequencial e o tempo paralelo estiver acima do máximo teórico (linear). Essa característica é possível e pode ocorrer devido à hierarquia de memória, à memória cache e à arquitetura da máquina. Além disso, as otimizações que são aplicadas aos algoritmos desenvolvidos influenciam também no resultado do *speedup*.

6.7 Síntese do Capítulo

Neste capítulo foram abordadas todas as metodologias utilizadas para implementar os métodos iterativos de forma paralela. Além disso, apresentou-se as definições dos casos-teste e suas configurações. No próximo capítulo serão apresentados os resultados e as devidas análises.

Resultados

7.1 Introdução

Neste capítulo serão apresentados os casos-teste e os resultados obtidos ao longo deste trabalho com todas as implementações elencadas no capítulo anterior. Esses resultados foram divididos em Estudos de Caso, e numerados em ordem crescente seguindo a ordem cronológica. Todos os resultados das simulações dos exemplos exibidos foram validados mediante a comparação da solução com métodos analíticos e não serão mostrados neste trabalho. Em todos os casos os desvios em relação aos valores de referência foram da mesma ordem daqueles obtidos por outros métodos.

7.2 Detalhamento dos Métodos Utilizados nos Casos-teste

Esta seção é dedicada aos métodos adotados na solução de cada um dos problemas: regimes estático e harmônico.

Estratégia de Solução para Configuração 1

A Tab. 7.1 contém todos os estudos de caso realizados, além dos métodos iterativos e dos pré-condicionadores. Já a Tab. 7.2 inclui os formatos das matrizes, as técnicas paralelas, as bibliotecas e em quais artigos, fruto deste trabalho, os resultados foram apresentados.

Tabela 7.1 – Detalhamento dos métodos e pré-condicionador dos estudos de caso - configurações C1A e C1B

Configuração	Métodos Iterativos	Pré-condicionador	Máquina
Estudo de Caso 1	CG	—	H1
Estudo de Caso 2	CG, BiCG e CR	—	H1 e H2
Estudo de Caso 3	CG	ILU e IC	H1
Estudo de Caso 4	CG	ILU e IC	H2
Estudo de Caso 7	CG	AMG, DIA, AINV, ILU e IC	H2 e H3
Estudo de Caso 9	CG, BiCG e BiCGSTab	AMG, DIA, AINV, ILU e IC	H3

Tabela 7.2 – Detalhamento dos formatos das matrizes, técnicas paralelas, bibliotecas e referências dos estudos de caso - configurações C1A e C1B

Configuração	Formato	Técnica	Bibliotecas	Referências
Estudo de Caso 1	CSR	1 GPU	CUSP, CUBLAS e CUSPARSE	(CAMARGOS, et.al; 2013a)
Estudo de Caso 2	CSR, COO, HYB	1 GPU	CUSP, CUBLAS e CUSPARSE	(CAMARGOS, et.al; 2013b)
Estudo de Caso 3	CSR	1 GPU	CUBLAS e CUSPARSE	(CAMARGOS, et.al; 2013c)
Estudo de Caso 4	CSR	1 GPU	CUBLAS e CUSPARSE	(CAMARGOS, et.al; 2014a)
Estudo de Caso 7	CSR	1 GPU e OpenMP	CUSP, CUBLAS e CUSPARSE	(CAMARGOS, et.al; 2014d)
Estudo de Caso 9	CSR	1 GPU e OpenMP	CUSP, CUBLAS e CUSPARSE	(CAMARGOS, et.al; 2014f)

Estratégia de Solução para Configuração 2

Similarmente para as configurações anteriores, apresenta-se aqui os métodos de solução do sistema linear de vários estudos de caso (regime harmônico) que serão apresentados no decorrer deste capítulo. As Tab. 7.3 e Tab. 7.4 são similares àquelas apresentada para o caso estático.

Tabela 7.3 – Detalhamento dos métodos e pré-condicionador dos estudos de caso - configurações C2A e C2B

Configuração	Métodos Iterativos	Pré-condicionador	Máquina
Estudo de Caso 4	CG	SIC e SILU	H2
Estudo de Caso 5	CG e BiCG	SIC e SILU	H2
Estudo de Caso 6	CG, BiCG e BiCGSTab	SIC e SILU	H2
Estudo de Caso 7	CG, BiCG	SILU e SIC	H2 e H3
Estudo de Caso 8	CG, BiCG e BiCGSTab	SILU e SIC	H2
Estudo de Caso 9	CG e BiCG	SILU e SIC	H2

Tabela 7.4 – Detalhamento dos formatos das matrizes, técnicas paralelas, bibliotecas e referências dos estudos de caso - configurações C2A e C2B

Configuração	Formato	Técnica	Bibliotecas	Referências
Estudo de Caso 4	CSR	1 GPU	CUBLAS e CUSPARSE	(CAMARGOS, et.al; 2014a)
Estudo de Caso 5	CSR	1 GPU	CUBLAS e CUSPARSE	(CAMARGOS, et.al; 2014b)
Estudo de Caso 6	CSR	1 GPU	CUBLAS e CUSPARSE	(CAMARGOS, et.al; 2014c)
Estudo de Caso 7	CSR	1 GPU e 2 GPU's	CUBLAS e CUSPARSE	(CAMARGOS, et.al; 2014d)
Estudo de Caso 8	CSR	1 GPU	CUBLAS e CUSPARSE	(CAMARGOS, et.al; 2014e)
Estudo de Caso 9	CSR	1 GPU e 2 GPU's	CUBLAS e CUSPARSE	(CAMARGOS, et.al; 2014f)

7.3 Estudo de Caso 1

Neste estudo de caso foi utilizada a configuração C1A, detalhada no capítulo anterior, além da máquina H1 (CAMARGOS, 2013a). Dois níveis de discretização, nomeados MS1 e MS2, respectivamente 325.992 e 682.689 elementos não nulos da matriz, foram utilizados em 4 implementações. As duas primeiras são sequenciais (executadas na CPU), enquanto que as duas últimas foram processadas na GPU. Os métodos são aqueles detalhados na seção anterior. Os tempos de execução totais em segundos, que neste caso incluem as operações do MEF (integração e montagem) e a resolução do sistema linear, são mostrados na Tab. 7.5.

Tabela 7.5 – Tempos de execução totais (s) - Estudo de Caso 1

Discretizações	Seq. - Esperso	Seq. - CSR	CUSP	CUBLAS + CUSPARSE
MS1	31.058,54	32,41	175,77	12,20
MS2	142.475,39	92,54	294,62	35,82

As entradas das matrizes foram armazenadas no formato CSR para todos os testes.

Ao analisar a Tab. 7.5 pode-se concluir que é imprescindível o tratamento da esparsidade das matrizes, como por exemplo ao comparar a primeira e a segunda coluna. O tempo sequencial da primeira coluna, para a malha MS1, foi aproximadamente 9 horas, enquanto que para MS2 foi de 40 horas. O tempo diminui bastante quando um dos métodos de armazenamento das matrizes é aplicado (CSR), como por exemplo para MS1 que teve uma diminuição de quase 1000 vezes (ao comparar a primeira e segunda coluna).

Já para as implementações paralelas, as bibliotecas CUBLAS e CUSPARSE apresentaram melhor desempenho que a biblioteca CUSP. Essa diferença nos tempos pode ser visualizada na Fig. 7.1.

Figura 7.1 – Tempos de execução totais (s) das implementações do método CG (Seq. - CSR, CUSP e CUBLAS + CUSPARSE) - Estudo de Caso 1

Ao analisar essa figura pode-se concluir que o tempo paralelo da quarta implementação é quase três vezes menor que aqueles obtidos no segundo teste, Seq. - CSR. Em termos de tempo computacional, o quarto método foi o melhor (CUBLAS + CUSPARSE).

7.4 Estudo de Caso 2

Neste estudo de caso foi utilizada a configuração C1A e as máquinas H1 e H2 (CAMARGOS, 2013b). Três níveis de discretização, MS1, MS2 e MS3, foram utilizados em quatro implementações. Os dois primeiros níveis são iguais ao Estudo de Caso 1, enquanto que o terceiro contém 1.009.092 elementos não nulos da matriz. A primeira implementação representa o tempo computacional para o algoritmo sequencial, no formato CSR. Os outros testes foram executados na GPU, utilizando as bibliotecas CUBLAS e CUSPARSE.

Os tempos de execução totais em segundos (MEF + resolução do sistema linear) estão disponíveis na Tab. 7.6.

Tabela 7.6 – Tempos de execução totais (s) - Estudo de Caso 2

Discretizações	Seq. - CG	CG	BiCG	CR
MS1	18,56	7,27	7,17	5,96
MS2	52,27	19,15	19,14	17,35
MS3	85,21	37,98	37,74	36,03

Os tempos de execução de todos os métodos e os *speedups* para o método CG podem ser visualizados nas Fig. 7.2 e Fig. 7.3, respectivamente.

Figura 7.2 – Tempos de execução totais (s) de vários métodos - Estudo de Caso2

Em todos os casos, o tempo de simulação do método CR foi menor comparado aos outros métodos de resolução do sistema linear. O BiCG apresentou tempo menor que o CG. Isso foi possível porque não foi necessário calcular a transposta da matriz, já que o sistema é simétrico. É importante notar que o BiCG sem transposta requer somente um produto matriz-vetor, o que permite melhor performance computacional.

Pode-se visualizar na Fig. 7.3 que o melhor *speedup* foi obtido para a segunda malha e seu valor foi próximo a 2,72, considerando a primeira configuração de máquina (H1). Entretanto, em todos os casos obteve-se êxito ao utilizar CUDA e GPU. O tempo computacional somente do método iterativo, por exemplo para obter o CG, representa 33,94% para MS1, 15,69% para a MS2 e 8,47% para a MS3. Pode-se concluir que ao aumentar a ordem das matrizes, o tempo computacional para as etapas do MEF aumenta também. Nas três configurações de malha, o tempo gasto no método iterativo, por exemplo para o CG, variou de 2,47 a 3,22 s, uma variação relativamente pequena.

Figura 7.3 – *Speedup* do método CG - Estudo de Caso 2

A Fig. 7.4 ilustra a diferença de tempos entre MS1 e MS2 ao utilizar as máquinas H1 e H2. Conclui-se que as configurações tanto em CPU quanto da placa gráfica influenciam muito no tempo computacional das simulações. Em todas as implementações realizadas com esses métodos o número de iterações foi próximo de 6.000, entretanto o tempo de execução foi muito rápido devido ao processamento paralelo realizado na GPU.

Outras implementações foram realizadas utilizando a biblioteca CUSP, como ilustrado na Fig. 7.5, também sem utilizar o pré-condicionador. Três testes foram realizados para demonstrar o tempo de processamento para três tipos diferentes de armazenamento das matrizes: formatos CSR, COO e HYB. A variação do tempo de execução entre os três formatos foi muito pequena ao comparar, por exemplo, a malha MS3. Esse foi um dos motivos para a escolha definitiva do formato CSR. Outro fator importante para essa opção é que as funções das bibliotecas CUBLAS e CUSPARSE trabalham com esse formato, o que elimina a necessidade de conversão de formatos.

Figura 7.4 – Comparação entre os tempos de execução totais (s) em dois hardwares diferentes (H1 e H2) - Estudo de Caso 2

Figura 7.5 – Tempos de execução totais (s) para três formatos de armazenamentos das matrizes - Estudo de Caso 2

7.5 Estudo de Caso 3

Neste estudo de caso foi utilizada a configuração C1A e a máquina H1 (CAMARGOS, 2013c). Dois níveis de discretização, MS1 e MS2 (iguais do Estudo de Caso 1), foram utilizados em quatro implementações. Os dois primeiros testes correspondem aos tempos computacionais sequencial e paralelo do algoritmo CG, respectivamente, enquanto que os outros

dois se referem às aplicações com os pré-condicionadores IC e ILU. Todas as implementações paralelas utilizaram as bibliotecas CUBLAS e CUSPARSE.

Os tempos de execução totais em segundos (MEF + resolução do sistema linear) estão disponíveis na Tab. 7.7.

Tabela 7.7 – Tempos de execução totais (s) - Estudo de Caso 3

Discretizações	Seq. CG	CG	ILU	IC
MS1	32,41	12,20	10,88	10,79
MS2	92,54	35,82	35,78	30,67

Ao analisar a Tab. 7.7 pode-se concluir que em todas as implementações paralelas o tempo computacional foi menor comparado ao tempo sequencial (Seq. CG). Nestes casos, o tempo contabilizado para solução do sistema inclui também os pré-condicionadores + PCG. Para os métodos ILU e IC, o tempo de execução foi menor que o tempo para calcular o CG e o algoritmo sequencial. Nesses casos, houve uma redução no tempo de processamento, mesmo com outras operações incluídas no método iterativo, como por exemplo, a matriz pré-condicionada M e as substituições progressivas e regressivas. A Fig. 7.6 ilustra o *speedup* das aplicações.

Figura 7.6 – *Speedup* dos métodos CG e PCG - Estudo de Caso 3

Ao analisar a Fig. 7.6 vê-se que o melhor *speedup* foi obtido para o método IC + PCG. Tanto para a malha MS1 quanto para a MS2, o *speedup* foi acima de 3. O número de iterações diminuiu bastante comparado com aqueles obtidos nos Estudos de Caso 1 e 2, para a mesma configuração de malha, devido ao uso eficiente dos pré-condicionadores. Neste caso, o número de iterações foi igual a 111 para MS1 e 105 para MS2. Para outra configuração de máquina, por exemplo, a H2, o tempo de processamento diminuiu ainda mais.

7.6 Estudo de Caso 4

Neste estudo de caso foram utilizadas as configurações C1A, C2A e C2B, que estão ilustradas na Fig. 7.7 (CAMARGOS, 2014a). Todas as implementações paralelas utilizaram as bibliotecas CUBLAS e CUSPARSE.

Figura 7.7 – Fluxogramas das operações para os casos C1A, C2A e C2B - Estudo de Caso 4

Para o caso C1A utilizou-se o método iterativo PCG e os dois pré-condicionadores IC e ILU. Neste caso, tanto os pré-condicionadores quanto o método PCG foram executados na GPU. Nos casos C2A e C2B também utilizou-se os pré-condicionadores IC e ILU, no entanto, eles foram executados na CPU e o PCG foi implementado na GPU. A matriz pré-condicionada, M , foi obtida com a adição do *shift*, introduzido no Capítulo 5. Além disso, o método RCM foi aplicado em algumas implementações, antes da solução do PCG.

Caso C1A

Os tempos de execução totais em segundos, da resolução do sistema linear, estão disponíveis na Tab. 7.8. A configuração de máquina utilizada nestes testes foi a H2 e os níveis de discretização são iguais ao utilizados nos Estudos de Caso 1 e 2.

Tabela 7.8 – Tempos de execução totais (s) do caso C1A - Estudo de Caso 4

Discretizações	Sequencial	CG	ILU	IC
MS1	5,34	2,47	1,10	1,15
MS2	10,71	3,01	1,25	1,25
MS3	14,45	3,22	1,59	1,43

Pode-se notar uma grande diminuição em relação à Tab. 7.7. Isso ocorreu devido ao tempo contabilizado ser somente do sistema linear e devido à melhor configuração da máquina. No entanto, ao comparar somente o tempo de resolução do sistema iterativo, vê-se que para a malha MS1 o tempo foi igual a 22,44 s, contra 5,34 s da Tab. 7.8. O tempo de processamento da malha MS2, na H1, foi igual a 56,01 s, contra 10,71 s na máquina H2. No primeiro teste de MS1, houve uma redução de 4,2 vezes, enquanto que para MS2, de 5,23 vezes. O número de iterações foi o mesmo. O gráfico do *speedup* relacionado aos tempos da Tab. 7.8 pode ser visualizado na Fig. 7.8, para a máquina H2.

Figura 7.8 – *Speedup* do caso C1A dos métodos CG e PCG - Estudo de Caso 4

Pode-se notar que para as malhas MS2 e MS3 o *speedup* foi maior com o método IC + PCG, de valor máximo 10,12. As Fig. 7.9 e Fig. 7.10 ilustram o percentual das principais operações realizadas nos métodos iterativos: IC + PCG e ILU + PCG, respectivamente.

Figura 7.9 – Percentual das principais operações do IC + PCG - Estudo de Caso 4

Figura 7.10 – Percentual das principais operações do ILU + PCG - Estudo de Caso 4

Ao observar as Figs. 7.9 e Fig. 7.10 pode-se concluir que o maior tempo computacional foi nas operações de substituições progressivas e regressivas, para obter o CG pré-condicionado. Essas operações somadas resultam em 78% e 83%, respectivamente, para os métodos IC + PCG e ILU + PCG. O menor tempo computacional foi obtido no cálculo do produto matriz-vetor. A partir dessas figuras, conclui-se que o IC e o ILU não são os métodos mais adequados à paralelização, pois envolvem operações intrinsecamente sequenciais, no caso, o *solver* triangular (maior tempo de execução). Entretanto, são pré-condicionadores muito robustos e de uso muito difundido e, mesmo com as operações de substituições progressivas e regressivas, os *speedups* observados são consideráveis.

Casos C2A e C2B

Para o regime harmônico, seis testes foram realizados e são detalhados na Tab. 7.9, que inclui o número de elementos não nulos da matriz, *nnzA*. A configuração de máquina utilizada foi a H2.

Tabela 7.9 – Diferentes discretizações e truncamentos para os casos C2A e C2B - Estudo de caso 4

Configuração	Ordem do Elemento	PML	nnzA
C2B - i	1º	—	2.292.543
C2B - ii	2º	—	22.228.109
C2B - iii	1º	X	2.292.543
C2A - i	1º	—	2.821.983
C2A - ii	2º	—	15.654.177
C2A - iii	1º	X	2.821.983

Os pré-condicionadores foram executados sequencialmente na CPU, enquanto que o PCG, na CPU e GPU. Os métodos implementados estão disponíveis na Tab. 7.10 e os tempos computacionais referentes à resolução do sistema linear estão na Tab. 7.11.

Tabela 7.10 – Descrição dos métodos implementados dos casos C2A e C2B - Estudo de Caso 4

Experimentos	RCM	SIC	SILU	PCG
MTH1	—	X	—	X
MTH2	X	X	—	X
MTH3	X	—	X	X

Tabela 7.11 – Tempos de execução totais (s) dos casos C2A e C2B - Estudo de Caso 4

Métodos	C2B			C2A		
MTH1	i	ii	iii	i	ii	iii
nnzU	1.170.159	11.207.250	1.170.159	1.440.045	7.894.125	1.440.045
CPU	430,02	19.934,01	797,26	1.870,71	31.015,70	2.885,37
iter.	229	1.126	427	811	2.524	1.255
GPU	3,65	61,18	6,78	18,98	136,54	28,78
iter.	229	1.122	425	826	2.523	1.257
MTH2	i	ii	iii	i	ii	iii
nnzU	1.170.159	11.207.250	1.170.159	1.440.045	7.894.125	1.440.045
CPU	289,47	5.534,60	585,19	878,64	8.018,48	1.536,38
iter.	156	312	315	386	652	681
GPU	3,53	35,24	7,12	13,45	94,18	24,01
iter.	157	313	315	384	654	684
MTH3	i	ii	iii	i	ii	iii
nnzLU	1.644.166	9.959.778	1.528.120	1.834.010	6.393.946	1.691.262
GPU	4,38	30,25	7,2	12,69	75,95	21,09
iter.	129	315	247	318	738	534

A variável *nnzU*, da Tab. 7.11, representa o número de elementos não nulos da matriz triangular superior da matriz *A*, e *nnzLU* o número de elementos não nulos da matriz triangular superior e inferior. A variável *iter* refere-se ao número de iterações, enquanto que as variáveis CPU e GPU indicam o tempo de processamento do sistema linear, em segundos.

Analisando essa tabela fica evidente a redução ao utilizar GPU comparado com o algoritmo sequencial. Por exemplo, para o C2B - ii (matriz com 22 milhões de elementos não nulos), o tempo total de processamento na CPU foi 19.934,01 s (mais de 5 horas), enquanto que o tempo na GPU foi de apenas 61,18 s, com o método MTH1. Isso significa que o algoritmo paralelo é 325 vezes mais rápido que o algoritmo sequencial. Para o C2A - ii, o tempo total de processamento na CPU foi de 31.015,70 s (mais de 9 horas), contra 136,54 s na GPU.

Para o método MTH2, que utiliza a técnica RCM, o tempo de processamento, na maioria dos casos, foi menor que os obtidos no MTH1. Por exemplo, para o C2B - ii, o tempo na CPU foi de 19.934,01 s para MTH1, e 5.534,60 s para MTH2. Isso representa uma redução de 3,6%. Já na GPU, o tempo foi de 61,18 s contra 35,24 s, uma redução de 1,74%. Isso demonstra a grande importância de utilizar essa técnica, pois reduz o número de iterações e, consequentemente, o tempo de processamento. Os *speedups* referentes à Tab. 7.11 podem ser ilustrados na Fig. 7.11.

Ao analisar a Fig. (7.11) infere-se que o *speedup* foi maior para o MTH1, devido ao elevado número de iterações na solução do PCG. Ao aplicar a técnica RCM, esse número diminui, entretanto, o *speedup* continua elevado para o MTH2. Por exemplo, para C2A - ii com MTH1, o tempo na CPU foi de 31.015,70 s e na GPU de 136,54 s, enquanto que com MTH2, o tempo na CPU foi de 8.018,48 s e na GPU de 94,18 s. O algoritmo sequencial teve uma redução de

Figura 7.11 – Speedup do PCG (MTH1 e MTH2) dos casos C2A e C2B - Estudo de Caso 4

3,87 vezes, enquanto que o paralelo foi de 1,45 vezes.

O algoritmo SIC sequencial foi aquele da biblioteca LMAGLIB, enquanto que o SILU foi do software MATLAB (GILBERT; MOLER; SCHREIBER, 1992). Neste último caso, a função utilizada foi a $ilu(k)$ com uma tolerância de eliminação de elementos não nulos, $k = 10e^{-2}$.

O tempo de execução em segundos dos pré-condicionadores na CPU (MTH1 ao MTH3), dos métodos SIC + PCG (MTH1 e MTH2) e a porcentagem do tempo dos pré-condicionadores sob o tempo total de processamento podem ser encontrados na Tab. 7.12.

Tabela 7.12 – Principais operações do PCG dos casos C2A e C2B - Estudo de Caso 4

Métodos		C2B			C2A		
MTH1		i	ii	iii	i	ii	iii
SIC		95,48	1.845,33	95,83	119,99	1.268,98	119,19
PCG		430,02	19.934,01	797,26	1.870,71	31.015,70	2.885,37
SIC + PCG		525,50	21779,34	893,09	1990,70	32284,68	3004,56
Porcentagem		18%	8%	11%	6%	4%	4%
MTH2		i	ii	iii	i	ii	iii
SIC		108,57	2.283,58	106,47	139,54	1.505,10	139,59
PCG		289,47	5.534,60	585,19	878,64	8.018,48	1.536,38
SIC + PCG		398,04	7818,18	691,66	1018,18	9523,58	1675,97
Porcentagem		27%	29%	15%	14%	16%	8%
MTH3		i	ii	iii	i	ii	iii
SILU		35,33	549,37	35,17	51,82	279,31	52,01

De acordo com a Tab. 7.12, o maior tempo consumido para obter a matriz pré-condicionada foi para o C2B - ii, com o MTH2. Esse tempo representa 29% do tempo total de processamento do SIC + PCG. Pode-se notar pelas Tab. 7.11 e Tab. 7.12 que, na primeira, o uso da técnica RCM reduziu bastante o tempo de processamento do PCG. Entretanto, isso não aconteceu para o cálculo da matriz pré-condicionada, provavelmente porque o algoritmo dos pré-

condicionadores, disponível na LMAGLIB, não está otimizado para trabalhar com essa técnica RCM. Além disso, conclui-se que os melhores tempos de processamento foram obtidos para o SILU. A função *ichol* do MATLAB (decomposição incompleta de Cholesky) falhou em todas as matrizes do caso harmônico, por isso optou-se por continuar utilizando os algoritmos da biblioteca LMAGLIB, para o SIC.

A Tab. 7.13 contém as principais operações para resolver os métodos iterativos, com as respectivas porcentagens em relação ao tempo total de processamento.

Tabela 7.13 – Porcentagem das operações na GPU do caso C2B - ii - Estudo de Caso 4

Operação (MTH1)	Porcentagem do Tempo Total
Converter formato COO para CSR	0,05%
Alocar memória, declarar e transferir as matrizes da CPU para GPU	0,45%
Resolução do sistema linear (PCG)	99,50%
Operação (MTH2)	Porcentagem do Tempo Total
Converter formato COO para CSR	0,08%
Alocar memória, declarar e transferir as matrizes da CPU para GPU	1,04%
Resolução do sistema linear (PCG)	98,88%
Operação (MTH3)	Porcentagem do Tempo Total
Converter formato COO para CSR	0,09%
Alocar memória, declarar e transferir as matrizes da CPU para GPU	0,99%
Resolução do sistema linear (PCG)	98,91%

Pode-se concluir por essa tabela que os tempos para converter o formato COO para o CSR e para alocar e transferir os dados da CPU para GPU são insignificantes comparados aos tempos de processamento do sistema linear. Em todos os casos, o tempo para resolver o sistema linear é o mais expressivo, com valores acima de 98%.

7.7 Estudo de Caso 5

Neste estudo de caso foi utilizada a configuração C2A, com e sem PML (CAMARGOS, 2014b). Seis níveis de discretização foram usados, conforme indicado na Tab. 7.14.

Tabela 7.14 – Níveis de discretização das malhas (MS1 a MS5 com elementos de 1ª ordem e MS6 com elementos de 2ª ordem) - Estudo de Caso 5

Elementos não nulos	MS1	MS2	MS3	MS4	MS5	MS6
nnzA	1.117.770	2.080.786	3.015.732	4.077.671	5.773.607	15.654.177

As malhas MS1 até a MS5 foram geradas com elementos de 1ª ordem, e a MS6, de 2ª ordem. A configuração de máquina foi a H2. As combinações de métodos testadas estão indicadas na Tab. 7.15.

Tabela 7.15 – Descrição dos métodos implementados - Estudo de Caso 5

Experimentos	PML	RCM	SIC	PCG	PBiCG
MTH1	—	—	X	X	X
MTH2	X	—	X	X	X
MTH3	—	X	X	X	X
MTH4	X	X	X	X	X

Em todos os casos, a diagonal foi compensada com o *shift* e a decomposição incompleta de Cholesky foi obtida. Os pré-condicionadores foram executados na CPU, enquanto que o PCG e o PBiCG foram implementados na CPU e GPU.

Os tempos computacionais referentes à resolução do sistema linear estão disponíveis nas Tabs. 7.16 e 7.17. A primeira refere-se aos resultados dos métodos MTH1 e MTH2, e a segunda dos métodos MTH3 e MTH4.

Tabela 7.16 – Tempos de execução totais (s) dos métodos MTH1 e MTH2 - Estudo de Caso 5

Métodos		Discretização Crescente				
MTH1 - PCG	MS1	MS2	MS3	MS4	MS5	MS6
nnzU	571,157	1.062,281	1.538,866	2.079,971	2.944,066	7.894,125
CPU	393,34	782,98	1.143,79	2.636,07	4.954,67	31.015,70
iter.	427	458	469	788	1.047	2.524
GPU	5,30	8,86	10,28	21,45	30,79	136,54
iter.	424	446	443	792	1.034	2.523
MTH1 - PBiCG	MS1	MS2	MS3	MS4	MS5	MS6
CPU	260,53	2.135,09	3.159,7	5.788,43	—	—
iter.	696	632	660	887	—	—
GPU	9,19	15,91	18,31	39,33	57,42	264,21
iter.	424	446	443	792	1.034	2.523
MTH2 - PCG	MS1	MS2	MS3	MS4	MS5	MS6
CPU	573,39	1.063,50	1.647,47	3.996,18	8.425,15	58.660,20
iter.	631	630	673	1.205	1.805	4.764
GPU	7,59	12,41	15,70	32,01	54,63	260,28
iter.	612	627	674	1.182	1.822	4.823
MTH2 - PBiCG	MS1	MS2	MS3	MS4	MS5	MS6
CPU	2.043,51	3.370,08	4.780,97	11.184,7	—	—
iter.	1.146	1.024	995	1.753	—	—
GPU	13,15	22,24	27,70	58,54	101,01	502,29
iter.	612	627	674	1.182	1.822	4.823

Essa tabela contém os elementos não nulos da matriz triangular superior (*nnzU*), o tempo de processamento do PCG (CPU e GPU) e do PBiCG (CPU e GPU), além do número de iterações. Ao analisar essa tabela pode-se concluir que:

1. Para o MTH1, o tempo de processamento na CPU para o PBiCG foi mais que o dobro em todas as discretizações de malha, comparado ao PCG, exceto para MS1. Entretanto, isso não aconteceu na GPU, pois o tempo não chegou a ser o dobro.
2. Para o MTH1, o tempo na GPU foi menor que o sequencial. Por exemplo, no PCG (MS5) o tempo na GPU foi aproximadamente 165 vezes mais rápido que na CPU, enquanto que para MS6 foi de 228 vezes.

3. O algoritmo PBiCG teve sua execução interrompida com as malhas MS5 e MS6, devido à não otimização do algoritmo e configuração de memória da CPU.
4. Quando se insere o PML, o condicionamento da matriz se degrada e o tempo de execução aumenta; o seu maior impacto é para as maiores malhas. Por exemplo, no PCG (MTH1) o tempo da MS1 foi de 394,34 s, e para MTH2, foi de 573,39 s. Entretanto, para a malha MS6 o tempo foi de 31.015,70 s contra 58.660,20 s. Para esse último caso, a diferença entre os métodos foi de aproximadamente 8 horas.
5. O tempo de processamento na CPU para o PBiCG (MTH2), comparado ao PCG, foi mais que o dobro em todas as discretizações, chegando ao triplo nas malhas MS1 e MS2, o que não se observou na GPU.

Os *speedups* referentes aos valores da Tab. 7.16 podem ser visualizado na Fig. 7.12.

Figura 7.12 – *Speedup* do PCG (MTH1 e MTH2) - Estudo de Caso 5

Ao analisar a Fig. 7.12 conclui-se que o *speedup* aumentou com o aumento da discretização. Neste caso, o maior impacto foi para MS6, com *speedup* próximo a 230.

Ao observar horizontalmente essa figura e a Tab. 7.16 (MTH1) pode-se inferir que o tempo na CPU, para MS6, aumentou 79 vezes em relação à MS1, enquanto que na GPU foi somente 26 vezes. Para MTH2, o tempo na CPU aumentou cerca de 103 vezes em relação à MS1, e na GPU, 35 vezes.

O *speedup* para o método PBiCG pode ser encontrado na Fig. 7.13.

Figura 7.13 – Speedup do PBiCG (MTH1 e MTH2) - Estudo de Caso 5

Ao analisar a Fig. 7.13 conclui-se que o *speedup* foi de ao menos 100 em todos os testes.

A Tab. 7.17 contém os resultados dos métodos MTH3 e MTH4.

Tabela 7.17 – Tempos de execução totais (s) dos métodos MTH3 e MTH4 - Estudo de Caso 5

Métodos		Discretização Crescente					
MTH3 - PCG		MS1	MS2	MS3	MS4	MS5	MS6
nnzU		571.157	1.062.281	1.538.866	2.079.971	2.944.066	7.894.125
CPU		196,84	412,30	588,23	1.374,40	2.552,41	8.018,48
iter.		215	243	239	414	500	652
GPU		4,36	9,61	11,88	20,17	34,31	94,18
iter.		217	245	247	416	509	654
MTH3 - PBiCG		MS1	MS2	MS3	MS4	MS5	MS6
GPU		8,37	18,54	22,83	38,60	65,81	184,07
iter.		217	245	247	416	509	654
MTH4 - PCG		MS1	MS2	MS3	MS4	MS5	MS6
CPU		308,83	613,41	931,19	2.018,35	4.270,80	16.465,30
iter.		341	365	380	617	877	1.356
GPU		6,78	14,24	18,17	29,75	58,59	192,76
iter.		341	368	382	618	874	1.341
MTH4 - PBiCG		MS1	MS2	MS3	MS4	MS5	MS6
GPU		13,05	27,68	35,15	57,20	112,96	377,15
iter.		341	368	382	618	874	1.341

Ao analisar as Tabs. 7.17 e 7.16 pode-se concluir que a técnica RCM é bastante eficaz. Por exemplo, ao comparar o MTH1 e o MTH3 para o PCG (CPU), o tempo de processamento diminuiu 3 vezes ao utilizar a técnica RCM, exceto na malha MS6, que diminuiu quase 4 vezes. Isso já não aconteceu na GPU, o tempo em todos os casos foram parecidos, exceto para a malha MS6. Além disso, o número de iterações foi bem menor para todos os testes dos métodos MTH3 e MTH4, comparados aos métodos MTH1 e MTH2. Na presença do PML, o tempo

computacional foi o dobro, mostrando o impacto das propriedades físicas no condicionamento da matriz.

Os *speedups* referentes aos valores contidos na Tab. 7.17, do método PCG, podem ser visualizados na Fig. 7.14.

Figura 7.14 – *Speedup* do PCG (MTH3 e MTH4) - Estudo de Caso 5

Neste caso, como os valores dos tempos computacionais para os métodos MTH3 e MTH4 foram menores, os *speedups* também foram. Entretanto, ainda assim o *speedup* foi significativo, chegando a valores próximos de 90.

A tabela a seguir contém a porcentagem das principais operações realizadas no PCG, para todos os métodos.

Tabela 7.18 – Porcentagens (%) em relação ao tempo total das principais operações do PCG (MS6) - Estudo de Caso 5

Métodos	Substituição Progressiva	Substituição Regressiva	SpMV	Outras
MTH1	45,15	44,77	6,99	3,08
MTH2	45,06	44,78	7,07	3,10
MTH3	49,79	45,91	2,60	1,70
MTH4	49,93	46,07	2,61	1,39

Ao observar a Tab. 7.18 conclui-se que o maior tempo de processamento, para o PCG (MS6), é nas substituições progressiva e regressiva, que pode ser melhor visualizado na Fig. 7.15. Para MTH2 o tempo de execução foi de 117,7 e 117,0 s para as substituições progressiva e regressiva, respectivamente. O tempo do produto matriz-vetor foi de 18,5 s, enquanto que as outras operações totalizaram 8,1 s. Além disso, o tempo para transferir e alocar a memória da GPU representou cerca de 1% do tempo total. Os 98% restantes foram para resolver o sistema linear.

Figura 7.15 – Principais operações do PCG (MS6) em C2A e C2B - Estudo de Caso 5

O histórico da evolução da norma residual do PCG para a malha MS6 está ilustrado na Fig. 7.16.

Figura 7.16 – Histórico da evolução da norma residual do PCG para os vários métodos (MTH1 a MTH4) - Estudo de Caso 5

Ao analisar essa figura percebe-se que o processo de convergência não é estável. De fato, os métodos baseados no processo de Lanczos são inicialmente instáveis comparados aos outros. Entretanto, no final do processo iterativo a norma residual desce rapidamente. Especial-

mente para essa classe de problema, o PBiCG foi mais estável e robusto. A convergência para MTH3 e MTH4 foi mais rápida que para MTH1 e MTH2, indicando que a técnica RCM é altamente recomendada.

Alguns testes foram implementados com as seguintes bibliotecas: *Matrix Template Library* (MTL) (LUMSDANIE; SIEK, 1998) e *Iterative Template Library* (ITL) (LUMSDANIE; SIEK, 1998). A implementação do PCG sequencial da biblioteca ITL, para esse caso especial de matriz, falhou na convergência em todos os casos testados.

7.8 Estudo de Caso 6

Neste estudo de caso foram utilizadas as configurações C2A e C2B e a máquina H2 (CA-MARGOS, 2014c). As malhas utilizadas para mensurar o desempenho computacional estão ilustradas na Tab. 7.19.

Tabela 7.19 – Níveis de discretização das malhas (MS1 a 5 com elementos de 1ª ordem e MS6, 2ª ordem) - Estudo de Caso 6

Elementos não nulos	MS1	MS2	MS3	MS4	MS5	MS6
nnzA	1.117.770	2.080.786	3.015.732	4.077.671	5.773.607	15.654.177

As características de cada método implementado podem ser encontradas na Tab. 7.20.

Tabela 7.20 – Descrição dos métodos implementados - Estudo de Caso 6

Experimentos	RCM	SIC	SILU	PCG	PBiCG	PBiCGStab
MTH1	X	X	—	X	—	—
MTH2	—	X	—	—	X	—
MTH3	X	X	—	—	X	—
MTH4	X	—	X	X	—	—
MTH5	X	—	X	—	X	—
MTH6	X	—	X	—	—	X

Ao todo, dois testes foram realizados na CPU e os outros foram realizados na GPU.

Os tempos computacionais referentes à resolução do sistema linear estão disponíveis na Tab. 7.21. Essa tabela contém o tempo de processamento do PCG (CPU e GPU), do PBiCG (CPU e GPU) e do PBiCGStab (GPU). Para ambos CPU e GPU, o desempenho computacional do PCG foi superior ao do PBiCG, chegando em alguns casos três vezes mais rápido, devido às operações extras (matriz transposta e substituição progressiva e regressiva) que o algoritmo PBiCG necessita para convergir. Adicionalmente, a técnica RCM é extremamente relevante, tanto nas aplicações envolvendo a CPU, quanto a GPU.

A Fig. 7.17 ilustra os tempos computacionais dos métodos implementados.

Tabela 7.21 – Tempos de execução totais (s) - Estudo de Caso 6

Métodos		Discretização Crescente				
MTH1	MS1	MS2	MS3	MS4	MS5	MS6
CPU	308,83	613,41	931,19	2.018,35	4.270,80	16.465,30
iter.	341	365	380	617	877	1.356
GPU	6,78	14,24	18,17	29,75	58,59	192,76
iter.	341	368	382	618	874	1.341
MTH2	MS1	MS2	MS3	MS4	MS5	MS6
CPU	2.043,51	3.370,08	4.780,97	11.184,7	—	—
iter.	1.146	1.024	995	1.753	—	—
GPU	13,15	22,24	27,70	58,54	101,01	502,29
iter.	612	627	674	1.182	1.822	4.823
MTH3	MS1	MS2	MS3	MS4	MS5	MS6
GPU	13,05	27,68	35,15	57,20	112,96	377,15
Iter.	341	368	382	618	874	1.341
MTH4	MS1	MS2	MS3	MS4	MS5	MS6
GPU	5,62	10,70	13,15	27,65	45,48	133,45
Iter.	287	297	307	491	640	1.315
MTH5	MS1	MS2	MS3	MS4	MS5	MS6
GPU	11,06	21,21	25,96	55,23	89,98	264,90
Iter.	287	297	307	491	640	1.315
MTH6	MS1	MS2	MS3	MS4	MS5	MS6
GPU	33,20	49,00	70,90	220,13	503,55	1.474,84
Iter.	860	690	839	1.984	3.589	7.270

Ao analisar os tempos contidos na Tab. 7.21 e na Fig. 7.17 pode-se inferir que o maior impacto foi observado para a malha MS6, em MTH1 e MTH2. Para os outros métodos, o melhor desempenho foi para MTH4 (SILU + PCG), seguidos pelos métodos MTH1, MTH5, MTH3 e MTH6. O pior tempo de execução foi para o SILU + PBiCGStab. O número de iteração, dos casos que utilizaram o SILU, foi bem menor comparado ao SIC, exceto para o PBiCGStab. Entretanto, os dois pré-condicionadores são muito eficientes para esse caso especial de matriz.

Figura 7.17 – Tempos de execução totais (s) do caso C2A - Estudo de Caso 6

7.9 Estudo de Caso 7

Neste estudo de caso foram utilizadas as configurações C1A e C1B, bem como C2A e C2B (CAMARGOS, 2014d). As implementações utilizaram as bibliotecas CUSP, CUBLAS e CUSPARSE. As máquinas utilizadas foram a H2 e H3.

Casos C1A e C1B

Três níveis de discretização foram adotados e o número de elementos não nulos na matriz são iguais aos definidos nos Estudos de Caso 1 e 2 (MS1, MS2 e MS3).

Os testes realizados estão descritos na Tab. 7.22. Os métodos MTH1, MTH3, MTH7 e MTH8 já foram introduzidos anteriormente, no entanto, agora, eles serão apresentados em outra configuração de máquina (H3).

Tabela 7.22 – Descrição dos métodos MTH1 até MTH8 do caso C1A - Estudo de Caso 7

Métodos	Seq.	OpenMP	CG	PCG	AMG	DIA	AINV	IC	ILU
MTH1	X	—	—	—	—	—	—	—	—
MTH2	—	X	—	—	—	—	—	—	—
MTH3	—	—	X	—	—	—	—	—	—
MTH4	—	—	—	X	X	—	—	—	—
MTH5	—	—	—	X	—	X	—	—	—
MTH6	—	—	—	X	—	—	X	—	—
MTH7	—	—	—	X	—	—	—	X	—
MTH8	—	—	—	X	—	—	—	—	X

Outros pré-condicionadores também foram implementados, além do IC e ILU, tais como: *Multigrid* Algébrico, Diagonal e Inversa Aproximada. A biblioteca utilizada nesses três últimos casos foi a CUSP. O método MTH1 corresponde a versão sequencial, enquanto que os outros utilizam as técnicas OpenMP e CUDA. Para o caso em OpenMP, a programação paralela foi aplicada no produto matriz-vetor, com quatro núcleos físicos e quatro emulados.

A Tab. 7.23 contém os tempos totais para todos os métodos da Tab. 7.22.

Tabela 7.23 – Tempos de execução totais (s) dos métodos MTH1 até MTH8 - Estudo de Caso 7

Métodos	MS1	MS2	MS3
MTH1	6,85	15,33	22,34
MTH2	4,5	7,84	10,33
MTH3	2,55	2,85	3,65
MTH4	1,46	1,98	2,52
MTH5	2,06	2,32	3,11
MTH6	12,25	37,97	85,37
MTH7	1,44	1,49	1,70
MTH8	1,38	1,46	1,73

A Fig. 7.18 ilustra os tempos de processamento do CG, sem pré-condicionador, para os métodos MTH1 até MTH3.

Figura 7.18 – Tempos de execução totais (s) dos métodos MTH1 até MTH3 - Estudo de Caso 7

Ao analisar a Tab. 7.23 e a Fig. 7.18 conclui-se que a técnica OpenMP reduz bastante o tempo de execução, comparado ao algoritmo sequencial, no entanto a programação CUDA reduz esse tempo ainda mais. Para a última malha a redução foi acima de 6 vezes. Esse tempo pode diminuir ainda mais ao inserir os pré-condicionadores. Por exemplo, ao utilizar a biblioteca CUSP (MTH4, MTH5 e MTH6), o melhor tempo de processamento foi obtido para o método MTH4, com o pré-condicionador AMG. Entretanto, os melhores desempenhos computacionais foram encontrados para os métodos MTH7 e MTH8, que foram pré-condicionados através dos métodos IC e ILU.

O gráfico do *speedup* para os métodos da Tab. 7.23 está disponível na Fig. 7.19. Ao analisar essa figura conclui-se que o *speedup* maior foi obtido para o método MTH7, aproximadamente 13,14 mais rápido que o algoritmo sequencial. O pior caso foi obtido para o método MTH6 (AINV + PCG).

Outros testes foram realizados com os pré-condicionadores AMG, DIA e AINV, no entanto, relacionados à resolução do sistema linear PBiCG e PBiCGStab. Esses testes estão representados na Tab. 7.24 e seus respectivos tempos de processamento disponíveis na Tab. 7.25.

Tabela 7.24 – Descrição dos métodos MTH9 até MTH14 do caso C1A - Estudo de Caso 7

Métodos	PBiCG	PBiCGStab	AMG	DIA	AINV
MTH9	X	—	X	—	—
MTH10	X	—	—	X	—
MTH11	X	—	—	—	X
MTH12	—	X	X	—	—
MTH13	—	X	—	X	—
MTH14	—	X	—	—	X

Figura 7.19 – Speedup dos métodos CG e PCG - Estudo de Caso 7

Tabela 7.25 – Tempos de execução totais (s) dos métodos MTH9 até MTH14 do caso C1A - Estudo de Caso 7

Métodos	MS1	MS2	MS3
MTH9	1,86	2,09	2,65
MTH10	2,23	2,45	3,19
MTH11	12,7	39,17	85,76
MTH12	1,78	2,00	2,58
MTH13	2,88	2,56	3,33
MTH14	14,10	38,89	88,84

Para o PBiCG, o melhor tempo obtido foi para o método MTH9, que utiliza o pré-condicionador AMG. Em seguida, vem o DIA, e logo após o AINV. Para o método PBiCGStab isso aconteceu também. O pior caso foi obtido para o método PBiCGStab + AINV.

Casos C2A e C2B

As características físicas do problema e as discretizações das malhas são as mesmas do artigo anterior, sendo que essa última pode ser visualizada na Tab. 7.19. A máquina utilizada foi a H2.

As configurações de testes implementadas nesse artigo podem ser encontradas na Tab. 7.26.

Tabela 7.26 – Descrição dos métodos implementados dos casos C2A e C2B - Estudo de Caso 7

Experimentos	RCM	SIC	SILU	PCG	PBiCG
MTH15	X	X	—	X	—
MTH16	X	X	—	—	X
MTH17	X	—	X	X	—
MTH18	X	—	X	—	X

A Tab. 7.27 contém resultados dos tempos de processamento para uma GPU e duas GPUs. No último caso, utilizou-se a tecnologia *GPUDirect*, que permite a comunicação entre as GPUs. A Fig. 7.20 ilustra as principais operações realizadas em cada GPU.

Figura 7.20 – Distribuição das operações num sistema *Multi-GPU* dos casos C2A e C2B - Estudo de Caso 7

Essa figura ilustra as operações realizadas na GPU 1 e na GPU 2. O produto matriz-vetor foi realizado na GPU 2, enquanto que as outras operações, na GPU 1. Para que o sistema iterativo realize todos os cálculos, de forma correta, é necessário que a cada iteração a GPU 2 gere uma cópia do vetor para a GPU 1. As matrizes A e a pré-condicionada M foram armazenadas, respectivamente, na GPU 2 e na GPU 1. De um lado cada GPU individual tem seu próprio armazenamento, no entanto, por outro lado, ocorre a cópia dos vetores de uma para outra GPU.

Os tempos computacionais referentes à resolução do sistema linear estão disponíveis na Tab. 7.27. Para cada método da Tab. 7.26, duas implementações foram realizadas, sendo a primeira somente com uma GPU e a segunda com duas GPUs.

Tabela 7.27 – Tempos de execução totais (s) dos casos C2A e C2B - Estudo de Caso 7

Métodos		Discretização Crescente				
MTH15	MS1	MS2	MS3	MS4	MS5	MS6
1 GPU	6,78	14,24	18,17	29,75	58,59	192,76
2 GPU	7,32	14,72	18,56	29,98	58,08	190,34
MTH16	MS1	MS2	MS3	MS4	MS5	MS6
1 GPU	13,05	27,68	35,15	57,20	112,96	377,15
2 GPU	13,88	28,58	36,25	56,90	114,65	379,21
MTH17	MS1	MS2	MS3	MS4	MS5	MS6
1 GPU	5,62	10,70	13,15	27,65	45,48	133,45
2 GPU	6,07	10,59	13,03	24,43	40,07	126,72
MTH18	MS1	MS2	MS3	MS4	MS5	MS6
1 GPU	11,06	21,21	25,96	55,23	89,98	264,90
2 GPU	11,29	20,63	25,18	48,48	79,81	250,74

Conclui-se ao analisar a Tab. 7.27 que o uso de GPU é muito eficaz, no entanto, quando se utiliza duas GPUs o tempo de processamento pode diminuir ainda mais. Por exemplo, para os métodos MTH17 e MTH18, os tempos de duas GPUs são menores quando comparado ao de

uma GPU, exceto a menor malha (MS1). A redução fica ainda mais expressiva quando a ordem do sistema aumenta. Assim, para aplicações de larga escala essa tecnologia tende a ser ainda mais eficaz. Mesmo que em cada iteração seja realizada a cópia do vetor resultante do SpMV, ainda assim existe uma redução no tempo total de processamento.

Esse recurso, porém, não foi tão eficaz ao utilizar o SIC + PCG e SIC + PBiCG, comparado aos tempos de uma CPU, mas para as outras duas, houve ganho computacional. De qualquer forma, as implementações paralelas são muito mais eficientes que aquelas realizadas sequencialmente.

A Fig. 7.21 ilustra os tempos computacionais para os métodos MTH17 e MTH18, tanto para uma GPU quanto para duas GPUs. Conclui-se que vale a pena investigar novas tecnologias, principalmente quando os quesitos memória e volume de dados são críticos em aplicações de ordem elevada.

Figura 7.21 – Tempos de execução totais (s) dos métodos MTH17 e MTH18 - Estudo de Caso 7

7.10 Estudo de Caso 8

Neste estudo de caso foi utilizada a configuração C2A (CAMARGOS, 2014e). Seis níveis de discretização foram implementados tais como disponíveis na Tab. 7.14. As combinações de métodos testadas estão indicadas na Tab. 7.28, sendo que alguns resultados já foram introduzidos no Estudo de Caso 5. No entanto, neste estudo de caso o foco será comparar os dois pré-condicionadores SIC e SILU, bem como introduzir o método PBiCGStab. A configuração de máquina utilizada foi a H2.

Tabela 7.28 – Descrição dos métodos implementados - Estudo de Caso 8

Experimentos	RCM	SIC	SILU	PCG	PBiCG	PBiCGStab
MTH1	X	X	—	X	—	—
MTH2	—	X	—	—	X	—
MTH3	X	X	—	—	X	—
MTH4	X	—	X	X	—	—
MTH5	X	—	X	—	X	—
MTH6	X	—	X	—	—	X

Os pré-condicionadores foram executados na CPU, enquanto que o PCG, o PBiCG e o PBiCGStab foram implementados na CPU e GPU. Os tempos computacionais referentes à resolução do sistema linear estão disponíveis nas Tabs. 7.29.

Tabela 7.29 – Tempos de execução totais (s) dos métodos MTH1 até o MTH6 - Estudo de Caso 8

Métodos		Discretização Crescente				
MTH1	MS1	MS2	MS3	MS4	MS5	MS6
CPU	308,83	613,41	931,19	2.018,35	4.270,80	16.465,30
iter.	341	365	380	617	877	1.356
GPU	6,78	14,24	18,17	29,75	58,59	192,76
iter.	341	368	382	618	874	1.341
MTH2	MS1	MS2	MS3	MS4	MS5	MS6
CPU	2.043,51	3.370,08	4.780,97	11.184,70	—	—
iter.	1.146	1.024	995	1.753	—	—
GPU	13,15	22,24	27,70	58,54	101,01	502,29
iter.	612	627	674	1.182	1.822	4.823
MTH3	MS1	MS2	MS3	MS4	MS5	MS6
GPU	13,05	27,68	35,15	57,20	112,96	377,15
iter.	341	368	382	618	874	1.341
MTH4	MS1	MS2	MS3	MS4	MS5	MS6
GPU	5,62	10,70	13,15	27,65	45,48	133,45
iter.	287	297	307	491	640	1.315
MTH5	MS1	MS2	MS3	MS4	MS5	MS6
GPU	11,06	21,21	25,96	55,23	89,98	264,90
iter.	287	297	307	491	640	1.315
MTH6	MS1	MS2	MS3	MS4	MS5	MS6
GPU	33,20	49,00	70,90	220,13	503,55	1.474,84
Iter.	860	690	839	1.984	3.589	7.270

Essa tabela contém o tempo de processamento do PCG (CPU e GPU), do PBiCG (CPU e GPU) e do PBiCGStab (CPU e GPU), além do número de iterações. Pode-se concluir que o melhor desempenho foi encontrado para o teste MTH4 (SILU + PCG), enquanto que o pior caso foi obtido para o método MTH6 (SILU + PBiCGStab). Ao comparar os métodos MTH3 e MTH5 (mesmo método iterativo) conclui-se que o pré-condicionador SILU foi mais eficiente, pois o tempo de execução foi de 377,15 s contra 264,90 s, para MS6.

Os *speedups* referentes aos métodos MTH1 e MTH2 estão contidos na Fig. 7.22. Ao analisar essa figura conclui-se que o *speedup* aumentou com o aumento da discretização. Para a malha MS4, o *speedup* ficou entre 68 e 191, respectivamente, para MTH1 e MTH2.

Figura 7.22 – *Speedup* do PCG e PBiCGStab (MTH1 e MTH2) - Estudo de Caso 8

7.11 Estudo de Caso 9

Neste estudo de caso foram utilizadas as configurações C1A e C1B, bem como C2A e C2B (CAMARGOS, 2014f). Alguns resultados já foram introduzidos anteriormente, entretanto este trabalho é a versão estendida do Estudo de Caso 7. As implementações utilizaram as bibliotecas CUSP, CUBLAS e CUSPARSE. As máquinas utilizadas foram a H2 e H3. Para os casos C1A e C1B nada foi alterado, entretanto para o caso harmônico alguns resultados foram acrescentados.

Casos C2A e C2B

As configurações de testes implementadas nesse artigo podem ser encontradas na Tab. 7.30 (alteração da Tab. 7.26).

Tabela 7.30 – Descrição dos métodos implementados dos casos C2A e C2B - Estudo de Caso 7

Experimentos	RCM	SIC	SILU	PCG	PBiCG
MTH15	—	X	—	X	—
MTH16	X	X	—	X	—
MTH17	—	X	—	—	X
MTH18	X	X	—	—	X
MTH19	X	—	X	X	—
MTH20	X	—	X	—	X

O pseudo-código das principais operações do PCG realizadas em cada GPU estão ilustradas na Fig. 7.23.

```

Para  $i = 1, 2, \dots$ 
  Calcule  $Mz^{(i-1)} = r^{(i-1)} \rightarrow$  Kernel 1 – GPU 0
   $\rho_{i-1} = r^{(i-1)T} z^{(i-1)} \rightarrow$  Kernel 2 – GPU 0
  if  $i = 1$ 
     $p^{(1)} = z^{(0)}$ 
  else
     $\beta_{i-1} = \rho_{i-1} / \rho_{i-2} \rightarrow$  Kernel 3 – GPU 0
     $p^{(i)} = z^{(i-1)} + \beta_{i-1} p^{(i-1)} \rightarrow$  Kernel 4 – GPU 0
  endif
   $q^{(i)} = Ap^{(i)} \rightarrow$  Kernel 1 – GPU 1
   $\alpha_i = \rho_{i-1} / p^{(i)T} q^{(i)} \rightarrow$  Kernel 5 – GPU 0
   $x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)} \rightarrow$  Kernel 6 – GPU 0
   $r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)} \rightarrow$  Kernel 7 – GPU 0

Fim

```

Figura 7.23 – Distribuição das operações do PCG na *Multi-GPU* dos casos C2A e C2B - Estudo de Caso 9

Os tempos computacionais referentes à resolução do sistema linear estão disponíveis na Tab. 7.31. Para cada método da Tab. 7.30, duas implementações foram realizadas, sendo a primeira somente com uma GPU e a segunda com duas GPUs. Os métodos MTH15 e MTH17 foram implementados sem a técnica RCM.

Tabela 7.31 – Tempos de execução totais (s) dos casos C2A e C2B - Estudo de Caso 9

Métodos		Discretização Crescente				
MTH15	MS1	MS2	MS3	MS4	MS5	MS6
1 GPU	7,59	12,41	15,70	32,01	54,63	260,28
2 GPU	6,70	11,61	15,00	29,72	52,33	250,80
MTH16	MS1	MS2	MS3	MS4	MS5	MS6
1 GPU	6,78	14,24	18,17	29,75	58,59	192,76
2 GPU	7,32	14,72	18,56	29,98	58,08	190,34
MTH17	MS1	MS2	MS3	MS4	MS5	MS6
1 GPU	13,15	22,24	27,70	58,54	101,01	502,29
2 GPU	13,92	20,92	27,46	57,97	100,64	500,67
MTH18	MS1	MS2	MS3	MS4	MS5	MS6
1 GPU	13,05	27,68	35,15	57,20	112,96	377,15
2 GPU	13,88	28,58	36,25	56,90	114,65	379,21
MTH19	MS1	MS2	MS3	MS4	MS5	MS6
1 GPU	5,62	10,70	13,15	27,65	45,48	133,45
2 GPU	6,07	10,59	13,03	24,43	40,07	126,72
MTH20	MS1	MS2	MS3	MS4	MS5	MS6
1 GPU	11,06	21,21	25,96	55,23	89,98	264,90
2 GPU	11,29	20,63	25,18	48,48	79,81	250,74

A Fig. 7.24 ilustra os *speedups* para os métodos MTH15, MTH16 e MTH17. Ao analisar essa figura conclui-se que o pré-condicionador SILU foi mais eficiente que o SIC e houve uma redução significativa ao comparar, por exemplo, MTH18 e MTH20 (cerca de 34% em 2 GPU, para MS6). A redução do tempo de execução poderia ter sido mais expressiva se a função

Figura 7.24 – *Speedup* do PCG e PBiCG (MTH15, MTH16 e MTH17) - Estudo de Caso 9

peer-to-peer memory access estivesse habilitada para a configuração de máquina H2. Isto só é suportado por máquinas de 64 *bits* e GPUs compatíveis com a série Tesla da NVIDIA. Além disso, a cópia dos dados e a comunicação são expressivas nesta tecnologia (*GPUDirect*).

7.12 Estudo de Caso 10

As implementações deste estudo de caso estão disponíveis no Apêndice A deste trabalho (CAMARGOS, 2014g). Esses resultados adicionais foram submetidos para uma revista e ainda se encontravam sob análise no momento da submissão deste manuscrito.

7.13 Outros Resultados

Além de todos os métodos iterativos desenvolvidos, o CGNE, o CGNR e o CGS também foram implementados. No entanto, eles não convergiram com o uso dos pré-condicionadores. Por esse motivo, eles não foram detalhados.

O problema mais desafiador testado neste trabalho foi o de um sistema gerado a partir de uma modelagem no regime harmônico pelo MEF de um sistema grande, cuja ordem atingiu 242.160. O número de elementos não nulos, complexos, dessa matriz foi de 29.199.314. Além de armazenar a própria matriz foi necessário armazenar as matrizes pré-condicionadas: triangular inferior (*nnzL*) e superior (*nnzU*). Cada matriz triangular tem como número de elementos não nulos 5.683.253. O pré-condicionador utilizado foi o SILU, conforme detalhado no item 7.5. O método de solução do sistema linear foi o PCG. Para esse caso específico, o tempo sequencial

foi superior a 5 dias ininterruptos, sendo que o tempo na GPU foi de somente 3 minutos e 23 segundos. O número de iterações foi 1.182. Assim, mesmo utilizando um pré-condicionador sequencial, o ganho foi considerável e merece destaque, mostrando ser essa, uma tecnologia na qual se deve continuar investindo.

7.14 Síntese do Capítulo

Neste capítulo foram apresentados todos os resultados obtidos no decorrer deste trabalho. Vários métodos de solução do sistema linear foram paralelizados utilizando CUDA e executados, seja em plataformas *Single-GPU* ou *Multi-GPU*. Os resultados demonstraram, através dos *speedups* observados, que os ganhos computacionais são apreciáveis, tornando essa técnica de computação paralela uma alternativa muito competitiva, quando comparada às tradicionais, envolvendo *clusters* de CPUs. Além de utilizar CUDA, a técnica OpenMP também foi implementada, entretanto não se mostrou tão eficiente quanto a programação em GPU. Em todos os testes realizados, o PCG se demonstrou o mais rápido, seguido por PBiCG e PBiCGStab. No próximo capítulo serão apresentadas as principais conclusões do trabalho e os direcionamentos para trabalhos futuros.

Conclusões

8.1 Síntese do Trabalho

Com este trabalho buscou-se trazer os benefícios de uma tecnologia nova, acessível e promissora, a computação paralela em GPU, para a resolução de sistemas de equações algébricas provenientes da modelagem de fenômenos eletromagnéticos pelo Método de Elementos Finitos. O objetivo foi o desenvolvimento de um conjunto de ferramentas que alie os benefícios da computação paralela a uma plataforma de custo relativamente baixo, sem a necessidade de montagem de *clusters* e sem a complicação de técnicas de decomposição de domínio.

Nos concentramos naqueles sistemas derivados da modelagem dos fenômenos envolvidos na difusão de correntes em meios condutores abertos com excitação DC e harmônica, que são de interesse para Engenharia Elétrica, como é o caso dos sistemas de aterramento. Esses problemas têm sido objeto de pesquisa do LMAG nos últimos anos, propiciando a geração de um banco de dados de sistemas razoavelmente amplo para a realização dos testes de desempenho. Some-se a isso o fato da resolução dos sistemas de equações resultantes ser árdua, pelas características próprias do problema, descritas ao longo deste manuscrito, e a ausência de trabalhos nessa área por parte de outros autores.

Em particular, a resolução do sistema de equações algébricas complexo oriundo da formulação $A - V$ aresta-nodal do MEF, aplicado à solução da Equação de Onda completa, fasorial e não homogênea, não possui equivalente na literatura pertinente, e sua solução não é trivial. Poucos são os métodos em uso atualmente aptos a resolvê-lo de forma robusta, estável e satisfatória, e que o custo computacional da sua solução seja considerável.

Com relação aos métodos de resolução, dada a esparsidade, além de outras características, das matrizes geradas pelo MEF, os métodos iterativos, em particular aqueles do subespaço de Krylov, são aqueles nos quais este trabalho se concentrou, além de sua adequação ao paralelismo. Os métodos utilizados foram: Gradiente Conjugado, Gradiente Biconjugado, Conjugado Residual, Gradiente Biconjugado Estabilizado, Gradiente Conjugado para equações normais e Gradiente Conjugado ao Quadrado. Além desses métodos, foram testados os seguintes pré-condicionadores: fatorizações incompletas LU e de Cholesky, *Multigrid* Algébrico, Diagonal

e Inversa Aproximada. Todos os algoritmos foram implementados em CUDA e linguagem C/C++, e testados em três máquinas, cada uma delas equipada com um modelo distinto de GPU do tipo GeForce da NVIDIA.

Para os sistemas a coeficientes reais, foi possível, tanto implementar, quanto executar com sucesso na GPU, praticamente todas as combinações de método-iterativo/pré-condicionador. Dos métodos iterativos, o PCG foi o que apresentou a melhor performance, apesar do mais estável ter sido o PBiCG. Em todas as situações foram observados elevados *speedups*, crescentes com a ordem dos sistemas, e que portanto justificam plenamente o investimento nessa tecnologia, graças ao seu custo-benefício.

Para o sistema a variáveis complexas, porém, o uso do pré-condicionador impôs restrições. Até o momento, todas as versões de pré-condicionadores com dados complexos (não apenas aquele particular mencionado acima, como outros, oriundos de outros métodos) falharam em nossos testes. Para contornar esse problema, adotamos uma solução híbrida CPU-GPU, na qual o pré-condicionador é executado de forma serial na CPU e somente o algoritmo iterativo implementado na GPU. Mesmo com essa solução híbrida, foram obtidos *speedups* surpreendentes, que justificam plenamente a adoção dessa abordagem.

Os resultados de *speedups* mencionados acima referem-se ao processamento sequencial, na CPU. Para complementar o estudo, foram introduzidos implementações e testes de desempenho adicionais, a fim de se comparar as alternativas acessíveis e disponíveis em *desktops* pessoais. As configurações adicionais testadas foram OpenMP e Multi-GPU. Esta última representa uma alternativa à baixa capacidade de memória, característica das GPUs. Os resultados obtidos confirmaram a superioridade das implementações em GPU, mesmo comparando-as a implementações paralelizadas em CPU, como é o caso da OpenMP.

Um caso-teste que merece destaque é aquele realizado com um sistema de mais de 242.000 linhas. O tempo de execução na CPU chegou a 5 dias, contra apenas 3 minutos e 23 segundos na GPU, resultado esse que consideramos justificar o esforço despendido até aqui e o investimento futuro nessa tecnologia por parte dos pesquisadores envolvidos com o MEF.

8.2 Principais Dificuldades

No início deste trabalho, muito esforço foi despendido no aprendizado e utilização das bibliotecas CUBLAS E CUSPARSE, pois são ainda recentes, não completamente depuradas. Outra dificuldade que merece destaque é a documentação deficiente, incompleta, e com muitos erros. Assim, quase todo o trabalho de implementação e depuração dos códigos, no início, teve de ser realizado principalmente mediante a consulta a fóruns de desenvolvedores.

Além de todos os métodos iterativos citados, o CGNE, o CGNR e o CGS também foram implementados. Entretanto, eles não convergiram com o uso dos pré-condicionadores. Por esse motivo, eles não receberam mais destaques. Como as matrizes para o caso harmônico são muito mal-condicionadas, vários testes foram realizados, sem sucesso. Os principais são enumerados

a seguir.

1. Utilização no pré-condicionador somente a parte real da matriz com tentativa de solucionar o sistema normalmente com os métodos PCG, PBiCG e PBiCGStab.
2. Implementação de precisão simples no caso harmônico.
3. Utilização de precisão mista (IKUNO et al., 2010).
4. Aplicação do pré-condicionador proposto por Made et al. (2000) e por Made (2001).
5. Implementação do pré-condicionador LDL^T .
6. Utilização das funções IC e ILU das bibliotecas CUBLAS e CUSPARSE na versão complexas, para pré-condicionar as matrizes oriundas do MEF nodal-aresta.
7. Aplicação da biblioteca CUSP para matrizes complexas.
8. Separação da matriz complexa em parte real e imaginária para formar um sistema real não simétrico. Foi utilizado o ILU + PBiCG.
9. Implementação e utilização do ILU em Bloco.
10. Utilização do GMRES.

8.3 Principais Contribuições

As principais contribuições proporcionadas por este trabalho foram as enumeradas a seguir.

1. Trazer os benefícios de uma tecnologia nova, a computação paralela em GPU, para a resolução de sistemas de equações algébricas de larga escala provenientes da modelagem de fenômenos eletromagnéticos pelo MEF em uma plataforma de custo relativamente baixo, sem a necessidade de montagem de *clusters* e uso de técnicas de decomposição de domínio.
2. Propor a adoção de uma solução híbrida CPU-GPU, na qual o pré-condicionador é executado de forma serial na CPU e somente o algoritmo iterativo na GPU, para resolver os sistemas a coeficientes complexos oriundos da formulação $\mathbf{A} - \mathbf{V}$ nodal-aresta do MEF aplicada à Equação de Onda completa não homogênea.
3. Desenvolvimento de uma ferramenta computacional moderna, robusta e capaz de auxiliar a resolução de uma ampla gama de sistemas de equações algébricas com tempos de processamento drasticamente reduzidos e com boa precisão numérica.

8.4 Perspectivas e Trabalhos Futuros

São apontados a seguir os desdobramentos e perspectivas deste trabalho, citando apenas os principais.

1. Pesquisa de um pré-condicionador paralelizável mais robusto para casos de aritmética complexa e para problemas em altas frequências, pois as modelagens no domínio da frequência são de grande interesse na Engenharia Elétrica. A definição do pré-condicionador ótimo ainda é dependente do problema. Exige estudo aprofundado das características da matriz, envolvendo a distribuição dos auto-valores. Uma possibilidade que merece mais investigação é o uso do AMG, ainda que as versões disponíveis em GPU até a conclusão deste trabalho, como a da biblioteca CUSP, não tenham obtido sucesso.
2. Comparação dos desempenhos das versões CUDA com versões em MPI com uso de *clusters*.
3. Comparação das versões CUDA com as versões paralelizadas dos algoritmos de álgebra linear de bibliotecas mais consagradas e consolidadas, como PETSc (BALAY et al., 2013) e Trilinos (GEE et al., 2006), além do *toolkit* de paralelismo do Matlab, que inclui suporte a GPU.
4. Extensão dos testes a problemas não lineares, transitórios, acoplamentos multifísicos, estudos paramétricos e de otimização, etc.
5. Recodificação de todas as etapas do MEF (geração de malha, integração, montagem, pós-processamento, cálculo de grandezas a partir dos resultados) em CUDA para processamento na GPU.
6. Comparação com outras abordagens do MEF paralelizado, como a Element-by-element (KISS et al., 2012).
7. Agregação dessas ferramentas à biblioteca LMAGLIB para serem disponibilizadas a outros estudantes e professores.
8. Aprimoramento das entradas de dados dos algoritmos paralelos relacionados à manipulação das matrizes.
9. Implementação de uma arquitetura híbrida que combine duas ou mais bibliotecas em um único programa, ou seja, MPI com OpenMP ou MPI com CUDA.

Referências Bibliográficas

ALMASI, G. S.; GOTTLIEB, A., **Highly parallel computing**. 2^a ed. Redwood City: Benjamin Cummings, 1994. 689 p.

BALAY, S. *et al.*, **PETSc users manual**, Argonne National Laboratory, USA, Tech. Rep. ANL-95/11, May 2013. Disponível em <<http://www.mcs.anl.gov/petsc/petsc-current/docs/manual.pdf>>. Acesso em: junho de 2014.

BARTOM, M. L.; RATTNER, R. J., Parallel computing and its impact on computational electromagnetics. **IEEE Transactions on Magnetics**, vol. 28, n^o 2, pp. 1690-1695, March 1992.

BARRETT, R.; BERRY, M.; CHAN, T. F.; DEMMEL, J.; DONATO, J. M.; DONGARRA, J.; EIJKHOUT, V.; POZO, R.; ROMINE C.; VORST, H. V., **Templates for the solution of linear systems: building blocks for iterative methods**. 2^a ed. Siam, August 2006. 117p.

BELL, N.; GARLAND, M., **Efficient sparse matrix-vector multiplication on CUDA**, NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, December 2008.

CAMARGOS, A. F. P.; SILVA, V. C., Efficient conjugate gradient Pparallelization on GPU. **The 9th International Symposium on Electric and Magnetic Fields (EMF)**, Bruges, April 2013a.

CAMARGOS, A. F. P.; SILVA, V. C., Iterative Krylov solution methods parallelization on GPU. **GPU Technology Conference (GTC)**, San José, March 2013b.

CAMARGOS, A. F. P.; SILVA, V. C., Efficient preconditioned conjugate gradient parallelization on GPU. **The 19th Conference on the Computation of Electromagnetic Fields (COM-PUMAG)**, Budapest, July 2013c.

CAMARGOS, A. F. P.; SILVA, V. C. ; GUICHON, J.-M ; MEUNIER, G., Efficient parallel preconditioned conjugate gradient solver on GPU for FE modeling of electromagnetic fields

in highly dissipative media. **IEEE Transactions on Magnetism**, vol. 50, n° 2, pp. 569-572, February 2014a.

CAMARGOS, A. F. P.; SILVA, V. C.; GUICHON, J.-M.; MEUNIER, G., Iterative solution on GPU of linear systems arising from the A-V edge-FEA of time-Harmonic electromagnetic phenomena. **22nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)**, Turim, February 2014b.

CAMARGOS, A. F. P.; SILVA, V. C., Krylov-subspace methods on GPU for FEA of time-harmonic electromagnetic problems. **19th International Conference on Computation in Electromagnetics (CEM)**, Londres, April 2014c.

CAMARGOS, A. F. P.; SILVA, V. C., Krylov subspace solvers using multi-GPU computing in FEA of electromagnetic phenomena. **16th Biennial IEEE Conference on Electromagnetic Field Computation (CEFC)** Annecy, May 2014d.

CAMARGOS, A. F. P.; SILVA, V. C., GPU computing for FEA of time-harmonic electromagnetic problems. **16º SBMO - Simpósio Brasileiro de Micro-ondas e Optoeletrônica e 11º CBMag - Congresso Brasileiro de Eletromagnetismo (MOMAG)**, Brasil, Agosto 2014e.

CAMARGOS, A. F. P.; SILVA, V. C., Performance analysis of Multi-GPU implementations of Krylov-subspace methods applied to FEA of electromagnetic phenomena. Artigo submetido em 08/06/2014 ao **IEEE Transactions on Magnetism**, 2014f.

CUBLAS, NVIDIA Corporation. CUDA basic linear algebra subroutines, Disponível em <<https://developer.nvidia.com/cublas>>. Acesso em: abril 2014.

CUSP, NVIDIA Corporation. Generic parallel algorithms for sparse matrix and graph computations, Disponível em <<https://developer.nvidia.com/cusp>>. Acesso em: abril 2014.

CUSPARSE, NVIDIA Corporation. Sparse matrix library, Disponível em <<https://developer.nvidia.com/cuSPARSE>>. Acesso em: abril 2014.

DEHNAVI, M. M.; FERNÁNDEZ, D. M.; GIANNACOPOULOS, D., Finite element sparse matrix vector multiplication on graphic processing units. **IEEE Transactions on Magnetism**, vol. 46, n° 8, pp. 2982-2985, August 2010.

DEHNAVI, M. M.; FERNÁNDEZ, D. M.; GIANNACOPOULOS, D., Enhancing the performance of conjugate gradient solvers on graphic processing units, **IEEE Transactions on Magnetism**, vol. 47, n° 5, pp. 1162-1165, May 2011.

DZIEKONSKI, A.; LAMECKI, A.; MROZOWSKI, M., GPU acceleration of multilevel solvers for analysis of microwave components with finite element method. **IEEE Microwave and Wireless Components Letters**, vol. 21, n° 1, January 2011.

FERNANDEZ, D. M.; GIANNACOPOULOS, D.; GROSS, W. J., Multicore acceleration of CG algorithms using blocked pipeline matching techniques. **IEEE Transactions on Magnetics**, vol. 46, n° 8, pp. 3057-3060, August 2010.

FERNANDEZ, D. M.; DEHNAVI, M. M.; GROSS, W. J.; GIANNACOPOULOS, D., Alternate parallel processing approach for FEM. **IEEE Transactions on Magnetics**, vol. 48, n° 82, pp. 399-402, February 2012.

FLETCHER, R., **Conjugate gradient methods for indefinite systems**. Lecture Notes in Mathematics, 2^a ed., vol. 506, Berlin, Springer, 1976, pp. 73-89.

FOSTER, I., **Designing and building parallel programs - concepts and tools for parallel software engineering**. Addison-Wesley Publishing Company, Inc, 1995. 381 p.

FUJIWARA, K.; NAKATA, T.; FUSAYASU, H., Acceleration of convergence characteristic of the ICCG method, **IEEE Transactions on Magnetics**, vol. 29, n° 2, pp. 1958-1961, March 1993.

GEE, M.; SIEFERT C.; HU, J.; TUMINARO, R.; SALA, M., **ML 5.0 smoothed aggregation user's guide**, Sandia National Laboratories, Albuquerque, USA, Tech. Rep. SAND2006-2649, 2006. Disponível em <<http://trilinos.sandia.gov/packages/ml/mlguide5.pdf>>. Acesso em: junho de 2014.

GEORGE, A.; LIU, J., **Computer solution of large sparse positive definite systems**, Prentice-Hall, 1981.

GILBERT, J. R.; MOLER, C.; SCHREIBER, R., Sparse matrices in MATLAB: design and implementation, **SIAM Journal on Matrix Analysis**, 1992.

GEUZAIN, C.; REMACLE, J-F., Gmsh Copyright - **Gmsh: A Three-dimensional finite element mesh generator with built-in pre-and post-processing facilities**. Version 1.65, 2014. <http://geuz.org/gmsh/>.

GODEL, N.; SCHOMANN, S.; WARBURTON, T.; CLEMENS, M., GPU accelerated adams bashforth multirate discontinuous galerkin FEM dimulation of high frequency electromagnetic fields. **IEEE Transactions on Magnetics**, vol. 46, n° 8, pp. 2735-2738, August 2010.

GODEL, N.; NUNN, N.; WARBURTON, T.; CLEMENS, M., Scalability of higher order discontinuous galerkin FEM computations for solving electromagnetic wave propagation problems on GPU clusters. **IEEE Transactions on Magnetics**, vol. 46, n° 8, pp. 3469-3472, August 2010.

GOLUB, G. H.; VAN LOAN, C. F., Matrix computations. 2^a ed., London, The Johns Hopkins Press Ltd., 1989. 642 p.

- HAYT Jr, W. H., **Eletromagnetismo**. 6^o ed., Rio de Janeiro, LTC Editora, 1999. 339 p.
- HESTENES, M. R.; STIEFEL, E., Methods of conjugate gradients for solving linear systems. **Journal of Research of the National Bureau of Standards**, vol. 49, n^o 6, pp. 409-436, December 1952.
- IDA, N.; WANG, J. S., Parallel implementation of field solution algorithms. **IEEE Transactions on Magnetics**, vol. 24, n^o 1, pp. 291-294, January 1988.
- IKUNO, S.; FUJITA, N.; YAMAMOTO, S.; NAKATA, S., Implementation of variable preconditioned GCR with mixed precision on GPU using CUDA. **14th Biennial IEEE Conference on Electromagnetic Field Computation**, May 2010, Chicago.
- IKUNO, S.; KAWAGUCHI, Y.; FUJITA, N.; ITOH, T.; NAKATA, S.; WATANABE, K., Iterative solver for linear system obtained by edge element: variable preconditioned method with mixed precision on GPU, **IEEE Transactions on Magnetics**, vol. 48, n^o 2, pp. 467-470, February 2012.
- JIN, J., **The finite element method in electromagnetics**. 2^a ed., Wiley e Sons, 2002. 753 p.
- KAKAY, A.; WESTPHAL, E.; HERTEL, R., Speedup of FEM micromagnetic simulations with graphical processing units. **IEEE Transactions on Magnetics**, vol. 46, n^o 6, pp. 2303-2306, June 2010.
- KIRK, D.; HWU W., **Programming massively parallel processors: a hands on approach**. 1^a ed., Morgan Kaufmann, ELSEVIER. 2010. 350 p.
- KISS, I.; GYIMÓTHY, S.; BADICS, Z., PÁVÓ, J., Parallel Realization of the Element-by-Element FEM Technique by CUDA. **IEEE Transactions on Magnetics**, vol. 48, n^o 2, pp. 507-510, February 2012.
- KOMATITSCH, D.; MICHEAA, D.; ERLEBACHER, G., Porting a high order finite element earthquake modeling application to NVIDIA graphics cards using CUDA. **Elsevier: Journal Parallel Distributed Computing**, vol. 69, pp. 451-460, June 2009.
- LI, S.; LIVSHITZ, B.; LOMAKIN, V., Graphics processing unit accelerated O(N) micromagnetic solver. **IEEE Transactions on Magnetics**, vol. 46, n^o 6, pp. 2373-2375, June 2010.
- LI, R., SAAD, Y., GPU-accelerated preconditioned iterative linear solvers. **The Journal of Supercomputing**, vol. 63, n^o 2, pp. 443-466, February 2013.
- LUMSDANIE, A.; SIEK, J., *MTL: The matrix template library 2*, Disponível em <<http://osl.iu.edu/research/mtl/mtl2.php3>>. Acesso em: abril de 2014.

LUMSDANIE, A.; SIEK, J., **ITL: The iterative template library**, Disponível em <<http://www.osl.iu.edu/research/itl>>. Acesso em: abril de 2014.

MADE, M. M. M., Incomplete factorization-based preconditionings for solving the Helmholtz equation, *International Journal for Numerical Methods in Engineering*, vol. 50, pp.1077-1101, January 2001.

MADE, M. M. M., BEAUWENS, R.; WARZEE G., Preconditioning of discrete helmholtz operators perturbed by a diagonal complex matrix, *Communications in Numerical Methods in Engineering*, vol. 16, n° 11, pp. 801-817, October 2000.

MARTINHO, L. B., **Contribuição à modelagem de sistemas de aterramento pelo método dos elementos finitos no domínio harmônico**. 2009. 93 p. Dissertação - Escola Politécnica, Universidade de São Paulo, São Paulo. 2009.

NVIDIA, NVIDIA Corporation. Disponível em <<http://www.nvidia.com/>> Acesso em: abril de 2014.

OKIMURA, T.; SASAYAMA, T.; TAKAHASHI, N.; IKUNO, S., Parallelization of finite element analysis of nonlinear magnetic fields using GPU, *IEEE Transactions on Magnetics*, vol. 49, n° 5, pp. 1557-1560, May 2013.

OWENS, B. J.; HOUSTON, M.; LUEBKE, D.; GREEN, S.; STONE, J. E.; PHILLIPS, J. C., GPU computing. **Proceedings IEEE**. vol. 96, n° 5, pp. 879-899, May 2008.

PATTERSON, D. A.; HENNESSY, J. L., **Arquiteturas de computadores: uma abordagem quantitativa**. 5ª ed. Rio de Janeiro: Elsevier, 2014. 744 p.

PARHAMI, B., **Introduction to parallel processing: algorithms and architectures**. Kluwer Academic Publishers, 2002. 577 p.

REN, D. Q.; GIANNACOPOULOS, D.; SUDA, R., Power performance analysis of 3D finite element mesh refinement with tetrahedra by CUDA/MPI on multicore and GPU architecture. **14th Biennial IEEE Conference on Electromagnetic Field Computation**, 2010, Chicago.

RICHTER, C.; SCÖPS, S.; CLEMENS, M., GPU Acceleration of algebraic multigrid preconditioners for discrete elliptic field problems, **IEEE Transactions on Magnetics**, vol. 50, n° 2, February 2014.

RODRIGUES, A. W. O.; GUYOMARC'H, F.; DEKEYSER, J.-L.; MENACH, Y. L., Automatic Multi-GPU code generation applied to simulation of electrical machines, **IEEE Transactions on Magnetics**, vol. 48, n° 2, pp. 831-834, February 2012.

RODRIGUES, A. W. O.; CHEVALLIER, L.; MENACH, Y. L.; GUYOMARC'H, F., Test harness on a preconditioned conjugate gradient solver on GPUs: an efficiency analysis, **IEEE**

Transactions on Magnetics, vol. 49, nº 5, pp. 1729-1732, May 2013.

SAAD, Y., **Iterative methods for sparse linear systems**. 2^a ed. New York: PWS Publishing, 2003.

SAAD, Y.; SCHULTZ, M. H., GMRES: a generalized minimal residual algorithm for solving non-symmetric linear systems. **SIAM Journal on Scientific and Statistical Computing**, vol. 7, nº 3, pp. 856-869, July 1986.

SADIKU, M. N. O, **Numerical techniques in electromagnetics**. CRC Press LLC, 2000. 742 p.

SANDERS, J.; KANDROT, E., **CUDA by example: an introduction to general-purpose GPU programming**. Addison-Wesley, 2011. 290 p.

SILVA, V. C., **Método de elementos finitos aplicado à solução de problemas de aterramento elétrico**. 2006. 78 p. Tese (Livre Docência) - Escola Politécnica, Universidade de São Paulo, São Paulo, 2006.

SILVA, V. C.; CARDOSO, J. R.; NABETA, S. I.; PALIN, M. F.; PEREIRA, F. H., Determination of frequency-dependent characteristics of substation grounding systems by vector finite-element analysis, **IEEE Transactions on Magnetics**, vol. 43, nº 4, pp. 1825-1828, April 2007.

SILVA, V. C.; MARTINHO, L. B.; CARDOSO, J. R.; PEREIRA FILHO, M. L., VERARDI, S. L. L., Efficient modeling of thin wires in a lossy medium by finite elements applied to grounding systems, **IEEE Transactions on Magnetics**, vol. 47, nº 5, pp. 966-969, May 2011.

SHEWCHUK, J. R., **An introduction to the conjugate gradient method without the agonizing pain**. Carnegie Mellon University, Pittsburgh, PA, 1994.

VAN DER SLUIS, A.; VAN DER VORST, H. A., The rate of convergence of conjugate gradients. **Numerische Mathematik** (Springer-Verlag New York), vol. 48, nº 5, pp. 543-560, 1986.

SONNEVELD, P., CGS, A fast lanczos-type solver for non-symmetric linear systems. **SIAM Journal on Scientific and Statistical Computing**, vol. 10, nº 1, pp. 36-52, 1989.

STEINBERG, P. *et al.*, Journeyman's programming Tour. **ACM Parallel Computing Tech Pack**, November 2009.

TANEMBAUM, A. S., **Sistemas operacionais modernos**. 3^a ed. São Paulo, Pearson Prentice Hall, 2010. 638 p.

VAN DER VORST, H. A., Bi-CGSTAB: a fast and smoothly converging variant of Bi-CG for

the solution of non-symmetric linear systems. **SIAM Journal on Scientific and Statistical Computing**, vol. 13, n° 2, pp. 631-644, March 1992.

VAN DER VORST, H. A., **Lecture notes on parallel iterative solution methods for linear systems arising from discretized PDE's**. Institute Mathematical, University of Utrecht. 1995.

XU, X.; LIU, G.; QU, H.; XU, W.; ZHANG, Y., Study on GPU-accelerated extraction of interconnects parasitic using CUDA and MPI. **14th Biennial IEEE Conference on Electromagnetic Field Computation**, 2010, Chicago.

Apêndice A - Artigo Submetido e em Análise

Nas páginas a seguir encontra-se o artigo, fruto deste trabalho, recentemente submetido e em análise no momento da finalização deste documento:

CAMARGOS, A. F. P.; SILVA, V. C., GPU-accelerated iterative solution of complex-entry systems issued from 3D Edge-FEA of electromagnetics in the frequency domain. Artigo submetido em 12/06/2014 ao **International Journal of High Performance Computing Applications**, 2014g.

GPU-accelerated Iterative Solution of Complex-entry Systems Issued from 3D Edge-FEA of Electromagnetics in the Frequency Domain

Ana F. P. Camargos^{1,2}

¹Instituto Federal de Minas Gerais
Formiga, Brasil
anaflavia@ifmg.edu.br

Jean-M. Guichon

Grenoble Génie Electrique Laboratoire, CNRS
Saint Martin d'Hères, France
jean-michel.guichon@g2elab.grenoble-inp.fr

Viviane C. Silva²

²Escola Politécnica da Universidade de São Paulo
São Paulo, Brasil
vivianecs@usp.br

Gérard Meunier

Grenoble Génie Electrique Laboratoire, CNRS
Saint Martin d'Hères, France
Gerard.Meunier@g2elab.grenoble-inp.fr

Abstract—We present a performance analysis of a parallel implementation for both preconditioned Conjugate Gradient and preconditioned Bi-conjugate Gradient solvers running on Graphic Processing Units with CUDA programming model. The solvers were mainly optimized for the solution of sparse systems of algebraic equations at complex entries, arising from the three-dimensional Edge Finite Element Analysis of the electromagnetic phenomena involved in the open-bound earth diffusion of currents under time-harmonic excitation. We used a shifted Incomplete Cholesky factorization as preconditioner. Results show a significant speedup by using either a Single-GPU or a Multi-GPU device, compared to a serial CPU implementation, thereby allowing the simulations of large-scale problems in low-cost personal computers. Additional experiments of the optimized solvers show that its use can be successfully extended to other complex systems of equations arising in Electrical Engineering, as those obtained in Power-System analysis.

Keywords—Finite Elements; Graphic Processing Unit; Preconditioner; Incomplete Factorization, Edge Elements, Electromagnetics.

I. INTRODUCTION

The Finite Element Analysis (FEA) has proven to be one of the most powerful, robust and versatile tools to solve boundary value problems governed by partial differential equations. It is able to run on any computer architecture, and large scale problems can be solved easily nowadays, as computational platforms equipped with parallel architectures are becoming more affordable.

The greatest challenge for those dealing with large scale systems originated from FEA is to provide techniques for solving linear systems that combine scalability, portability, fine-grained parallelism and flexibility across the assortment of parallel platforms and programming models.

Graphic Processing Units (GPU) are examples of such technologies that have drawn much attention in recent years for scientific computing, although originally conceived for

computer games. It provides access to massively parallel computing capacity, while reducing energy consumption [1], [2] at relatively low cost.

However, the utilization of GPUs in an everyday practice faced two obstacles. First, its use requires a deep adaptation of implemented applications and algorithms to a target architecture, so as to match its internal features. Thus, moving numerical procedures from one platform to another becomes an important task. This drawback has been circumvented satisfactorily with the advent of NVIDIA CUDA (Compute Unified Device Architecture), a standard C language extension for parallel application development on NVIDIA GPUs [3]. CUDA application consists of SPMD (single program, multiple data) computations (kernels) performed by threads running in parallel on the GPU streaming multiprocessors. Thus, CUDA is designed for general-purpose computing. It facilitates heterogeneous computing, CPU + GPU (combined use of both CPU and GPU in the same algorithm), such as illustrated in Fig. 1 [2], [3].

Fig. 1. Data processing flow in a CPU + GPU computing process.

Another bottleneck of the GPUs is their low memory capacity. Indeed, in many current desktop PCs, the memory on the CPU motherboard is often four to eight times larger than the amount installed on the graphics card. In this case, Multi-GPU devices became a viable solution for many real-life applications. A technology called NVIDIA GPUDirect can provide a good solution for communication issues. It allows communication between GPUs on the same shared memory server and allows the use of high-speed direct memory access (DMA) transfers to copy data between the memories of two GPUs on the same system/PCIe bus. The data can be communicated directly, and it can reduce the communication time using peer-to-peer memory access and peer-to-peer memory copy between GPUs [5].

Thenceforth, GPUs have been used successfully for solving large-scale sparse linear systems of algebraic equations arising in the FEA of 3D Electromagnetic phenomena [3], especially in parallel implementations of Krylov subspace iterative methods, the class of methods that are the most widespread for solving systems generated by partial differential equations (PDE) [14].

The sequential execution times are usually large for 3D boundary value problems, since they yield matrices with high order, many of them exhibiting bad conditioning. This leads to a large number of iterations when one uses Krylov iterative methods for solving linear systems. Both Conjugate Gradient (CG) and Bi-conjugate Gradient (BiCG) methods are the more commonly used. However, in these algorithms many iterations are required to reach a certain specified accuracy. A technique to accelerate the convergence is preconditioning the matrix, whose aim is to improve its condition number [4]. The Incomplete Cholesky factorization technique (IC) is often used as preconditioner along with the CG and BiCG methods in FEA of electromagnetic phenomena [4], [23].

This paper deals with the problem of performing a GPU adaptation for both the preconditioned Conjugate Gradient (PCG) and preconditioned Bi-conjugate Gradient (PBiCG) methods efficiently. Our aim is to solve large-scale systems of complex entries which arise from the 3D edge-finite element modeling of the problem. We performed two test cases. **Case 1**, solves the problem of 3D underground diffusion of harmonic currents, as presented in [6], [7]. For the sake of completeness, and to evaluate both robustness and applicability of the presented GPU-based tool in other electrical engineering-related systems, we include additional numerical experiments we have performed on systems arising from the application nodal analysis (Kirchhoff to an electric circuit of a power delivery network of a real-world metropolitan area, named **Case 2**.

We demonstrate its improved performance over the serially optimized C++ code. Furthermore, comparisons of the performance of CPU/Single-GPU, CPU/Multi-GPU and Single-GPU/Multi-GPU versions were analyzed as a function of the grid size simulation.

II. STATE OF THE ART

The CG method was introduced by Hestenes et al. [8] and has a long and successful history in computational mathematics [9], [10]. In [4] several iterative methods for the

solution of linear systems, preconditioners and parallel implementation with both shared and distributed memory are reported.

As mentioned previously, preconditioning is a necessary step for faster convergence of Krylov subspace iterative methods, which can be accomplished using algorithms such as IC. This method decomposes the coefficient matrix into two triangular matrices, which can then be solved by forward and backward substitution, tasks that represent the dominant computational cost of this method and are strictly sequential [3], [4], [11].

Due to the emergence of CUDA, the use of GPUs for sparse linear algebra and the solution of sparse linear systems [12] has become more accessible, thanks to GPU-accelerated libraries such as CUSPARSE and CUSP [5]. There are several works which attempt to optimize the PCG method [12], [13], [14]. In the first one, the authors analyze the performance of the GPU using the Incomplete Cholesky CG method (ICCG) and incomplete LU (ILU) preconditioned GMRES method. The second one presents an ICCG implementation optimized on both multi-core and GPU architectures by using domain decomposition. In [14], the authors implemented a variable preconditioned Krylov subspace method with mixed precision on GPU for solving the linear system obtained from the edge finite element formulation in steady state. The execution times of various VP Krylov subspace methods using Single-GPU and Multi-GPU were analyzed.

Other authors [15], [16] presented some PCG implementations for GPU using CUDA, with Jacobi, SSOR and Approximate Inverse Preconditioner.

Implementations of both ILU and IC preconditioned iterative methods in GPU are presented in [11]. The author used them with both Bi-conjugate Gradient Stabilized and CG, respectively. The performance achieved in this case was a speedup of 2 at most.

Although Krylov-subspace methods are not still of widespread usage in the Power Systems community, as is for the PDE-simulation researchers, their use has gained increasing attention, as well as the emerging GPU computing [26][27].

Most of the aforementioned works on FEA deal mainly with linear systems at real coefficients. None of them, to the best of our knowledge, have been tested to solve the system of complex entries arising from the ungauged edge-AV formulation in the frequency domain. The preconditioner is a key point in such class of problems, but the GPU-CUDA implementations of all available preconditioners have failed in our tests. The hybrid CPU-GPU solution strategy we propose here has been, up to now, the only one which has been able to solve this class of problem efficiently in the GPU.

III. PROBLEM DEFINITION

A. Case 1

An eddy-current problem arises if a body made of conducting material is excited by a time varying magnetic field. The excitation can be in the form of an impressed current, when the total current through a conductor is prescribed, but the current density distribution therein is

unknown. This is the case of a skin effect problem with a prescribed current [6], such as in Fig. 2.

The domain includes a set of grounding electrodes as well as the semi-infinite conducting medium, the soil, which will be truncated with the aid of perfectly matched layers (PML), as described in [6].

Fig. 2. Finite element domain for modeling the diffusion of underground currents.

The phenomenon is governed by Maxwell equations, namely:

$$\nabla \times \mathbf{H} = \mathbf{J} \quad (1)$$

$$\nabla \cdot \mathbf{B} = 0 \quad (2)$$

$$\nabla \times \mathbf{E} = -j\omega \mathbf{B} \quad (3)$$

$$\nabla \cdot \mathbf{J} = 0 \quad (4)$$

where, $\mathbf{H}$ is the magnetic field intensity, $\mathbf{B}$ is the magnetic flux density, $\mathbf{E}$ is the electric field intensity, and $\mathbf{J}$ is the total current density, which includes both conduction and displacement currents.

After applying the constitutive equations, the fields quantities are derived from the potentials (magnetic vector potential $\mathbf{A}$ and electric scalar potential V) as follows:

$$\mathbf{B} = \nabla \times \mathbf{A} \quad (5)$$

$$\mathbf{E} = -j\omega(\mathbf{A} + \nabla V) \quad (6)$$

where, V is defined so that:

$$\mathbf{V} = j\omega V \quad (7)$$

Substitution of (5) and (6) leads to the following differential equation:

$$\nabla \times (\nabla \times \mathbf{A}) + j\omega \sigma (\mathbf{A} + \nabla V) = 0 \quad \text{in } \Omega \quad (8)$$

$$\nabla \cdot [-j\omega \sigma (\mathbf{A} + \nabla V)] = 0 \quad \text{in } \Omega \quad (9)$$

After imposing the boundary conditions, the vector potential can be approximated by vector basis function as:

$$\mathbf{A} \cong \sum_{k=1}^{N_e} \varpi_k \mathbf{a}_k \quad (10)$$

where, $\mathbf{a}_k$ is the linear integral of $\mathbf{A}$ along edges k , ϖ_k is the vector basis function of edge k , and N_e is the number of edges [6]. The scalar potential V is expanded by nodal basis function as:

$$V \cong \sum_{k=1}^{N_n} N_k v_k \quad (11)$$

where, N_k is the nodal shape function of node k , v_k is the nodal values of V and N_n is the number of nodes [6].

Substitution of (10) and (11) into (8) and (9) leads to the Galerkin residual equations:

$$\begin{aligned} \int_{\Omega} \nabla \times \varpi_l \cdot \nabla \times \mathbf{A} d\Omega + \int_{\Omega} j\omega \sigma \varpi_l \cdot (\mathbf{A} + \nabla V) d\Omega \\ + \frac{j\omega}{Z_r} \int_{\Gamma_e} \varpi_l \cdot (\mathbf{A} + \nabla V) d\Gamma = 0 \end{aligned} \quad (12)$$

$$\begin{aligned} \int_{\Omega} j\omega \sigma \nabla N_l \cdot (\mathbf{A} + \nabla V) d\Omega + \frac{j\omega}{Z_r} \int_{\Gamma_e} \nabla N_l \cdot (\mathbf{A} + \nabla V) d\Gamma \\ = \int_{\Gamma_e} N_l \mathbf{J} \cdot \mathbf{n} d\Gamma = I \end{aligned} \quad (13)$$

This yields the following sparse symmetric system of algebraic equations:

$$\begin{pmatrix} \mathbf{M} & \mathbf{P} \\ \mathbf{P}^T & \mathbf{Q} \end{pmatrix} \begin{pmatrix} \mathbf{a} \\ v \end{pmatrix} = \begin{pmatrix} 0 \\ I \end{pmatrix} \quad (14)$$

where, I stands for the net current imposed to the grounding grid at the feed point.

The case involves the harmonic behavior, which is governed by a 3D current-driven wave equation and is discretized by the ungauged $\mathbf{A}$ - V edge-element formulation [6], [7]. In this particular case, the resulting system of algebraic equations of complex unknowns is very ill-conditioned, especially for excitation currents of high frequencies (some tens of MHz). This is due to several reasons, including high aspect ratios in: (a) dimensions of the domain components; (b) physical properties; and (c) use of anisotropic absorbers (PML) for truncating the domain. A detailed analysis of this issue can be found in [22].

As a result, it is nearly impossible to avoid a substantial amount of low-quality elements in the mesh, thereby leading to slow convergence of the iterative solution. In large scale problems, as in [7], with about 2 millions unknowns, the computational cost of the solution becomes prohibitive and the ability of the implemented methods to explore the features of the parallel architecture of the GPU will be key points to consider, as our aim is to be able to employ a standalone, low-cost platform, without using domain decomposition.

B. Case 2

The data we used here are derived from the application of the nodal analysis to power-delivery networks, the largest one belonging to a real-world metropolitan network. The system to be solved in this situation is as follows:

$$[Y][v] = [i] \quad (15)$$

where, Y is the matrix of nodal admittances (sparse symmetric) of the network, v is the vector of nodal voltages and i , the vector of nodal currents.

IV. METHODOLOGY

A. Parallel Processing and Flowchart of the Performed Tests

Fig. 1 represents the data processing flow in a CPU + GPU computing process. The mesh generation, as well as the assembly of the sparse global systems are executed in the host (CPU), whereas the solution of the sparse linear systems is performed in the device (GPU). In this case, the GPU is allocated and the data are copied from CPU memory to GPU memory and n *Kernels* are invoked to calculate each iteration of the algorithm illustrated in Fig. 6 and Fig. 7.

Then, the results are copied back from GPU to CPU memory and the performance of the algorithm is analyzed. According to this model, each *Kernel* is executed as a batch of threads organized as a grid of thread blocks, whose configuration is defined by the programmer [5].

By using the Multi-GPU card, the solution of the sparse linear systems is performed in two devices with the GPUDirect technology as illustrated in Fig. 3.

Fig. 3. Schematic view of the flux of communication in the NVIDIA GPUDirect technology [5].

In this case, the GPU 0 runs the following operations: back and forward substitution, inner products and so on, whereas GPU 1 executes the sparse matrix-vector product (see Fig. 4). The matrices are distributed according to the operations of each GPU and the vector obtained by GPU 1 is sent to GPU 0.

Fig.4. Operations performed by each device in a Multi-GPU architecture.

For the solution of sparse linear systems we have performed tests in two situations: without and with renumbering. For the latter, the method used was the Reverse Cuthill-McKee Ordering (RCM) applied to reduce the matrix bandwidth [17]. In addition, a diagonal compensation is performed by applying a diagonal shift factor, in order to accomplish the IC factorization, which would fail otherwise for the particular physical problem we are modeling. For the iterative solution step, we adopted a residual tolerance of 10^{-6} . The preconditioning was executed sequentially in the host, whereas either PCG or PBICG were run either in the CPU or GPU. For the serial CPU routine of the IC-BiCG method we used a C++ implementation of the following linear algebra libraries: Matrix Template Library (MTL) and Iterative Template Library (ITL) [18], [19]. In the case of ICCG we used our own C++ sequential code.

B. Reverse Cuthill-McKee Reordering

The Reverse Cuthill-McKee ordering is an effective, inexpensive profile-limiting permutation. It is frequently used to reduce the matrix bandwidth. The bandwidth of a row of the matrix is essentially the number of matrix entries between the first and last nonzero entries in the row, with the provision that the diagonal entry is always treated as though it were non-zero [17]. Fig. 5 illustrates the effect of RCM reordering on the matrix sparsity pattern of system (14).

Fig. 5. Sparsity pattern of system (14) for Case 1 (a) without RCM; (b) with RCM.

C. Linear System and Preconditioner

Equation (14) is a symmetric, sparse system of linear equations of complex entries, which can be written as:

$$Kx = b \quad (16)$$

By applying a preconditioning matrix H [4] the new system to be solved becomes:

$$H^{-1}Kx = H^{-1}b \quad (17)$$

Up to now, and according to our experience, the most efficient preconditioners for the system issued from the A - V edge-element formulation applied to our physical problem have been the shifted Incomplete Cholesky decomposition (SIC) [20], [21], [22]. In this case, H is obtained as:

$$H = LL^T \quad (18)$$

where, L is a lower triangular matrix with the same sparsity pattern as K .

The SIC factorization is not guaranteed to succeed for any arbitrary symmetric positive definite matrix. All the conditions that guarantee the existence of the factorization rely on some form of diagonal dominance. One idea suggested in [4] to minimize this difficulty was to use a diagonal compensation in H by means of a shift factor, α , such as:

$$H + \alpha I \quad (19)$$

An attempt to determine the optimal shift value for a given problem is proposed in [20]. In our experiments the SIC was implemented as in [22] with a shift factor of 0.1.

Although the number of iterations to achieve convergence is still very high, this method is the one that keep this value at the least possible. Block preconditioners [10], which are more amenable to parallelism, and hence better adapted to the GPU architecture, failed in our problems, as they yielded a worse-conditioned matrix, even after applying reordering techniques.

Since incomplete factorizations are essentially sequential algorithms in nature, we have kept the preconditioning step running serially on the CPU. Nevertheless, as the time spent for preconditioning represents only a negligible amount of the overall solving time for the system considered, the impact of this approach will not affect the results significantly, as will be shown, since the most time-consuming step is the iterative process. This study has not received such enhancements, and we aim to demonstrate that improved solver performance can still be accomplished.

The PCG and PBiCG algorithms [23], running in two GPUs, are given in Fig. 6 and Fig. 7, respectively.

```

Compute  $r^{(0)} = b - Ax^{(0)}$  for some initial guess  $x^{(0)}$ 
for  $i = 1, 2, \dots$ 
  solve  $Mz^{(i-1)} = r^{(i-1)} \rightarrow \text{Kernel 1 - GPU 0}$ 
   $\rho_{i-1} = r^{(i-1)T} z^{(i-1)} \rightarrow \text{Kernel 2 - GPU 0}$ 
  if  $i = 1$ 
     $p^{(1)} = z^{(0)}$ 
  else
     $\beta_{i-1} = \rho_{i-1} / \rho_{i-2} \rightarrow \text{Kernel 3 - GPU 0}$ 
     $p^{(i)} = z^{(i-1)} + \beta_{i-1} p^{(i-1)} \rightarrow \text{Kernel 4 - GPU 0}$ 
  endif
   $q^{(i)} = Ap^{(i)} \rightarrow \text{Kernel 1 - GPU 1}$ 
   $\alpha_i = \rho_{i-1} / p^{(i)T} q^{(i)} \rightarrow \text{Kernel 5 - GPU 0}$ 
   $x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)} \rightarrow \text{Kernel 6 - GPU 0}$ 
   $r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)} \rightarrow \text{Kernel 7 - GPU 0}$ 
  check convergence; continue if necessary
end

```

Fig. 6. Pseudo-code of PCG algorithm.

Therefore, to implement this PCG algorithm, four main operations are required, namely: (i) SpMV; (ii) back and forward substitution; (iii) inner products; and (iv) vector updates. All these operations are executed in the GPU, where each row corresponds to a specific *Kernel* and where the operation is being performed (GPU 0 or GPU 1), as shown in Fig. 6. In this case, the preconditioner must be applied in each iteration. To compute z_0 , both forward and back substitutions must be performed. Thus, this step requires two more operations when compared to the pure CG algorithm. However, the number of iterations is drastically reduced.

```

Compute  $r^{(0)} = b - Ax^{(0)}$  for some initial guess  $x^{(0)}$ 
Choose  $\tilde{r}^{(0)}$  (for example,  $\tilde{r}^{(0)} = r^{(0)}$ ).
for  $i = 1, 2, \dots$ 
  solve  $Mz^{(i-1)} = r^{(i-1)} \rightarrow \text{Kernel 1 - GPU 0}$ 
  solve  $M^T \tilde{z}^{(i-1)} = \tilde{r}^{(i-1)} \rightarrow \text{Kernel 2 - GPU 0}$ 
   $\rho_{i-1} = z^{(i-1)T} \tilde{r}^{(i-1)} \rightarrow \text{Kernel 3 - GPU 0}$ 
  if  $\rho_{i-1} = 0$ , method fails
  if  $i = 1$ 
     $p^{(i)} = z^{(i-1)}$ 
     $\tilde{p}^{(i)} = \tilde{z}^{(i-1)}$ 
  else
     $\beta_{i-1} = \rho_{i-1} / \rho_{i-2} \rightarrow \text{Kernel 4 - GPU 0}$ 
     $p^{(i)} = z^{(i-1)} + \beta_{i-1} p^{(i-1)} \rightarrow \text{Kernel 5 - GPU 0}$ 
     $\tilde{p}^{(i)} = \tilde{z}^{(i-1)} + \beta_{i-1} \tilde{p}^{(i-1)} \rightarrow \text{Kernel 6 - GPU 0}$ 
  endif
   $q^{(i)} = Ap^{(i)} \rightarrow \text{Kernel 1 - GPU 1}$ 
   $\tilde{q}^{(i)} = A^T \tilde{p}^{(i)} \rightarrow \text{Kernel 2 - GPU 1}$ 
   $\alpha_i = \rho_{i-1} / \tilde{p}^{(i)T} q^{(i)} \rightarrow \text{Kernel 7 - GPU 0}$ 
   $x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)} \rightarrow \text{Kernel 8 - GPU 0}$ 
   $r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)} \rightarrow \text{Kernel 9 - GPU 0}$ 
   $\tilde{r}^{(i)} = \tilde{r}^{(i-1)} - \alpha_i \tilde{q}^{(i)} \rightarrow \text{Kernel 10 - GPU 0}$ 
  check convergence; continue if necessary
end

```

Fig. 7. Pseudo-code of PBiCG algorithm.

In PBiCG algorithm both forward and back substitution, as well as SpMV, must be performed twice, such as Fig. 7. Thus, three more steps are required in this method, compared to PCG method. The SpMV is executed in GPU 1, whereas the others operations in GPU 0.

D. Formats and Hardware Configuration

The following methods of accessing and manipulating entries of the matrix were implemented: Coordinate Format (COO) in the serial CPU code and Compressed Sparse Row Format (CSR) [3] in the GPU code.

The tests, for **Case 1**, were run on an Intel Core i7 with 3.4 GHz, 32GB RAM and a NVIDIA GeForce GTX 690 card with 3072 cores and 384 GB/sec memory. For **Case 2** the tests were run on an Intel Core i7, 3.4 GHz and NVIDIA GeForce GTX TITAN card (2688 cores and 288.4 GB/sec). All codes were built under *MS-Visual Studio* compiler, double precision, 64-bits arithmetics. For compiling the parallel code we used CUDA 5.0 with CUSPARSE and CUBLAS libraries [5].

V. RESULTS AND ANALYSES

A. Tested Configuration

All numerical experiments were carried out in one canonical configurations of grounding systems, namely a single bare electrode, vertically-buried in homogeneous soil, as the solution can be easily checked by analytical methods. In all cases the domains were meshed with either 8- or 20-node hexahedral elements. Details about dimensions and physical properties of the tested configuration are given in Table I [6].

TABLE I
DIMENSIONAL DATA OF THE TESTED CONFIGURATION

Rod length (m)	Rod radius (mm)	Soil resistivity ($\Omega \cdot m$)	Frequency (MHz)
32	4	450	1

We used either none or the UPML technique (uni-axial perfectly-matched layers) to truncate the domain, such as illustrated in Fig. 1. This technique is widely used, but has the drawback to considerably deteriorate the convergence of the iterative process, since it worsens the matrix conditioning [24]. Even in this canonical example, the only successful, convergent solution was achieved with the classical, sequential incomplete factorization with diagonal compensation, namely SIC and SILU.

B. Numerical experiments

Table II summarizes the experiments we have performed in order to evaluate the effectiveness of the GPU implementation of the solvers in the solution of systems for **Case 1** and **Case 2**. Five numerical experiments were accomplished, namely: MTH1, MTH2, MTH3, MTH4 and MTH5.

For **Case 1**, the experiments were conducted with six increasing mesh densities, as outlined in Table III, which includes the corresponding number of non-zero entries of the matrix, $nnzA$. For the situations MS1 to MS5 the domain was meshed with 8-node hexahedra, whereas for MS6 20-node hexahedra were used. For **Case 2**, three sizes of matrices, corresponding to three networks with different number of nodes, have been used, as outlined in Table IV.

These tests are aimed to illustrate the impact of the Single-GPU and Multi-GPU performances on both numbers of iterations and the achieved speedup.

TABLE II
NUMERICAL EXPERIMENTS (DOUBLE PRECISION) FOR CASE 1

Experiment	PML	RCM	SIC	PCG	PBiCG
MTH1			x	x	x
MTH2	x		x	x	x
MTH3		x	x	x	x
MTH4	x	x	x	x	x
MTH5			x	x	x

TABLE III
NON-ZERO ENTRIES OF THE MATRIX FOR SEVERAL MESH SIZES - CASE 1

Size	$nnzA$	Test case	$nnzA$
MS1	1,117,770	MS4	4,077,671
MS2	2,080,786	MS5	5,773,607
MS3	3,015,732	MS6	15,654,177

TABLE IV

NON-ZERO ENTRIES OF MATRICES OF 3 NETWORKS - CASE 2

Size	S1	S2	S3
$nnzA$	311,626	2,877,842	4,985,004

C. Case 1

Table V presents the total run times achieved with the experiments MTH1 and MTH2 (without renumbering) for three mesh densities with both SIC-PCG and SIC-PBiCG. The tests for Single-GPU were named as 1GPU, whereas for Multi-GPU, as 2GPU. This table shows the number of non-zero elements, $nnzU$, of the upper-triangular factorized matrix, total execution time of the PCG (CPU, 1GPU and 2GPU) and PBiCG (CPU, 1GPU and 2GPU) methods, and the number of iterations. Double precision was used in all implementations. It becomes evident the significant reduction in the GPU execution time compared to the sequential algorithm, for a given tolerance.

TABLE V
TOTAL EXECUTION TIME (S) OF PCG AND PBiCG - No RCM
RENUMBERING - CASE 1

Experiment		MESH SIZE - without PML		
MTH1		MS2	MS4	MS6
PCG	$nnzU$	1,062,281	2,079,971	7,894,125
	CPU	782.98	2,636.07	31,015.70
	iterations	458	788	2,524
	1GPU	8.86	21.45	136.54
	iterations	446	792	2,523
	2GPU	8.84	19.32	126.20
PBiCG	CPU	2,135.09	5,788.43	-
	iterations	632	887	-
	1GPU	15.91	39.33	264.21
	iterations	446	792	2,523
	2GPU	14.98	37.52	260.26
Experiment		MESH SIZE - with PML		
MTH2		MS2	MS4	MS6
PCG	CPU	1,063.50	3,996.18	58,660.20
	iterations	630	1,205	4,764
	1GPU	12.41	32.01	260.28
	iterations	627	1,182	4,823
	2GPU	11.61	29.72	250.80
	CPU	3,370.08	11,184.7	-
PBiCG	iterations	1,024	1,753	-
	1GPU	22.24	58.54	502.29
	iterations	627	1,182	4,823
	2GPU	20.92	57.97	500.67

In Table V, we can see that when running sequentially in the CPU the time for PCG method is around three times faster than PBiCG, whereas in 1GPU the time is about twice; with 2GPU the time is even lower, especially for the largest mesh. The corresponding speedup are plotted in Fig. 8 and illustrates the high impact in performance achieved with both the 1GPU and 2GPU implementations of PCG, for increased mesh densities, even with no reordering. The improvement achieved by the Multi-GPU, compared to a Single one, as well as the Single-CPU, becomes more significant for PCG method with growing problem sizes.

Fig. 8. Speedups 1GPU and 2GPU based on the serial CPU implementation for schemes MTH1 and MTH2, applied to several mesh size (PCG).

As it can be seen in Fig. 8, for example, in MS6 (MTH1) the speedup was around 227 with 1GPU, against 246 with 2GPU. This represents around 8% of reduction. The benefit of this Multi-GPU scheme is not so pronounced, since most of the execution time is spent in either data copy or communication. This is mainly due to copy of vectors from GPU 1 to GPU 0 in *MS-Win7* OS. A reduction in this task could be more significant by using peer-to-peer memory access. This is only supported for 64-bit process and devices of computing capability 2.0 and higher from the Tesla series.

Although at a first sight this could not be a significant gain, one must keep in mind that we are interested in solving large-scale problems, and the limited memory capacity of the single GPU devices would be an obstacle to this purpose.

Fig. 9 compares the performance of the CUDA implementation against the serial optimized C++ code without RCM scheme for the various grids outlined in Table III. It shows the total execution time as a function of the non-zero matrix entries on a log-log scale.

Fig. 9. Comparison of total run times of CPU and 1GPU implementation of PCG in experiments MTH1 and MTH2, with increasing grid size.

Table VI, as well as Fig. 10, show even better results with RCM reordered systems, as expected. They show the total run times achieved with the experiments MTH3 and MTH4, for several mesh densities with both PCG and PBiCG methods.

TABLE VI
TOTAL EXECUTION TIME (S) OF PCG AND PBiCG - WITH RCM
RENUMBERING - FOR CASE I

Experiment		MESH SIZE - without PML		
MTH3		MS2	MS4	MS6
PCG	nnzU	1,062,281	2,079,971	7,894,125
	CPU	412.30	1,374.40	8,018.48

	iterations	243	414	652
	1GPU	9.61	20.17	94.18
	iterations	245	416	654
	2GPU	9.54	17.85	87.62
PBiCG	1GPU	18.54	38.60	184.07
	iterations	245	416	654
	2GPU	18.35	38.44	180.70
Experiment		MESH SIZE - with PML		
MTH4		MS2	MS4	MS6
PCG	CPU	613.41	2,018.35	16,465.30
	iterations	365	617	1,356
	1GPU	14.24	29.75	192.76
	iter.	368	618	1,341
PBiCG	2GPU	14.72	29.98	190.34
	1GPU	27.68	57.20	377.15
	iter.	368	618	1,341
	2GPU	28.58	58.90	379.21

In all simulations the PBiCG run times were about twice as higher than the PCG times, which was also as expected, since the former requires extra operations (+ 1 SpMV + 1 back and forward substitution). The speedup increases with grid size and the highest impact was for MS6, though the slope was different when compared to Fig. 10.

Fig. 10. Speedups 1GPU and 2GPU based on the serial CPU implementation for schemes MTH3 and MTH4, applied to several mesh size (PCG).

For the same problem, Fig. 11 illustrates the impact of the speedup for MTH2 (without RCM) and for MTH4 (with RCM) of the PCG method. For MS6, the speedup of the MTH4 is three times higher than MTH2. This represents a reduction in runtime of around 24% when using RCM technique.

Fig. 11. Speedups 2GPU based on the serial CPU implementation for schemes MTH2 and MTH4, applied to several mesh size (PCG).

Fig. 12 illustrates the total execution time of the GPU-PBiCG implementation as a function of the non-zero matrix entries on a log-log scale, for MTH2 and MTH4.

Fig. 12. Comparison of total run times of 1GPU implementation of PBiCG in experiments MTH2 and MTH4, with increasing grid size.

The overall additional time spent in minor tasks (transfer + allocation) represents a negligible amount, around 1% of the total time. At least 98% of the total times shown in both Table IV and V are spent in the iterative solution process. In all cases, the highest speedup was achieved for both MTH1 and MTH2 (see Fig. 8). This is due to the larger number of PCG iterations needed for convergence without renumbering. However, the speedup is also significant for both MTH3 and MTH4, with RCM ordering (see Fig. 10).

Fig. 13 depicts the computational cost for the three main operations of PCG algorithm: forward and back substitution, and SpMV.

Fig. 13. Run time of 1GPU implemented PCG for MS6 case.

In all situations the total execution time for forward and back substitution was the highest one. As an example, in MTH2 the time spent in those two steps was 117.4s and 117s, respectively. This is roughly 89.84% of the overall computational cost. The SpMV took 18.46 s, whereas other minor operations took 8.09 s.

It is worth mentioning that the post-processing results for the relevant figures in Grounding System Analysis, e.g. ground impedance and ground potential rise, showed negligible deviations among the different performed experiments, as well as with the reference values.

Finally, in Fig. 14 we show the residual norm of PCG (MS6), where one can see that it can be unstable.

Fig. 14. Convergence rate history of 1GPU code for MS6 and PCG.

Indeed, the methods based on Lanczos process are initially unstable compared to others. However, in the end the residual norm decreases sharply. Especially for the class of problem we treated, PBiCG algorithm proved to be more stable and robust. The convergence for MTH3 and MTH4 was faster than for MTH1 and MTH2, indicating that renumbering techniques are highly recommended when dealing with systems issued from FEA. Despite the better performance of PCG compared to PBiCG, at least for the class of problem we treated, the latter proved to be more stable and robust. Moreover, the PCG implementation of ITL library (serial CPU) failed to converge in all tested situations.

D. Case 2

Table VII presents the total run times achieved with the experiment MTH5 for three network sizes with both SIC-PCG and SIC-PBiCG. The tests for Single-GPU were named as 1GPU, whereas for optimized C++ serial was named CPU.

This table shows the number of non-zero elements, $nnzU$, of the upper-triangular factorized matrix, total execution time of the PCG (CPU and 1GPU) and PBiCG (CPU and 1GPU) methods, and the number of iterations. Double precision was used in all implementations.

TABLE VII
TOTAL EXECUTION TIME (S) OF PCG AND PBiCG - FOR CASE 2

Experiment		MESH SIZE		
MTH5		S1	S2	S3
PCG	$nnzU$	207,771	1,552,212	2,688,691
	CPU	1,450.88	9,658.47	18,781.60
	1GPU	13.69	53.93	77.33
	iterations	3,175	3,719	4,184
PBiCG	1GPU	25.46	105.44	151.60
	iterations	3,175	3,719	4,184

Fig. 14 illustrates the impact of the speedup for MTH5 of the PCG method. For S3, the speedup was around 243 times faster than optimized C++ serial code. The incomplete Cholesky factorization took around 3% of the total run time for S1, 1.31% for S2 and 1.21% for S3, thus confirming our previous assertion that, even with a classical, fully sequential implementation for the preconditioner, the overall speedup was still substantial.

Fig. 14. Speedups of 1GPU based on the serial CPU implementation for MTH5, applied to three network sizes (Case 2).

VI. CONCLUSIONS

We have shown that high speedups can be achieved with the use of GPU-implemented PCG and PBiCG algorithms, with respect to its serial CPU counterpart, in the solution of systems at complex unknowns issued from the 3D edge-FEA of eddy currents, as well as those arising from power flow network analysis. As the preconditioning step represents a small fraction of the total solution time in such problems, the hybrid implementation we adopted, namely, SIC on CPU along with CG iterations on GPU, still provides significant gains in performance for the simulation of large scale problems in low-cost desktop PCs.

The impact of Multi-GPU was not so pronounced, mainly due to the extra time spent in both data copy and communication [14], [25]. However, in all tested cases some level of speedup was observed with the Multi-GPU device, besides their ability to increase memory capacity of GPU-based platforms. Thus, in future works, more efforts should be addressed to improve this issue, as well as to the search of more robust, parallelizable preconditioners.

REFERENCES

- [1] GPGPU. Website, 2014. <http://gpgpu.org/>
- [2] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proc. of IEEE*, vol. 96, n° 5, May. 2008, pp. 879-899.
- [3] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec. 2008.
- [4] Y. Saad, *Iterative Methods for Sparse Linear Systems*, New York: PWS Publishing, 2nd ed., 2003, pp. 275-369.
- [5] NVIDIA Corporation, <http://www.nvidia.com/>.
- [6] A. F. P. Camargos, V. C. Silva, J.-M. Guichon, G. Munier, "Efficient Parallel Preconditioned Conjugate Gradient Solver on GPU for FE Modeling of Electromagnetic Fields in Highly Dissipative Media," *IEEE Transactions on Magnetics*, vol. 50, n° 2, Feb. 2014..
- [7] L. B. Martinho, et al., "3D Finite Element Analysis of Conductive Coupling Problems in Transmission Line Rights of Way," *IEEE Transactions on Magnetics*, vol. 50, n° 2, Feb. 2014.
- [8] M. R. Hestenes and E. Stiefel, "Methods of conjugate gradients for solving linear systems," *Journal of Research of the National Bureau of Standards*, vol. 49, n° 6, Dec. 1952, pp. 409-436.
- [9] J. R. Shewchuk, "An introduction to the conjugate gradient method without the agonizing pain," Technical report, Pittsburgh, PA, USA, 1994.
- [10] A. Van der Sluis and H. A. Van der Vorst, "The rate of convergence of conjugate gradients," *Numer. Math.*, Vol. 48, No. 5, Springer-Verlag New York (1986), Inc., pp. 543-560. ISSN 0029-599X.
- [11] M. Naumov, "Incomplete-LU and Cholesky preconditioned iterative methods using CUSPARSE and CUBLAS," NVIDIA Corporation, Jun. 2011. <https://developer.nvidia.com>.
- [12] R. Li, Y. Saad, "GPU-accelerated preconditioned iterative linear solvers," <http://citeseerx.ist.psu.edu/index>.
- [13] H. Moghnieh, D. A. Lowther, "Understanding the efficiency of parallel incomplete Cholesky preconditioners on the performance of ICCG solvers for multi-core and GPU systems," *IEEE 14th Biennial Conference on Electromagnetic Field Computation (CEFC)*, Jan. 2010.
- [14] S. Ikuno, Y. Kawaguchi, N. Fujita, T. Itoh, S. Nakata, K. Watanabe, "Iterative solver for linear system obtained by edge element: variable preconditioned method with mixed precision on GPU," *IEEE Transactions on Magnetics*, vol. 48, Feb. 2012, pp. 467-470.
- [15] R. Helfenstein, J. Koko, "Parallel preconditioned conjugate gradient algorithm on GPU," *Journal of Computational and Applied Mathematics*, Elsevier, vol. 236, issue 15, Sep. 2012, pp. 3584-3590.
- [16] M. Ament, G. Knittel, D. Weiskopf, W. Straßer, "A parallel preconditioned conjugate gradient solver for the Poisson problem on a Multi-GPU platform," *IEEE 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, Feb. 2010, pp. 583-592.
- [17] MATLAB®, <http://www.mathworks.com/products/matlab/>.
- [18] A. Lumsdanie, J. Siek, MTL: The Matrix Template Library 2, <http://osl.iu.edu/research/mtl/mtl2.php3>, 1998.
- [19] A. Lumsdanie, J. Siek, ITL: the iterative template library. <http://www.osl.iu.edu/research/itl>, 1998.
- [20] K. Fujiwara, T. Nakata, H. Fusayasu "Acceleration of convergence characteristic of the ICCG Method," *IEEE Transactions on Magnetics*, vol. 29, n° 2, Mar. 1993, 1958-1961.
- [21] R. C. Mesquita, "Additional properties of the incomplete gauge formulation for 3-D nodal finite-element magnetostatics," *IEEE Transactions on Magnetics*, vol. 30, n° 5, Sep. 1994, 2861-4.
- [22] V. C. Silva, L. B. Martinho, J. R. Cardoso, M. L. Pereira Filho, S. L. L. Verardi, "Efficient modeling of thin wires in a lossy medium by finite elements applied to grounding systems," *IEEE Transactions on Magnetics*, vol. 47, n° 5, May. 2011, 966-969.
- [23] R. Barret, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, 1994.
- [24] B. Stupfel, "A Study of the Condition Number of Various Finite Element Matrices Involved in the Numerical Solution of Maxwell's Equations," *IEEE Antennas Propagat.*, v. 52(11), 3048-59, Nov. 2004.
- [25] A. W. O. Rodrigues, F. Guyomarc'h, J.-L. Dekeyser, Y. L. Menach, "Automatic Multi-GPU code generation applied to simulation of electrical machines," *IEEE Transactions on Magnetics*, vol. 48, n° 2, Feb. 2012, pp. 831-834.
- [26] R. Idema, et al., "Towards faster solution of large power flow problems," *IEEE Transactions on Power Systems*, vol.28, n° 4, Nov. 2013, pp. 4918-4925.
- [27] A. A. Kamiabad, J. E. Tate, "Polynomial preconditioning of power system matrices with graphics processing units". *High Performance Computing in Power and Energy Systems*, Springer Berlin Heidelberg, 2013, pp. 229-246.