JONAS LAERTE ANSONI

RESOLUÇÃO DE UM PROBLEMA TÉRMICO INVERSO UTILIZANDO
PROCESSAMENTO PARALELO EM ARQUITETURAS DE MEMÓRIA
COMPARTILHADA

Dissertação apresentada à Escola de
Engenharia de São Carlos da
Universidade de São Paulo para
obtenção do título de Mestre em
Engenharia Mecânica

Área de concentração: Térmica e
Fluidos

Orientador: Prof. Tit. Paulo Seleglim Júnior

São Carlos

-2010-

FOLHA DE JULGAMENTO

Candidato(a): Licenciado JONAS LAERTE ANSONI.

Dissertação defendida e julgada em 03.09.2010 perante a Comissão Julgadora:

Prof. Titular **PAULO SELEGHIM JUNIOR** – (Orientador)
(Escola de Engenharia de São Carlos/USP)

Aprovado

Prof. Dr. **PAULO CELSO GRECO JÚNIOR**
(Escola de Engenharia de São Carlos/USP)

Aprovado

Prof. Dr. **GONZALO TRAVIESO**
(Instituto de Física de São Carlos/USP)

Aprovado

Prof. Associado **JONAS DE CARVALHO**
Coordenador do Programa de Pós-Graduação em
Engenharia Mecânica

Prof. Titular **GERALDO ROBERTO MARTINS DA COSTA**
Presidente da Comissão da Pós-Graduação da EESC

Prof. Associado Paulo César Lima Segantine
Vice-Presidente da CPG

*“Aos meus pais,
Laerte (in memoriam) e Lourdes
com todo amor, admiração e gratidão.”*

Agradecimentos

Ao meu pai Laerte (*in memoriam*) pelo exemplo de vida.

À minha mãe Lourdes, pela compreensão de minha ausência em muitos momentos, pelo apoio incondicional e por nunca medir esforços para prover aos filhos recursos necessários para que conseguissem estudar.

Às minhas queridas irmãs pelo amor, paciência e incentivo.

À Fernanda Castilho Bueno pelo companheirismo, incentivo e apoio em momentos tão importantes da minha vida.

Ao meu orientador prof. Paulo Seleglim Júnior, pela oportunidade e confiança a mim depositada para a realização desse trabalho.

À Analice Costacurta Brandi e Anderson de Marco pela amizade e incentivo inabalável, durante a realização deste trabalho.

À todos os meus amigos de moradia pela convivência agradável e apoio por todos esses anos tornando-se minha segunda família.

À todos os meus amigos e colegas do laboratório pela ajuda direta e indireta na realização deste trabalho.

Aos professores e funcionários do Instituto de Ciências Matemáticas e de Computação (ICMC – USP) e da Escola de Engenharia de São Carlos (EESC – USP).

À CAPES pelo suporte financeiro.

Finalmente a todos que de alguma maneira contribuíram para a conclusão deste trabalho.

Resumo

ANSONI, J. L. **Resolução de um problema térmico inverso utilizando processamento paralelo em arquiteturas de memória compartilhada**. 2010. 139 f. Dissertação (Mestrado) – Escola de Engenharia de São Carlos da Universidade de São Paulo, São Carlos, 2010.

A programação paralela tem sido freqüentemente adotada para o desenvolvimento de aplicações que demandam alto desempenho computacional. Com o advento das arquiteturas *multi-cores* e a existência de diversos níveis de paralelismo é importante definir estratégias de programação paralela que tirem proveito desse poder de processamento nessas arquiteturas. Neste contexto, este trabalho busca avaliar o desempenho da utilização das arquiteturas *multi-cores*, principalmente o oferecido pelas unidades de processamento gráfico (GPUs) e CPUs *multi-cores* na resolução de um problema térmico inverso. Algoritmos paralelos para a GPU e CPU foram desenvolvidos utilizando respectivamente as ferramentas de programação em arquiteturas de memória compartilhada NVIDIA CUDA (*Compute Unified Device Architecture*) e a API POSIX Threads. O algoritmo do método do Gradiente Conjugado Pré-condicionado para resolução de sistemas lineares esparsos foi implementado totalmente no espaço da memória global da GPU em CUDA. O algoritmo desenvolvido foi avaliado em dois modelos de GPU, os quais se mostraram mais eficientes, apresentando um *speedup* de quatro vezes que a versão serial do algoritmo. A aplicação paralela em POSIX Threads foi avaliada em diferentes CPUs *multi-cores* com distintas microarquiteturas. Buscando um maior desempenho do código paralelizado foram utilizados *flags* de otimização as quais se mostraram muito eficientes na aplicação desenvolvida. Desta forma o código paralelizado com o auxílio das *flags* de otimização chegou a apresentar tempos de processamento cerca de doze vezes mais rápido que a versão serial no mesmo processador sem nenhum tipo de otimização. Assim tanto a abordagem utilizando a GPU como um co-processador genérico a CPU como a aplicação paralela empregando as CPUs *multi-cores* mostraram-se ferramentas eficientes para a resolução do problema térmico inverso.

Palavras-chave: Processamento paralelo. GPGPU CUDA. POSIX Threads. Matriz esparsa. Gradiente Conjugado Pré-Condicionado. Problema térmico inverso.

Abstract

ANSONI, J. L. **Resolution of an inverse thermal problem using parallel processing on shared memory architectures.** 2010. 139 f. MSc. Dissertation in Mechanical Engineering – School of Engineering of São Carlos, University of São Paulo, São Carlos, 2010.

Parallel programming has been frequently adopted for the development of applications that demand high-performance computing. With the advent of multi-cores architectures and the existence of several levels of parallelism are important to define programming strategies that take advantage of parallel processing power in these architectures. In this context, this study aims to evaluate the performance of architectures using multi-cores, mainly those offered by the graphics processing units (GPUs) and CPU multi-cores in the resolution of an inverse thermal problem. Parallel algorithms for the GPU and CPU were developed respectively, using the programming tools in shared memory architectures, NVIDIA CUDA (Compute Unified Device Architecture) and the POSIX Threads API. The algorithm of the Preconditioned Conjugate Gradient method for solving sparse linear systems entirely within the global memory of the GPU was implemented by CUDA. It evaluated the two models of GPU, which proved more efficient by having a speedup four times faster than the serial version of the algorithm. The parallel application in POSIX Threads was evaluated in different multi-core CPUs with different microarchitectures. Optimization flags were used to achieve a higher performance of the parallelized code. As those were efficient in the developed application, the parallelized code presented processing times about twelve times faster than the serial version on the same processor without any optimization. Thus both the approach using GPU as a coprocessor to the CPU as a generic parallel application using the multi-core CPU proved to be more efficient tools for solving the inverse thermal problem.

Keywords: Parallel processing. GPGPU CUDA. POSIX Threads. Sparse numerical solver. Preconditioned Conjugate Gradient algorithm. Inverse thermal problem.

Lista de Figuras

Figura 2.1 – Esquema do processador TILE-Gx100 (TILERA, 2010).	28
Figura 2.2 – Evolução da capacidade do GPU face à mesma evolução no CPU (NVIDIA, 2010).	29
Figura 2.3 – Representação da execução da taxonomia SISD (BARNEY, 2010a).....	31
Figura 2.4 – Representação da execução da taxonomia SIMD (BARNEY, 2010a).	31
Figura 2.5 – Representação da taxonomia MISD (BARNEY, 2010a).....	32
Figura 2.6 – Representação da execução da taxonomia MIMD (BARNEY, 2010a).....	33
Figura 2.7 – Tipos de memórias compartilhadas (BARNEY, 2010a).....	34
Figura 2.8 – Representação da memória distribuída (BARNEY, 2010a).	34
Figura 2.9 – Representação do sistema híbrido de memória (BARNEY, 2010a).....	35
Figura 2.10 – Execução de rotinas pelo tempo (BARNEY, 2010c).....	40
Figura 2.11 – Condição de corrida (BARNEY, 2010c).	41
Figura 2.12 – Possível fluxo de execução.	43
Figura 2.13 – Modelo de execução OpenMP (fork/join) (BARNEY, 2010b).	45
Figura 2.14 – Evolução da <i>Bandwidth</i> das GPUs e CPUs (NVIDIA, 2010).....	49
Figura 2.15 – Destinação dos transistores em CPUs e GPUs (NVIDIA, 2008).....	50
Figura 2.16 – Memória compartilhada (NVIDIA, 2007b).	50
Figura 2.17 – Operações de <i>gather</i> e <i>scatter</i> na memória da GPU (NVIDIA, 2007b).	51
Figura 2.18 – Pilha de software da plataforma CUDA (HALFHILL, 2008).	52
Figura 2.19 – Arquitetura da GPU (NVIDIA, 2008).....	53
Figura 2.20 – Organização de um <i>grid</i> com seus blocos de um <i>kernel</i> (NVIDIA, 2007b).	55
Figura 2.21 – Fluxo de execução de um programa CUDA.	57
Figura 2.22 – Caminho de execução de uma aplicação CUDA (NVIDIA, 2008).	58
Figura 2.23 – Execução simultânea do kernel (NVIDIA, 2009a).	59
Figura 2.24 – OpenCL: plataforma heterogênea CPU e GPU (KHRONOS, 2010).....	60
Figura 3.1 – Idéia básica do método de regularização.	66
Figura 3.2 – Geometria do problema simulado numericamente	70
Figura 4.1 – Esquema típico de um sistema que use o GPU como um co-processador.....	77
Figura 4.2 – Armazenamento das matrizes esparsas.	80
Figura 4.3 – Execução do método do PCG na GPU.....	83
Figura 4.4 – Geometria do problema de condução bidimensional em uma placa.....	85

Figura 4.5 – Comparação malha grossa (25 nós) entre soluções: (a) numérica e (b) analítica.	86
Figura 4.6 – Comparação malha intermediária (81 nós) entre as soluções: (a) numérica e (b) analítica.	86
Figura 4.7 – Comparação malha fina (289 nós) entre as soluções: (a) numérica e (b) analítica.	86
Figura 4.8 – Erro relativo na norma L_{∞} para o problema de condução de calor.	88
Figura 4.9 – Execução do código utilizando Pthreads.	91
Figura 5.1 – GPU 1 – GeForce 9800 GT.	94
Figura 5.2 – GPU 2 – TESLA C1060.	94
Figura 5.3 – CPU 1 – Core 2 Duo.	95
Figura 5.4 – CPU 2 – Core 2 Quad.	95
Figura 5.5 – CPU 3 – Core i7 975.	95
Figura 5.6 – Geometria do problema simulado numericamente	97
Figura 5.7 – Estimativas iniciais em GPU e solução real do coeficiente de convecção, conforme a equação (5.2) caso 1.1.	100
Figura 5.8 – Estimativas iniciais em GPU e solução real do coeficiente de convecção, conforme a equação (5.2) caso 1.2.	100
Figura 5.9 – Estimativas iniciais em GPU e solução real do coeficiente de convecção, conforme equação (5.2) caso 1.3.	101
Figura 5.10 – Estimativas iniciais em GPU e solução real do coeficiente de convecção, conforme a equação (5.3) caso 2.1.	101
Figura 5.11 – Estimativas iniciais em GPU e solução real do coeficiente de convecção, conforme a equação (5.3) caso 2.2.	102
Figura 5.12 – Estimativas iniciais em GPU e solução real do coeficiente de convecção, conforme a equação (5.3) caso 2.3.	102
Figura 5.13 – Estimativas iniciais em GPU e solução real do coeficiente de convecção, conforme a equação (5.4) caso 3.1.	103
Figura 5.14 – Estimativas iniciais em GPU e solução real do coeficiente de convecção, conforme a equação (5.4) caso 3.2.	103
Figura 5.15 – Estimativas iniciais em GPU e solução real do coeficiente de convecção, conforme a equação (5.4) caso 3.3.	104
Figura 5.16 – Tempo de processamento dos casos analisados serial e paralelo em GPU.	105

Figura 5.17 – <i>Speedups</i> obtidos com o processamento em GPU.	105
Figura 5.18 – Números de iterações para convergência de um sistema linear esparso utilizando os métodos PBCG (precisão dupla em CPU) e o PCG (precisão simples em GPU).	106
Figura 5.19 – Diferença da resolução de um sistema linear esparso do problema utilizando precisão simples e dupla.	107
Figura 5.20 – Estimativas em Pthreads e solução real do coeficiente de convecção, conforme a equação (5.2) caso 1.1.....	108
Figura 5.21 – Estimativas em Pthreads e solução real do coeficiente de convecção, conforme a equação (5.2) caso 1.2.....	108
Figura 5.22 – Estimativas em Pthreads e solução real do coeficiente de convecção, conforme a equação (5.2) caso 1.3.....	109
Figura 5.23 – Estimativas iniciais em Pthreads e solução real do coeficiente de convecção, conforme a equação (5.3) caso 2.1.....	109
Figura 5.24 Estimativas iniciais em Pthreads e solução real do coeficiente de convecção, conforme a equação (5.3) caso 2.2.....	110
Figura 5.25 – Estimativas iniciais em Pthreads e solução real do coeficiente de convecção, conforme a equação (5.3) caso 2.3.....	110
Figura 5.26 – Estimativas iniciais em Pthreads e solução real do coeficiente de convecção, conforme a equação (5.4) caso 3.1.....	111
Figura 5.27 – Estimativas iniciais em Pthreads e solução real do coeficiente de convecção, conforme a equação (5.4) caso 3.2.....	111
Figura 5.28 – Estimativas iniciais em Pthreads e solução real do coeficiente de convecção, conforme a equação (5.4) caso 3.1.....	112
Figura 5.29 – Tempo de processamento dos casos analisados serial e paralelo em CPU.....	113
Figura 5.30 – <i>Speedups</i> obtidos com o processamento paralelo Pthreads.....	113
Figura 5.31 – Tempo de processamento dos casos analisados serial e paralelo com auxílio de <i>flags</i> de otimização.....	115
Figura 5.32 – <i>Speedups</i> obtidos com o processamento paralelo Pthreads com <i>flag</i> de otimização.	115
Figura 5.33 – Escalabilidade das CPUs com o processamento com uma <i>thread</i>	116
Figura C.1 – Pseudocódigo do método PBCG.	136

Lista de Tabelas

Tabela 2.1 – Ferramentas de programação em arquitetura de memória compartilhada.....	62
Tabela 4.1 – Número de vezes que o método PBCG é chamado e porcentagem de tempo gasto na montagem da matriz Hessiana	75
Tabela 4.2 – Erros para o problema de calor definido na Figura 4.4.	87
Tabela 5.1 – Características das GPUs utilizadas no desenvolvimento do trabalho.	94
Tabela 5.2 – Característica das CPUs utilizadas no desenvolvimento do trabalho.....	95
Tabela B.1 – Armazenamento <i>row-indexed sparse storage mode</i>	132
Tabela D.1 – Tempos de processamento e desvio padrão dos códigos serial e paralelo em GPU.....	137
Tabela E.1 – Tempos de processamento e desvio padrão dos códigos serial e paralelo em CPU.....	139

Lista de Algoritmos

Algoritmo 1: Exemplo de utilização de Pthreads em C.....	42
Algoritmo 2: Paralelização de regiões – diretiva <i>parallel</i>	45
Algoritmo 3: Paralelização de laços – diretiva <i>for</i>	47
Algoritmo 4: Estrutura de dados para armazenamento das N matrizes esparsas	79
Algoritmo 5: Armazenamento da matriz esparsa	79
Algoritmo 6: Pseudocódigo para o algoritmo dos Gradientes Conjugados Pré- condicionado (PCG).....	81
Algoritmo 7: <i>Row-indexed sparse storage mode</i>	132

Lista de Símbolos

Símbolo	Descrição
d	Direção de refinamento
$diag$	Matriz diagonal
e	Funcional de erro ou função objetivo
h	Coeficiente de convecção ou coeficiente de transferência de calor
$h_{inicial}$	Coeficiente de convecção inicial referente a estimativa inicial
h_{num}	Coeficiente de convecção numérico resultante do modelo prospectivo
h_{real}	Coeficiente de convecção real resultante do modelo de referência
H	Matriz Hessiana
i_{max}	Número máximo de iterações
k	Condutividade térmica
n	Vetor normal
q	Fluxo de calor
R	Resíduo
T	Temperatura
T_{∞}	Temperatura ambiente
U	Matriz ortogonal
v	Função de ponderação
x	Eixo horizontal
y	Eixo vertical
δ_x	Espaçamento da malha na direção x
δ_y	Espaçamento da malha na direção y
ε	Valor de tolerância de erro
λ	Passo de refinamento

Símbolo	Descrição
θ	Potencial térmico ou temperatura
θ_0	Temperatura inicial conhecida
θ_{num}	Temperatura numérica resultante do modelo prospectivo
θ_{real}	Temperatura real resultante do modelo de referência
θ_∞	Temperatura ambiente
Ω	Domínio do problema
$\partial\Omega$	Fronteira do domínio Ω

Lista de Siglas

API	Application Programming Interface
BCRS	Block Compressed Row Storage
CFD	Dinâmica dos Fluidos Computacional
CG	Gradiente Conjugado
CNC	Concurrent Number Cruncher
CPU	Central Processing Unit
CSR	Compressed Sparse Row
CUDA	Compute Unified Device Architecture
GPGPU	General Purpose Graphics Processing Unit
GPU	Graphic Processing Unit
MEF	Método dos Elementos Finitos
MIMD	Multiple Instruction, Multiple Data
MISD	Multiple Instruction, Single Data
MPI	Message Passing Interface
MPMD	Multiple Program Multiple Data
NUMA	Non-Uniform Memory Access
OpenCL	Open Computing Language
OpenMP	Open Multi-Processing
PBCG	Gradientes Biconjugados Pré-Condicionados
PCG	Gradientes Pré-Condicionados
PCs	Personal Computers
POSIX	Portable Operating System Interface
PVM	Parallel Virtual Machine
SIMD	Single Instruction, Multiple Data
SIMT	Single Instruction Multiple Thread
SISD	Single Instruction, Single Data
SM	Streaming multiprocessor
SMP	Symmetric Multiprocessing
SP	Scalar Processor
UMA	Uniform Memory Access

Sumário

CAPÍTULO 1	INTRODUÇÃO	23
1.1	Motivação	23
1.2	Escopo do trabalho	25
1.3	Organização do trabalho	25
CAPÍTULO 2	MODELOS DE COMPUTAÇÃO PARALELA	27
2.1	Introdução	27
2.2	Computadores paralelos	30
2.2.1	Noção de Thread	30
2.2.2	Classificação dos computadores paralelos	30
2.2.3	Arquitetura da memória em computadores paralelos.....	33
2.2.4	Modelos de programação paralela.....	35
2.2.5	Desempenho de um algoritmo paralelo.....	38
2.3	Ferramentas para programação paralela em arquitetura de memória compartilhada	39
2.3.1	POSIX Threads	39
2.3.2	OpenMP	44
2.3.3	NVIDIA CUDA	48
2.3.4	OpenCL	59
2.3.5	Comparativo entre as ferramentas de programação paralela	60
CAPÍTULO 3	PROBLEMA ABORDADO E SUA MODELAGEM	63
3.1	Tomografia Térmica.....	63
3.2	Métodos de Solução de Problemas Inversos e Técnicas de Regularização	65
3.3	Modelagem do Problema de Condução de Calor.....	66
3.4	Medição não intrusiva do coeficiente de convecção formulada como um problema térmico inverso.....	69
3.5	Métodos de Otimização e Estratégia de Regularização	71

CAPÍTULO 4	IMPLEMENTAÇÃO NA ARQUITETURA PARALELA.....	75
4.1	Implementação na GPU – NVIDIA CUDA	76
4.1.1	Armazenamento da matriz esparsa	78
4.1.2	Método dos Gradientes Conjugados na GPU	80
4.1.3	Verificação do modelo numérico e análise de convergência do Código em GPU	84
4.2	Implementação na CPU – Pthreads	88
4.2.1	Armazenamento da matriz esparsa	88
4.2.2	Resolução do sistema linear esparsa.....	89
4.2.3	Paralelização em Pthreads	90
CAPÍTULO 5	TESTES NUMÉRICOS E RESULTADOS	93
5.1	Infraestrutura para a realização dos testes	93
5.2	Aplicação em um problema térmico inverso	96
5.3	Aplicação com o NVIDIA CUDA.....	98
5.4	Aplicação com o Pthreads	107
CAPÍTULO 6	CONCLUSÕES E PERSPECTIVAS.....	117
REFERÊNCIAS BIBLIOGRÁFICAS.....		121
APÊNDICE A	PCG NA GPU – NVIDIA CUDA	127
APÊNDICE B	ALGORITMO PARA ARMAZENAMENTO DA MATRIZ ESPARSA – PTHREADS	131
APÊNDICE C	MÉTODO DOS GRADIENTES BICONJUGADOS PRECONDICIONADOS	133
APÊNDICE D	TABELA COM OS TEMPOS E OS DESVIOS PADRÕES DOS CASOS ANÁLISADOS EM GPU.....	137
APÊNDICE E	TABELA COM OS TEMPOS E OS DESVIOS PADRÕES DOS CASOS ANÁLISADOS EM PTHREADS	139

CAPÍTULO 1 INTRODUÇÃO

1.1 Motivação

Este trabalho teve início em virtude da necessidade premente em encontrar novas soluções em processamento científico para aplicações progressivamente mais complexas e computacionalmente custosas. Problemas e soluções devem e podem ser mais bem explorados por uma estrutura computacional mais robusta e potente. Nesse trabalho se explora uma nova tendência de processamento utilizando a programação paralela *multi-thread* em arquiteturas de memórias compartilhadas para acelerar a resolução de um problema térmico inverso. Mais precisamente se explora uma nova tecnologia de processamento escalável, baseada em unidades de processamento gráfico (GPUs) e CPUs *multi-cores*, que segundo os fabricantes de processadores (BROOKWOOD, 2010; NICKOLLS et al., 2008; SEILER et al., 2008; WILLIAMS et al., 2006) impõem ganhos substanciais de velocidade em diversas aplicações em computação científica.

Isto se traduz em novas propostas de paralelismo computacional baseadas em GPU e CPU *multi-cores* que tirando vantagem da velocidade de comunicação da memória compartilhada podem fornecer à área de simulação computacional o poder computacional necessário para fazer frente a antigos e novos problemas que exigem grande grau de processamento.

A evolução do computador programável viabilizou o aparecimento e desenvolvimento de novas áreas de pesquisas na matemática, física, engenharias e biologia. Mas ao longo das duas últimas décadas com o desenvolvimento de tecnologias de paralelismo em memória distribuída e troca de informações entre computadores pessoais (PCs) como o MPI (*Message Passing Interface*) e PVM (*Parallel Virtual Machine*), possibilitou a organização destas máquinas em aglomerados de computadores de menores custos tornando-se acessível o processamento de grandes quantidades de dados, levando as pesquisas a outro nível.

Concomitantemente à evolução do *hardware* dos computadores tornou-se possível que vários processadores operassem juntos em memória compartilhada ou SMP (*Symmetric Multiprocessing*) e como solução para manipular os processos e acessos à memória neste

novo paradigma surgiram novas ferramentas como o OpenMP e o POSIX Threads. Estas novas ferramentas computacionais ampliaram o interesse e a busca de novos algoritmos paralelos e novas maneiras de implementá-los nestas novas arquiteturas de memória compartilhada.

Acompanhando essa constante evolução do hardware e impulsionada pela crescente complexidade do processamento gráfico e financiada pela indústria dos jogos, as placas gráficas passaram por um grande progresso tecnológico. Atualmente as placas gráficas são pesadamente paralelas (são constituídas de diversos processadores) e dedicadas a realização de cálculos computacionalmente intensivos.

Tendo em vista esse enorme potencial computacional oferecido pelas GPUs a NVIDIA em fevereiro de 2007 tornou pública uma nova plataforma de software denominado CUDA (*Compute Unified Device Architecture*). Com o CUDA é possível explorar o potencial computacional das GPUs, até então restrito à computação gráfica, para a computação em geral.

Desde o seu lançamento diversos pesquisadores, cientistas e empresas têm apresentado novas soluções utilizando esse novo processamento altamente paralelo em GPU em áreas diversas como: CFD (Dinâmica dos Fluidos Computacional) (CORRIGAN et al., 2009; JACOBSEN; THIBAUT; SENOCAR, 2010; ZHAO, 2008), resolução de integrais estocásticas (JANUSZEWSKI; KOSTUR, 2009), tomografia médica (NOËL et al., 2008), análise de mercado financeiro (PREIS et al., 2009), biologia computacional (BHANDARKAR et al., 2010), e outros.

Embora tecnologicamente diferentes todos estes modelos de processamento em CPU *multi-core* e GPU podem ser classificados como sendo de memória compartilhada. O objetivo desse trabalho é desenvolver algoritmos que tirem proveito dessas arquiteturas paralelas de memórias compartilhadas (GPU e CPU *multi-core*) na aceleração da resolução de um problema térmico inverso.

Para isso foi necessário desenvolver algoritmos em linguagem C com a API POSIX Threads, sobre computadores com dois, quatro e oito núcleos e estruturas de dados para o armazenamento das matrizes esparsas originadas pela resolução do problema de modo a testar uma nova estratégia de utilização do poder de processamento da GPU para a resolução de sistemas lineares esparsos.

1.2 Escopo do trabalho

O escopo deste trabalho é então, avaliar a possibilidade de acelerar a resolução de um problema térmico inverso desenvolvido por Brandi (2010) recorrendo ao modelo de hardware e de programação fornecido pelo CUDA (GPU) e pelo POSIX Thread (CPU).

As principais contribuições deste trabalho são:

- Utilização de um modelo de hardware e programação para GPU, apresentando o GPU como um verdadeiro co-processador genérico.
- Desenvolvimento de uma estrutura de dados própria de modo a possibilitar o armazenamento de várias matrizes esparsas e possibilitar a sua transferência para a memória global da GPU.
- Desenvolvimento de um algoritmo inovador, recorrendo a este modelo, para a resolução de vários sistemas lineares esparsos, especialmente desenhados para que possam ser executados totalmente no espaço da memória global da placa gráfica.
- Realização de um estudo da convergência do método desenvolvido utilizando precisão simples e um estudo sobre a escalabilidade do código desenvolvido para diferentes GPUs, com diferentes quantidades de multiprocessadores.
- Desenvolvimento de um algoritmo eficiente para o processamento em paralelo utilizando o poder de processamento de CPUs *multi-cores*, utilizando a API POSIX Thread.
- Realização de um estudo da escalabilidade das CPUs com diferentes quantidades de núcleos e microarquitecturas.

1.3 Organização do trabalho

Os capítulos deste trabalho estão distribuídos da seguinte forma:

- No Capítulo 2 analisam-se alguns aspectos da computação paralela, onde são descritas as arquiteturas dos computadores paralelos bem como as ferramentas utilizadas para a programação paralela com arquitetura de memória compartilhada, que utilizaremos neste trabalho.
- No Capítulo 3 é apresentada uma introdução dos temas envolvidos no trabalho como tomografia térmica e problemas inversos e logo em seguida a formulação

matemática e numérica do problema de transferência de calor, bem como a discretização das equações pelo método dos elementos finitos (MEF).

- No Capítulo 4 apresentam-se os métodos empregados na construção dos códigos paralelizados em GPU e CPU. Neste capítulo é descrita a estrutura de dados desenvolvida para o armazenamento da matriz esparsa e o método de resolução dos sistemas lineares implementado na GPU e sua verificação de convergência. Em seguida é apresentada a estrutura de armazenamento das matrizes esparsas e o método de resolução dos sistemas lineares utilizados no código paralelizado na CPU.
- No Capítulo 5 são apresentadas as validações dos modelos numéricos paralelizados e a análise de convergência que foram aplicados a um estudo de caso.

Por fim, são apresentadas as conclusões da pesquisa desenvolvida e perspectivas para continuidade do trabalho.

CAPÍTULO 2 MODELOS DE COMPUTAÇÃO PARALELA

Este capítulo traz uma revisão dos modelos de processamento paralelo, que foi empregado para acelerar o tempo de execução da resolução do problema térmico inverso desenvolvido por Brandi (2010), sendo eles o NVIDIA CUDA e o POSIX Thread e de modelos que podem ser usados em trabalhos futuros como o OpenMP (*Open Multi-Processing*) e o OpenCL (*Open Computing Language*). Pretende-se estabelecer uma introdução às diferentes arquiteturas de computadores paralelos, dando ênfase na arquitetura de memória compartilhada e *multi-thread* os quais foram empregados neste trabalho, aos modelos de computação paralela e a algumas técnicas para a avaliação de desempenho de algoritmos paralelos.

2.1 Introdução

Os computadores estão em constante evolução, os processadores em especial são exemplos desta melhora tornando-se cada vez mais rápidos. Porém, assim como a capacidade dos processadores, a complexidade dos problemas também aumentou. Atualmente, problemas como os de simulação computacional de meteorologia, cristalografia, tomografia, problemas estruturais, etc., não podem ser resolvidos em tempo hábil por um computador contendo um único processador (computador sequencial), pois requerem um alto poder de processamento, no qual um único processador sozinho não pode oferecer. Visando suprir essas necessidades surgiu a computação de alto desempenho com o processamento em computadores paralelos, desejando-se obter o máximo possível em cálculos científicos e outras tarefas onde o desempenho é um fator crítico.

Existem diferentes técnicas de paralelismo, uma delas é a distribuição de pequenos problemas em memória distribuída em clusters de computadores (STERLING, 2001), a qual a comunidade científica já utiliza desde a década de 90. Entretanto esta técnica apresenta alguns custos elevados, como por exemplo, o custo relacionado ao suprimento de energia dos clusters de alto desempenho; pois além do custo da tarifa ao alimentar os computadores deve-se levar

em consideração a compra e a manutenção de dispendiosos sistemas de refrigeração, estabilizadores e *no-breaks*.

A introdução recente de processadores *multi-core*, colocando processadores em paralelo em um único chip, permitiu aos fabricantes de CPUs melhorias de desempenho e a possibilidade de resolver problemas mais complexos. Um exemplo dessa evolução são os processadores TILE-Gx100 que chegam a ter 100 *cores*, como pode ser observado na Figura 2.1, desenvolvido pela *Tilera Corporation*, uma empresa que nasceu a partir de um grupo de pesquisadores do *Massachusetts Institute of Technology* (MIT), que trouxe ao mercado a pesquisa do Dr. Anant Agarwal sobre a arquitetura de malha baseada em *multi-core* (*multi-core mesh-based*) (TILERA, 2010). Estações de trabalho com mais de um processador *multi-core* estão disponíveis no mercado, como os processadores Intel Core 2 Duo; de forma que a base necessária para a programação paralela está lançada, mesmo para o pequeno usuário doméstico.

Figura 2.1 – Esquema do processador TILE-Gx100 (TILERA, 2010).

Durante a última década, acompanhando o ritmo acelerado de evolução das CPUs e sob a influência da indústria dos jogos na tentativa de fornecer simulações cada vez mais detalhadas e realistas levou a uma tal evolução da capacidade de processamento das placas gráficas que o GPU moderno chegou a apresentar performances muito superiores às do CPU, no que diz respeito ao número de operações de ponto flutuante efetuadas por unidade de tempo, como pode ser observado na Figura 2.2.

Figura 2.2 – Evolução da capacidade do GPU face à mesma evolução no CPU (NVIDIA, 2010).

Nos últimos anos, o GPU evoluiu de uma peça de hardware de funcionalidades fixas sobre primitivas gráficas, para uma peça com um processador paralelo programável, com desempenho superior às do CPU moderno, em particular no que diz respeito a operações vetoriais paralelas. Mais especificamente, o GPU moderno é particularmente apropriado a problemas que possam ser expressos como computações paralelas com alta intensidade de cálculo matemático, pois contém vários processadores especializados em processar grandes quantidades de dados em paralelo.

Entretanto para tirar proveito de toda essa evolução do hardware é necessário desenvolver algoritmos específicos que juntamente com os processadores possibilitem obter o mais alto desempenho computacional possível. Esse é o nosso objetivo, buscamos desenvolver algoritmos paralelos que buscam tirar o máximo proveito desse poder de processamento tanto das CPUs *multi-core* como da nova geração das placas gráficas GPUs, na aplicação de um problema térmico inverso.

A solução do problema térmico inverso desenvolvido por Brandi (2010) que se pretende acelerar envolve fundamentalmente operações de cálculo vetorial e matricial. O método iterativo utilizado para a resolução do sistema linear de equações gerado é dominado fundamentalmente por operações de multiplicação entre matrizes esparsas e vetores densos, como será apresentado.

2.2 Computadores paralelos

Por computador paralelo segundo Almasi e Gottlieb (1990, 1994), entende-se um sistema computacional contendo múltiplos processadores que se comunicam e cooperam para resolver grandes problemas mais rapidamente. Os processadores e a memória de um computador podem ser organizados de diversas formas, por isso existem diferentes arquiteturas e algumas taxonomias/classificações que agrupam arquiteturas com características comuns que serão apresentadas a seguir.

2.2.1 Noção de Thread

Uma *thread* é uma linha de execução independente que possui uma memória compartilhada com o processo pai e que pode ser escalonada pelo sistema operacional. Normalmente, a utilização de *threads* ocorre em conjunto com a repartição dos dados a serem processados. Dessa forma, cada *thread* pode processar um subconjunto de dados paralelamente.

2.2.2 Classificação dos computadores paralelos

A classificação ou taxonomia de uma arquitetura de processamento proposta por Flynn em 1972 (FLYNN, 1972) é a mais conhecida. As quatro classificações definidas por ele se baseiam no número de instruções concorrentes e nos fluxos de dados disponíveis na arquitetura.

➤ *Single Instruction, Single Data (SISD)*

O SISD que em português significa Instrução Única, Dados Únicos é a taxonomia seqüencial, no qual não explora o paralelismo, seja por instruções ou fluxo de dados. Corresponde a arquitetura de Von Neumann. Existe um único fluxo de instruções, no qual são executadas seqüencialmente e cada instrução opera sobre um único fluxo de dados.

Na Figura 2.3 observa-se a representação da execução da taxonomia SISD, na qual enquanto uma instrução é executada a outra aguarda o seu término.

Figura 2.3 – Representação da execução da taxonomia SISD (BARNEY, 2010a).

➤ ***Single Instruction, Multiple Data (SIMD)***

Em arquiteturas SIMD (em português Instrução Única, Dados Múltiplos) existe um único fluxo de instruções que opera sobre múltiplos fluxos de dados. Aplicações que precisam executar a mesma operação em grandes vetores ou matrizes podem tirar proveito desse tipo de arquitetura (STALLINGS, 2002; TANENBAUN; GOODMAN, 1999). Atualmente GPUs, FPGAs se encontram nesta categoria, bem como os processadores vetoriais.

Na Figura 2.4 observa-se a representação de execução da taxonomia SIMD, onde a mesma instrução com dados diferentes é executada paralelamente.

Figura 2.4 – Representação da execução da taxonomia SIMD (BARNEY, 2010a).

➤ ***Multiple Instruction, Single Data (MISD)***

Em arquitetura MISD (Instruções Múltiplas, Dados Únicos), por definição, múltiplos fluxos de instrução operam sobre um mesmo fluxo de dados ao mesmo tempo. Não se

conhece muitos exemplos de máquinas desse tipo e ele é citado apenas por uma questão de completude.

Na Figura 2.5 observa-se uma representação da execução da classificação MISD, nota-se que o mesmo dado é utilizado para efetuar operações diferentes em múltiplos fluxos.

Figura 2.5 – Representação da taxonomia MISD (BARNEY, 2010a).

➤ *Multiple Instruction, Multiple Data (MIMD)*

Em arquitetura MIMD (Instrução Múltipla, Dados Múltiplos) existem dois ou mais processadores, cada qual executando um fluxo de instruções distinto sobre um fluxo de dados próprio, em espaços de memória diferentes. Podemos associá-los a tecnologias de processamento distribuído tais como o PVM e o MPI.

Na Figura 2.6 observa-se a execução de múltiplas instruções como a operação de multiplicação em $P1$, soma em $P2$ e exponencial em Pn sendo aplicados sobre dados diferentes ao mesmo tempo.

Figura 2.6 – Representação da execução da taxonomia MIMD (BARNEY, 2010a).

2.2.3 Arquitetura da memória em computadores paralelos

A classificação de Flynn é habitualmente considerada incompleta, por não representar diversos subtipos de arquiteturas que diferem com relação à organização da memória. Apresenta-se a seguir os diferentes tipos de arquiteturas de memórias.

➤ *Memória compartilhada*

Computadores com memória compartilhada possuem mais de um processador e uma área de memória comum a todos. Os processadores executam de forma independente uns dos outros, mas têm acesso aos mesmos endereços de memória, de forma que mudanças na memória realizadas por um sejam imediatamente visíveis a todos os demais processadores. Os computadores com memória compartilhada mais comuns atualmente são os multiprocessadores simétricos (SMP) e os sistemas com acesso não uniforme à memória (NUMA). Em um SMP processadores e memória são interconectados por um barramento, ou algum tipo de circuito de interconexão interno, fazendo com que o tempo de acesso a qualquer posição da memória seja o mesmo, independente do processador que está realizando o processamento (STALLINGS, 2002). Por este motivo, esse tipo de sistema é denominado como sistema com acesso uniforme à memória (UMA - *Uniform memory access*). Em um sistema NUMA, ao contrário, os tempos variam.

Figura 2.7 – Tipos de memórias compartilhadas (BARNEY, 2010a).

➤ *Memória distribuída*

Sistemas com memória distribuída geralmente são compostos por diversas unidades de processamento conectadas por uma rede de dados, onde cada processador possui uma área de memória local apenas. Os processadores se comunicam trocando/passando mensagens através de uma rede, que pode ser uma rede proprietária de alta velocidade ou uma rede comercial. Dentre os sistemas com memória distribuída, os agregados de computadores (ou *clusters* de computadores) são os exemplos de utilização. Um agregado consiste de uma coleção de computadores autônomos e de uma rede (por exemplo, uma rede *Fast Ethernet*), que interconecta estes computadores.

Figura 2.8 – Representação da memória distribuída (BARNEY, 2010a).

➤ *Sistemas híbridos*

São sistemas onde ambas as formas de organização de memória estão presentes. São comuns em sistemas distribuídos, onde cada computador presente no sistema possui duas ou mais unidades de processamento que compartilham endereços de memória entre si. A grande maioria dos supercomputadores possui arquitetura híbrida de memória.

Figura 2.9 – Representação do sistema híbrido de memória (BARNEY, 2010a).

2.2.4 Modelos de programação paralela

Desenvolver algoritmos paralelos é uma tarefa mais complexa que desenvolver algoritmos sequenciais. Além de todas as abstrações usadas no modelo sequencial de programação usam outros conceitos como: comunicação entre tarefas, concorrência, escalabilidade e localidade.

A comunicação entre tarefas diz respeito à necessidade de troca de dados entre duas tarefas. Dependendo do modelo de programação considerado, as tarefas não possuem acesso direto a todos os dados relacionados ao problema e é necessário que dados sejam trocados entre elas. A localidade diz respeito a quais dados são locais a cada tarefa ou não. A escalabilidade é a capacidade de a aplicação ser eficiente a medida que ocorrem acréscimos na quantidade de processadores em que ela é executada. A concorrência é quando há o acesso simultâneo por duas ou mais tarefas a um recurso do sistema, como um endereço de memória por exemplo. Daremos a seguir uma introdução aos modelos de programação paralela mais usados (CULLER; SINGH; GUPTA, 1998).

➤ *Memória compartilhada (Shared memory)*

No modelo de programação de memória compartilhada as tarefas compartilham um mesmo espaço de endereçamento, que podem ler e escrever de forma assíncrona. Nesses sistemas a programação pode ser simplificada pela ausência da necessidade de troca explícita de dados entre as tarefas, porém torna-se mais complicado controlar a localidade das ações da memória. Frequentemente é necessário o uso de semáforos ou travas (*locks*) para controlar o acesso de escrita na memória. As tarefas podem, eventualmente, serem independentes umas das outras e trabalhar com diversos conjuntos de dados diferentes, apesar de compartilharem um mesmo espaço de endereçamento.

Alguns exemplos de interfaces de programação de aplicativos (API – *Application Programming Interface*¹) para programação em ambientes de memória compartilhada são POSIX Threads (Pthreads), OpenMP e OpenCL, que serão apresentadas com mais detalhes a seguir. O CUDA também é considerado uma API para programação em memória compartilhada. A seguir serão apresentados mais detalhes destas APIs.

➤ *Threads*

Neste modelo, as *threads* entende-se como “tarefas” têm uma memória local e podem executar de forma independente umas das outras. Em alguns casos duas ou mais tarefas podem compartilhar uma área de endereçamento de memória de forma similar ao modelo de memória compartilhada. Tarefas podem ser criadas ou destruídas dinamicamente a medida que o programa é executado.

Aplicações onde se possa realizar a decomposição funcional, ou seja, o particionamento da aplicação em grupos de tarefas distintas encaixa-se bem nesse modelo. A implementação de modelos utilizando *threads* não é nova em computação. Historicamente fabricantes de hardware aplicavam suas próprias versões proprietárias de *threads*. Essas implementações diferenciavam substancialmente entre si, tornando difícil para os programadores desenvolver suas aplicações. Para resolver isso foram desenvolvidas as especificações APIs para a programação baseada em *threads* POSIX Thread, normalizada pela IEEE e a OpenMP desenvolvida pelo *The Open Group*. Além dessas APIs existe também um *framework*² para a escrita de programas que executam em plataformas heterogêneas como CPUs, GPUs e outros processadores que controla a utilização de *threads*, o OpenCL que será melhor explicado a seguir.

➤ *Troca de mensagens (Message Passing)*

O modelo de troca ou passagem de mensagens é um dos mais populares para a programação paralela, baseada no envio e recebimento de mensagens através da rede seguindo as regras do protocolo de comunicação entre vários processadores que possuam memória própria. As informações são enviadas da memória local do processo para a memória local do processo remoto. A transferência dos dados entre os processos requer operações de

¹ API pode ser resumido como um conjunto de rotinas e instruções de programação estabelecidos por um software para a utilização das suas funcionalidades por programas que não querem envolver-se em detalhes da implementação do software, mas apenas usar seus serviços.

² Representam uma estrutura formada por blocos pré-fabricados de software que os programadores podem usar, estender ou adaptar para uma solução específica (FAYAD, M.E.; SCHMIDT, D.C. (1997). Object-oriented Application frameworks. **Communications of the ACM**, v.40, n.10, p.10.).

cooperação entre cada processo de forma que operação de envio deve casar com uma operação de recebimento.

Esse modelo considera que um grupo de tarefas é criado no início da aplicação e cada uma é atribuída a um processador. As tarefas representam diferentes fluxos de execução e isso permite que o modelo de troca de mensagens seja visto como um modelo *multiple program multiple data* (MPMD), o qual claramente se relaciona com a arquitetura do tipo MIMD, já que nessa arquitetura cada processo pode ser executado por uma unidade de processamento diferente, permitindo uma execução mais rápida do programa paralelo. São exemplos de ambiente de passagem de mensagens o PVM e o MPI.

O PVM é um conjunto integrado de bibliotecas e de ferramentas de software, cuja finalidade é emular um sistema computacional concorrente heterogêneo, flexível e de propósito geral. O objetivo do PVM é permitir que um grupo de computadores interconectados, possivelmente com diferentes arquiteturas, possa trabalhar cooperativamente formando uma máquina paralela virtual. O sistema PVM permite que sejam escritas aplicações nas linguagens Fortran, C e C++.

O MPI é a especificação de uma interface para programação paralela com troca de mensagens, independente de linguagem ou protocolo de comunicação. Existem implementações dessa interface para as principais linguagens de programação como C, C++ ou Fortran. O objetivo do MPI é prover um amplo padrão para escrever programas com passagem de mensagens de forma prática, eficiente e flexível. MPI não é um padrão IEEE ou um ISO, mas chega a ser um padrão industrial para o desenvolvimento de programas com troca de mensagens. Todo paralelismo é explícito, ou seja, o programador é responsável por identificar o paralelismo e implementar o algoritmo utilizando chamadas aos comandos da biblioteca MPI.

➤ *Paralelismo de dados (Data Parallel)*

Neste modelo, o trabalho a ser realizado em paralelo consiste principalmente em operações de leitura e escrita de memória. Os dados são geralmente organizados em uma estrutura como um vetor ou matriz. Todas as tarefas operam sobre esta estrutura, acessando conjuntos disjuntos de dados evitando concorrência pelo acesso à mesma área da memória. As tarefas possuem a restrição de realizarem sempre as mesmas ações, mas em conjuntos distintos de dados.

Aplicações onde se possa realizar a decomposição de domínio de dados, ou seja, o particionamento dos dados em grupos para que sejam realizadas operações sobre eles,

encaixam-se bem nesse modelo. Um exemplo é a multiplicação de matrizes. As matrizes são divididas de modo uniforme e cada tarefa opera sobre uma porção dos dados de forma paralela.

2.2.5 Desempenho de um algoritmo paralelo

Uma vez implementado um programa paralelo é necessário que se faça uma análise do seu desempenho. Esta análise consiste na execução de procedimentos, computacionais ou não, com o objetivo de identificar possíveis gargalos de execução e/ou partes do programa que podem ter seu desempenho melhorado.

Para avaliar o desempenho empírico de um algoritmo paralelo podemos usar as medidas de aceleração (*speedup*) e eficiência. A medida mais comum é o *speedup*, que indica o número de vezes que o programa paralelo é mais rápido que o seqüencial. A medida do *speedup* é:

$$Speedup = \frac{T_s}{T_p}, \quad (2.1)$$

onde T_s é o tempo de execução do programa seqüencial e T_p é o tempo de execução do programa paralelo.

O *speedup* geralmente é um valor entre 0 e p , onde p representa a quantidade de processadores. Na prática, dificilmente o valor máximo é obtido por causa de sobrecargas relacionadas à comunicação e sincronização. A lei de Amdahl (AMDAHL, 1967), mostrava que o ganho de velocidade não era infinito, mesmo quando a porção serial fosse quase inexistente e o número de processadores em paralelo tendesse ao infinito. A fórmula apresentada por Amdahl é a seguinte

$$speedup_{Amdahl} = \frac{1}{f + \frac{(1-f)}{p}} \quad (2.2)$$

onde:

- p é a quantidade de processadores;
- f é a porcentagem de código que é executada seqüencialmente.

O resultado dessa formulação foi considerado um desestímulo ao paralelismo, devido ao fato de que o ganho com a utilização da paralelização sempre apresentava valores inexpressivos. A formulação de Amdahl só pode ser considerada válida se o tamanho do problema não variar com o número de processadores (o que não ocorre em problemas reais) e por este motivo é classificado como *speedup* de carga fixa ou tamanho fixo (GUSTAFSON, 1988).

Na tentativa de melhor traduzir o desempenho do programa, outras medidas são formuladas a partir do *speedup*. Uma destas medidas é a eficiência. A eficiência de um programa paralelo indica o quanto o *speedup* obtido está próximo ou não do ótimo (MARCARI JUNIOR, 2002).

$$Eficiência = \frac{speedup}{p}, \quad (2.3)$$

onde p é o número de processadores utilizados pelo programa paralelo.

2.3 Ferramentas para programação paralela em arquitetura de memória compartilhada

A seguir serão apresentadas as principais ferramentas para programação paralela em arquiteturas de memória compartilhada tanto para processamento em CPU como para processamento em GPU. São descritos as ferramentas utilizadas como POSIX Thread para CPU e NVIDIA CUDA para GPU que foram utilizadas para acelerar o problema térmico inverso e as ferramentas que podem ser utilizadas para trabalhos futuros como o OpenMP para CPU e o OpenCL para CPU e GPU.

2.3.1 POSIX Threads

A biblioteca POSIX Threads é um padrão IEEE POSIX (*Portable Operating System, Interface for Unix*) para *thread*, o qual define uma API padrão para criar e manipular *threads* nas linguagens C/C++. As bibliotecas que as implementam são chamadas também de Pthreads, sendo muito difundidas no universo Unix e outros sistemas operacionais semelhantes como Linux e Solares, mas também existem implementações para Windows (BARNEY, 2010c; BUTENHOF, 1997). Esta biblioteca nos fornece maneiras eficientes de expandir o processo em execução em novos processos concorrentes, que podem executar com

maior eficiência em sistemas computacionais com múltiplos processadores e/ou processadores com múltiplos núcleos.

Em geral, para um programa tirar proveito do Pthreads, ele deve ser capaz de ser organizado em tarefas independentes que possam ser executadas simultaneamente, como representado na Figura 2.10. Por exemplo, se a rotina 1 e a rotina 2 podem ser permutadas, intercaladas e/ou sobrepostas em tempo real elas são candidatas para utilizarem o Pthreads e executarem em um tempo computacional menor.

Figura 2.10 – Execução de rotinas pelo tempo (BARNEY, 2010c).

Pthreads são definidos como um conjunto de tipos da linguagem C e chamadas de procedimentos, implementados com um cabeçalho (*header/include*) `pthread.h`. Existem cerca de 100 procedimentos em Pthreads, todos com o prefixo “`pthread_`”. As operações com *threads* incluem a criação, finalização, sincronização, agendamento e sinalização. Todas as *threads* tem acesso à mesma memória global compartilhada, sendo o programador responsável pela sincronização (proteção) do acesso aos dados globalmente compartilhados. Devido ao fato de múltiplas *threads* poderem acessar concorrentemente as mesmas variáveis, podem ocorrer casos onde múltiplas *threads* acessem e alterem o mesmo dado ao mesmo tempo causando “condições de corrida”³, como pode ser observado na Figura 2.11.

³ Quando duas *threads* modificam uma variável ao mesmo tempo, ocorre uma condição de corrida; só que nessa corrida, a *thread* que modificar o valor por último é que tem seu valor armazenado.

Figura 2.11 – Condição de corrida (BARNEY, 2010c).

Um exemplo de condição de corrida é quando duas *threads* tentam realizar, ao mesmo tempo, a incrementação de uma variável. Pode ser que uma execute apenas depois de a outra acabar de incrementar a variável, como pode ser que as duas comecem a incrementá-la ao mesmo tempo, ocasionando um resultado incorreto e causando inconsistência nos dados.

Uma alternativa para evitar a ocorrência de condições de corrida é a utilização de fechaduras (*locks*) e semáforos. As fechaduras são utilizadas para garantir o acesso de uma única *thread* a determinada porção do código. Elas são utilizadas normalmente para proteger zonas críticas⁴. Quando uma *thread* atinge uma fechadura, ela confere se está trancada, se não estiver, ela a tranca, e continua a executar o código. Todas as *threads* que chegarem à fechadura após isso esperarão que a fechadura seja destrancada, que ocorre quando a primeira *thread* atinge o comando *unlock*. Quando isto ocorre, apenas umas das *threads* que estavam esperando consegue continuar, enquanto as outras esperam. Os semáforos são semelhantes às fechaduras, impõem uma condição para que determinada sessão do código seja acessada. Por exemplo, uma determinada operação só poderá ocorrer se uma determinada variável satisfizer a uma determinada condição. Se isso ocorre, o acesso é liberado para essa operação, senão, todas as *threads* que chegarem ao semáforo esperam que a condição seja atingida.

A seguir serão citadas algumas funções usadas para criar, destruir e sincronizar *threads* utilizando a biblioteca Pthreads. Para uma explanação mais detalhada sugere-se Barney (2010c), Butenhof (1997) e Lewis e Berg (1996).

⁴ Parte do código que tem comportamento indeterminado caso seja executada por mais de uma *thread* ao mesmo tempo.

- *pthread_create*: primitiva para criar uma *thread* que executa a função por ela especificada;
- *pthread_join*: primitiva que faz com que o processo fique em espera até que a *thread* identificada termine a execução;
- *pthread_exit*: primitiva que destrói uma *thread*;
- *pthread_mutex_lock*: inicializa um mutex, que implementa o paradigma de fechadura, explicado anteriormente;
- *pthread_mutex_unlock*: destranca um mutex;
- *pthread_mutex_destroy*: destrói um mutex, desalocando a memória que ele gasta;
- *pthread_cond_init*: inicializa a estrutura de condição de *threads*;
- *pthread_cond_wait*: essa função faz com que a *thread* que a chame destrave ‘mutex’ e espere até uma condição ser inicializada para continuar sua execução;
- *pthread_cond_destroy*: desaloca a memória gasta por uma estrutura de condição inicializada;
- *pthread_cond_signal*: envia um sinal identificado por uma condição para uma *thread* bloqueada;
- *pthread_cond_broadcast*: envia um sinal identificado por uma condição para todas as *threads* bloqueadas.

A seguir apresenta-se um exemplo de um programa que mostra duas *threads* executando ao mesmo tempo, realizando cada uma o seu trabalho (WIKILIVROS, 2010).

Algoritmo 1: Exemplo de utilização de Pthreads em C

```
01. #include <pthread.h>
02. #include <stdlib.h>
03. #include <stdio.h>
04.
05. type struct
06. {
07.     int id;
08. } thread_arg;
09.
10. int main()
11. {
12.     pthread_t tid[2];
13.     thread_arg a[2];
14.     int i = 0; int n_threads = 2;
15.
16.     //Cria as threads
17.     for(i = 0; i < n_threads; i++)
18.     {
19.         a[i].id = i;
```

```

20.     pthread_create(&(tid[i]), NULL, thread, (void *)&(a[i]));
21. }
22. //Espera que as threads terminem
23. for(i = 0; i < n_threads; i++)
24. {
25.     pthread_join(tid[i], NULL);
26. }
27. pthread_exit((void *)NULL);
28. }
29.
30. void *thread(void *vargp)
31. {
32.     int i = 0;
33.     thread_arg *a = (thread_arg *) varg;
34.     //Faz um trabalho qualquer
35.     for(i = 0; i < 10000; i++) printf("Terminou a thread %d\n", a->id);
36.     pthread_exit((void *)NULL);
37. }

```

No exemplo pode-se compreender a execução de múltiplas *threads*, o programa começa com a *thread* principal, que cria outras *threads* (`pthread_create`) e espera que elas terminem seu trabalho (`pthread_join`). Cada uma das *threads* realiza um trabalho, ao mesmo tempo. Após elas retornam e a *thread* principal termina a execução do programa serial. Na Figura 2.12 observa-se o possível fluxo de execução do exemplo.

Figura 2.12 – Possível fluxo de execução.

Algumas das principais vantagens da utilização de Pthreads são:

- Utiliza melhor o potencial dos novos processadores *multi-core* que estão ficando cada vez mais comuns atualmente;
- Portabilidade das aplicações escritas utilizando a biblioteca Pthreads para qualquer sistema operacional que suporte o padrão POSIX;
- O preço da troca de contextos entre *threads* é menor do que com processos, devido ao fato de as *threads* serem mais leves;

- Redução do *overhead*⁵ do sistema;
- Maior eficiência na exploração do paralelismo da aplicação.

Algumas das suas desvantagens são:

- Converter programas prontos para programas utilizando *threads* não é trivial;
- Complexidade do programa aumenta devido a utilização de diversos recursos para gerenciamento e sincronização entre as *threads*;
- Ocorrência de condições de corrida e *deadlocks*;
- Ocorrência de inversão de prioridade, quando uma *thread* de prioridade mais alta não executa devido a dependências não resolvidas com outras *threads* de prioridade mais baixas.

Atualmente, todo tipo de aplicação requer uma versão paralelizada, pois a tendência é o uso de processadores *multi-core*. O padrão POSIX Threads é uma opção que permite o uso maximizado desses processadores, sendo que não existe, portanto um tipo específico de aplicação para Pthreads. Devido a essas vantagens optou-se em utilizar Pthreads para melhorar o tempo computacional do problema térmico inverso em questão.

2.3.2 OpenMP

O OpenMP (*Open Multi-Processing* ou Multi-processamento aberto) é uma API multi-plataforma para processamento paralelo baseado em memória compartilhada para as linguagens C/C++ e Fortran, em plataformas Unix e Microsoft Windows. Ela consiste de um conjunto de diretivas para o compilador, funções de biblioteca e variáveis de ambiente que influenciam na execução do programa.

O OpenMP foi desenvolvido por um grupo de fabricantes de hardware e software que buscavam uma interface de utilização simples e escalável, a qual pudesse ser utilizada tanto para aplicações de grande porte, quanto para aplicações simples. Ele surgiu em 1997 para Fortran e no ano seguinte foi lançada a primeira versão para C/C++. Atualmente está na versão 3.0 que saiu em 2008 (FRIEDMAN, 2010).

O modelo de execução do OpenMP inicia-se com uma única *thread*, ou seja com um fluxo de execução principal (*master thread*) que executa até encontrar uma região paralela a

⁵ Geralmente considerado qualquer processamento ou armazenamento em excesso, seja de tempo de computação, de memória, de largura de banda ou qualquer outro recurso que seja requerido para ser utilizado ou gasto para executar uma determinada tarefa, piorando o seu desempenho.

qual deve ser executada por múltiplas *threads*. Neste ponto ela cria outras *threads*, e assim todas as *threads* inclusive a mestre, executam o código dentro dessa região. Quando as *threads* completam a execução do código na região paralela elas se sincronizam, que ocorre quase sempre de maneira implícita, com exceção da *thread* a qual é responsável por continuar a execução do programa até que ela encontre uma nova região paralela ou até que o programa termine. Este modelo de execução é conhecido como *fork-join* como pode ser observado na Figura 2.13.

Figura 2.13 – Modelo de execução OpenMP (fork/join) (BARNEY, 2010b).

No OpenMP não é possível ver como cada *thread* é criada e inicializada e também não é visível uma função separada contendo o código que cada *thread* executa. A divisão de trabalho realizado sobre um arranjo também não é visível explicitamente.

A criação de *threads* no programa em OpenMP é responsabilidade do compilador. O programador é responsável por definir as regiões paralelas através de diretivas, ou seja, uma linha de código com significado especial para o compilador, no caso do OpenMP C/C++ elas são identificadas por `#pragma omp`. O OpenMP oferece duas diretivas para a definição de regiões paralelas. Uma delas é o `#pragma omp parallel`, na qual *threads* são disparadas conforme necessário. A outra diretiva é para regiões contendo laços *for* identificada por `#pragma omp parallel for`.

A seguir será apresentado um algoritmo que ilustra a utilização da diretiva `parallel` que realiza a soma dos elementos de um vetor.

Algoritmo 2: Paralelização de regiões – diretiva *parallel*

```
01. #include <omp.h>
    .
    .
    .
18. qtde_elem = tamanho_vector(v)/num_threads;
19.
20. omp_set_num_threads (num_threads);
21.
22. #programa omp parallel private(inicio, fim, meu_rank)
```

```
23. {
24.     meu_rank = omp_get_thread_num();
25.     inicio = qtde_elem * meu_rank;
26.     fim = inicio + qtde_elem;
27.
28.     for (i = inicio; i < fim; i++)
29.         soma_parc[meu_rank] = soma_parc[meu_rank] + v[i];
30. }
31.
32. for (i = 0; i < num_threads; i++)
33.     soma_total = soma_total + soma_parc[i];
34.
35. printf("Resultado da soma: %d", soma_total);
36. ...
```

No código anterior, a diretiva `#pragma omp parallel` (linha 22) define uma região paralela que corresponde da linha 24 a 29. O número de *threads* que executam esta região é definido na linha 20, com a diretiva `omp_set_num_threads`, como sendo igual a `num_threads`. A quantidade de elementos de `v` que cada *thread* irá somar é definida na linha 18 em `qtde_elem`⁶. O primeiro e o último elemento, ou seja, o pedaço do vetor atribuído a cada *thread*, é calculado com base no seu *rank*⁷, que é obtido através da função `omp_get_thread_num`, como pode ser verificado nas linhas 24 a 26. A soma dos elementos é efetivamente realizada na linha 29 em um laço *for* que é executado por todas as *threads*. Dentro deste laço utiliza-se um vetor, `soma_parc`, onde cada *thread* armazena, na posição correspondente ao seu *rank*, o resultado da soma do seu pedaço do vetor. No fim região paralela todas as *threads* são destruídas, com exceção da mestre que é responsável por executar o resto do programa (parte serial). Então a mestre computa o resultado da soma (linha 32) utilizando os valores em `soma_parc`.

Já a utilização da diretiva `parallel for` divide automaticamente as iterações de um laço *for* entre todas as *threads*, não sendo necessário a utilização das variáveis `meu_rank`, `inicio` e `fim` para definir os diferentes “pedaços” do vetor para diferentes *threads*. Embora mais complexa, a diretiva `parallel` pode ser utilizada para paralelização de qualquer natureza.

⁶ Por simplificação do código, assume-se que o número de elementos do vetor seja múltiplo do número de *threads*.

⁷ Cada *thread* dentro de uma região paralela é identificado por um *rank*. Em uma região com *t threads* os *ranks* variam de 0 a *t-1* (o *rank* 0 sempre identifica a *thread mestre*).

Algoritmo 3: Paralelização de laços – diretiva *for*

```
.      // parte sequencial do programa
.
56. #pragma omp parallel for
57. {
58.     for(i = 0; i < n; i++)
59.         soma[i] = soma[i] + v[i];
60. }
```

Como a maior parte das variáveis no código OpenMP são visíveis a todas as *threads* por padrão, é necessário definir variáveis individuais para evitar condições de corrida, e existe a necessidade de passar valores entre a parte sequencial e a região paralela. Para isto existem as cláusulas, que consistem em palavras reservadas que são utilizadas junto com as diretivas geralmente para modificar o comportamento padrão destas. O OpenMP especifica várias cláusulas que podem ser classificadas em características de partilha de dados; sincronização; programação; comando *if*; inicialização; cópia de dados; Redução e outras (CHAPMAN; JOST; PASS, 2008).

O OpenMP foi implementado em vários compiladores comerciais, como por exemplo o Visual C++ e o *Intel Parallel Studio*. Os compiladores e ferramentas da Sun Studio suportam as últimas especificações com ressalva na produtividade para as plataformas Solaris SO e Linux. Os compiladores Fortran, C e C++ do *Portland Group* também suportam a versão OpenMP 2.5. O GCC tem suporte desde a versão 4.2 do compilador.

Espera-se obter um *speedup* de N vezes no tempo total de execução do programa paralelizado utilizando OpenMP em um processador com N núcleos. No entanto, isso raramente ocorre, devido às seguintes razões: uma grande parte do programa não pode ser paralelizada por OpenMP; a largura de banda da memória não escala até N vezes.

As principais vantagens da utilização do OpenMP são:

- Simplicidade, pois o OpenMP cuida da maioria das coisas para o usuário e a distribuição de tarefas é feita automaticamente pela implementação;
- Paralelizar o código já existente é simples, requer poucas modificações e o paralelismo pode ser implementado de forma incremental, não exigindo grandes modificações estruturais no código.

As desvantagens são:

- Só executa de maneira eficiente em arquiteturas de memórias compartilhada;

- Requer suporte do compilador;
- Escalabilidade é limitada pela arquitetura da memória;
- Tratamento de erros é difícil;
- Falta de controle mais “fino” sobre as *threads*.

Comparando OpenMP e Pthreads fica claro a sua simplicidade, no entanto, é possível obter uma melhor performance com Pthreads, caso otimize bastante o código, pois ele permite um controle mais fino das *threads*.

2.3.3 NVIDIA CUDA

Até há pouco tempo, aceder ao poder computacional da GPU de forma eficiente era difícil, pois o GPU apenas podia ser programado através de APIs gráficas como o OpenGL e DirectX. A arquitetura e o modelo de programação iniciais destas placas não apresentavam propriedades que permitissem a sua utilização para computação de problemas genéricos devido a fatores como: a dificuldade em aprender uma linguagem nova, restrições de uso quanto ao acesso à memória DRAM das GPUs e limitação na largura de banda nestes acessos.

Em 2002 Mark Harris (HARRIS, 2002, 2003) introduziu o termo Processamento Gráfico de Propósito Geral (GPGPU – *General Purpose Graphics Processing Unit*) que consistia em uma técnica para rodar aplicações comumente processadas pela CPU. Tempos depois outras linguagens de programação para GPUs foram desenvolvidas, tais como a PyGPU (LEJDFORS; OHLSSON, 2006) e Brooks for GPUs (BUCK et al., 2004). Estas novas linguagens aliadas a evolução gradual das GPUs tornaram a tarefa de programá-las mais acessível aos pesquisadores de diversas áreas do conhecimento. Além disso, os desenvolvedores de algoritmos paralelos encontraram nas GPUs, devido às suas estruturas de dezenas e até centenas de núcleos de processamento, máquinas nas quais podiam testar suas teorias .

Entretanto, foi em 2007 que o processamento geral em GPU ganhou grande destaque, com o lançamento da plataforma de software para computação massivamente paralela de alto desempenho desenvolvida pela NVIDIA para suas GPUs (a partir da série NVIDIA GeForce 8, Tesla e Quadro) denominado CUDA (*Compute Unified Device Architecture*).

A grande novidade é a maneira eficiente que o CUDA juntamente com a arquitetura da GPU, possibilita o desenvolvimento de aplicações que podem explorar ao máximo o paralelismo de dados. Como foi observado na Figura 2.2 e é representado na Figura 2.14 com

respeito a largura de banda da memória (*memory bandwidth*⁸), as GPUs da NVIDIA apresentaram uma grande evolução, quando comparados com os processadores de propósito geral do mercado.

Figura 2.14 – Evolução da *Bandwidth* das GPUs e CPUs (NVIDIA, 2010).

O principal motivo desse enorme ganho de desempenho é que as GPUs são dedicadas ao processamento de dados, não tendo preocupação em guardar grandes quantidades de informações em memórias *cache* nem em tratar de um controle de fluxo altamente complexo.

A memória *cache* tem como objetivo obter uma velocidade de acesso à memória próxima da velocidade das memórias mais rápidas. Elas possuem uma cópia de partes da memória principal, assim, quando o processador desejar ler uma palavra da memória, uma verificação é efetuada para determinar se a palavra está na memória *cache*, otimizando o acesso. Caso ela esteja é imediatamente fornecida ao processador. Caso contrário, um bloco de dados da memória principal é lido para a memória *cache* e em seguida a palavra requerida é entregue ao processador.

A Figura 2.15 exibe uma comparação entre a destinação dos transistores em CPUs e GPUs.

⁸ Taxa na qual os dados podem ser lidos ou armazenados na memória por um processador.

Figura 2.15 – Destinação dos transistores em CPUs e GPUs (NVIDIA, 2008).

Estas pequenas *caches* na GPU são também chamadas de memórias *on-chip* compartilhadas e possuem acesso rápido para leitura e escrita, permitindo o compartilhamento de tarefas entre *threads* (Figura 2.16). Essas memórias são parte da tecnologia de hierarquia de memória que diminui o acesso às memórias externas e, portanto reduz o tempo de execução dos aplicativos, como será explicado com mais detalhes adiante.

Figura 2.16 – Memória compartilhada (NVIDIA, 2007b).

O CUDA possibilitou também o acesso à memória de maneira a suportar operações antes somente suportadas por CPUs, como as operações de *gather* e *scatter* (Figura 2.17) com a memória DRAM da GPU. Os modelos mais antigos de placas suportavam apenas a operação de *gather*.

Figura 2.17 – Operações de *gather* e *scatter* na memória da GPU (NVIDIA, 2007b).

O CUDA assenta um modelo de programação paralelo e escalável que permite misturar programação sequencial com programação paralela de uma forma heterogênea, encarando o GPU (*device*) e o CPU (*host*) como unidades de processamentos distintas e com memórias diferentes, onde apenas o código que envolve as diretivas especiais introduzidas é executado na GPU, o resto é executado normalmente pela CPU.

O uso dessas diretivas ou prefixos são extensões mínimas ao ambiente de programação C que permitem especificar qual zona de memória a que determinada estrutura, variável ou função pertence. Esses prefixos no código permitem instruir o compilador de CUDA, o *nvcc*, acerca do tipo de código a gerar, como se segue:

- Código CPU, que posteriormente será compilado pelo *gcc/cl*.
- Código GPU, que posteriormente será compilado pelo *cudacc*.

A Figura 2.18 ilustra a pilha de software da plataforma CUDA, composta por um conjunto de marcadores e diretivas especiais para linguagem C que permitem que o compilador reconheça se o código em questão é para GPU ou CPU, uma biblioteca para tempo de execução (*runtime*) que é dividida em componentes que gerenciam operações do *host* (principalmente a comunicação deste com o *device*), do *device* (as funções específicas suportadas pela GPU e de duas bibliotecas matemáticas prontas para uso o CUBLAS (*Basic Linear Algebra Subroutines*) e o CUFFT (*Fast Fourier Transform*)) e de componentes comuns, como tipos adicionais e subconjunto de funções da biblioteca padrão C que podem ser utilizados tanto no *host* quanto no *device*. No topo desta pilha encontra-se a API da plataforma CUDA.

Figura 2.18 – Pilha de software da plataforma CUDA (HALFHILL, 2008).

Os trechos dos códigos em CUDA que executam na GPU são, normalmente, pequenas porções de código altamente paralelizáveis, as quais serão executados por milhares de *threads* em simultâneo. Para gerenciar as diversas *threads* a NVIDIA introduziu um novo conceito na computação paralela oriundo da taxonomia de Flynn SIMD, no entanto em vez de gerenciar dados são gerenciados *threads* recebendo o nome de SIMT (*Single Instruction Multiple Thread*). Essa arquitetura cria, gerencia, agenda e executa *threads* automaticamente em grupos de 32 *threads* paralelas, o que a NVIDIA denomina de *warp*. Caso um multiprocessador receba um bloco com mais de 32 *threads* para processar, ele quebra esse bloco em *warps*, agrupando as *threads* de acordo com o *thread ID*.

A noção de *warp* é importante quando se pretende obter um bom desempenho em tempo de execução, pois se pode evitar divergências no caminho de execução de *threads* de um mesmo programa paralelo. Essas divergências surgem, por exemplo, em uma instrução *if...then...else*. Se houver divergências entre *threads* de *warps* diferentes, não há perdas de performance, no entanto, se dentro de um mesmo *warp* de *threads* houver divergências entre *threads*, então todo o *warp* de *threads* executa todo o *if...then...else*, havendo *threads* no *warp* que nada fazem no corpo do *then* e outros que nada fazem no corpo do *else*, fenômeno que resulta em perda de desempenho.

Uma GPU (*device*) consiste em um vetor de multiprocessadores de *threads* ou *streaming multiprocessor* (SM), conforme pode ser observado na Figura 2.19.

Figura 2.19 – Arquitetura da GPU (NVIDIA, 2008).

Cada multiprocessador (ou SM) é constituído de oito processadores escalares (SP – *scalar processor*) e quatro tipos de memórias *on-chip*, como mostrado na Figura 2.19:

- Memória compartilhada (*shared memory*): com 16 Kb de tamanho e acesso ultra-rápido, compartilhada entre todas as *threads* de um bloco;
- Cache de textura (*Texture cache*): permitido somente leitura, tem por função otimizar o acesso à memória de texturas;
- Cache constante: com 64 Kb, permitido somente leitura, trabalha em cima do espaço de memória constante, aumentando a velocidade de leitura;
- Registradores: tipo de memória com pouca capacidade (32 bits), porém muito rápida, utilizada no armazenamento temporário durante o processamento. Presente em cada processador.

Adicionalmente, é possível acessar a memória principal da GPU (*device*) chamada de *device memory*, onde reside a memória local de cada *thread* e a memória global da aplicação. É a memória com maior latência de acesso dentre as memórias da GPU.

A memória compartilhada (*shared memory*) é dividida em 16 bancos de 1 Kb cada, esta divisão tem por objetivo tirar proveito do paralelismo dos processadores de cada SM. Assim, cada banco pode servir um endereço a um SP por cada *clock*, permitindo tantos acessos simultâneos por *clock* quanto o número total de bancos. No entanto, se existirem múltiplos acessos simultâneos a um mesmo banco de memória, gerem-se conflitos de bancos. Estes acessos em conflito são serializados, perdendo o paralelismo. Deste modo, quanto maior

o número de conflitos no acesso aos bancos de memória partilhada, maior a perda de desempenho, resultando perdas de paralelismo no manuseio da memória partilhada.

A NVIDIA disponibiliza um guia (NVIDIA, 2009b) no qual descreve como programar a placa para evitar problemas como descrito acima, a fim de obter o melhor desempenho possível no processamento dos dados pela placa gráfica. Algumas estratégias são:

- Maximização da execução em paralelo: através de uma estruturação do algoritmo, escolha apropriada da configuração de execução do *kernel*⁹ como número de *threads* por bloco múltiplo do tamanho do *warp*, mínimo de 64 *threads* por bloco e evitar divergência dentro de um mesmo *warp*.
- Otimização do uso de memória: Minimização de transferência de dados *host/device*; acesso coalescido à memória global; usar o máximo possível a memória compartilhada com acessos a ela sem conflitos.

O NVIDIA CUDA introduz dois novos conceitos para o escalonamento das *threads*, denominados bloco e *grid*, que são responsáveis para organizar a repartição dos dados entre as *threads* e sua distribuição no hardware.

O bloco é a unidade básica de organização das *threads* e de mapeamento para o hardware. Um bloco de *threads* é alocado a um multiprocessador da GPU, e é recomendado que ele contenha no mínimo oito *threads*, caso contrário haverão processadores ociosos, devido ao fato que existem oito processadores em cada multiprocessador (NVIDIA, 2007b). Os blocos podem ter uma, duas, ou três dimensões (*blockIdx.x*, *blockIdx.y* ou *blockIdx.z*). O número máximo de *threads* possíveis de executar em um bloco de *threads* CUDA é 512.

O *grid* é a unidade básica onde estão distribuídos os blocos. O *grid* é a estrutura completa de distribuição das *threads* que executam uma função. É nele que está definido o número total de blocos e de *threads* que serão criados e gerenciados pela GPU para uma dada função. Um *grid* pode ter uma ou duas dimensões.

Na Figura 2.20 observa-se a representação de duas invocações (*kernels*) da GPU (device) pelo *host* onde cada *kernel* é executado como um grupo de *threads* organizados como um *grid* de blocos de *threads*.

⁹ Programas que executam as *threads* da GPU.

Figura 2.20 – Organização de um *grid* com seus blocos de um *kernel* (NVIDIA, 2007b).

A API da plataforma CUDA introduz quatro extensões à linguagem C, que podem ser classificados como:

- Qualificadores de tipo de função, para definir a unidade lógica de execução do código: CPU ou GPU;
- Qualificadores de tipo de variável, para definir onde elas serão armazenadas: CPU ou GPU;
- Nova sintaxe de chamada de função para configurar os blocos e o *grid*;
- Quatro variáveis internas para acessar os índices e dimensões dos blocos, do *grid* e das *threads*.

Os qualificadores de tipo de função encontrados em CUDA são: `__device__` (define uma função que será executada na GPU, sendo que só pode ser chamada a partir da própria GPU), `__global__` (define o que a plataforma CUDA chama de *kernel*, ou seja, uma função que é executada na GPU e é chamada a partir da CPU), e o `__host__` (define uma função que será executada na CPU e que ela só pode ser chamada a partir da CPU).

São três os qualificadores de tipo de variáveis: `__device__` (define uma variável que reside na memória global da GPU, as quais são acessíveis por todas as *threads* de um *grid* e também a partir da CPU através do uso da variável *runtime* do CUDA, seu tempo de vida é o da aplicação), `__constant__` (difere apenas no fato de que a variável reside no espaço da

memória constante da GPU) e `__shared__` (define as variáveis que residem na memória partilhada da GPU e são acessíveis apenas pelas *threads* de um mesmo bloco, seu tempo de vida é o do bloco).

Para realizar uma chamada de função na GPU é preciso informar no código as dimensões do *grid* e do bloco. Isto é feito através de uma nova sintaxe na chamada da função. Utiliza-se, entre o nome da função e os argumentos passados a ela, um *array* bidimensional onde constam as dimensões do *grid* e do bloco, respectivamente. Esse *array* é delimitado pelos caracteres `<<<` e `>>>`.

O CUDA fornece uma maneira de acessar os índices e dimensões das *threads*, blocos e *grids*. Para acessar o valor dos índices das *threads* utiliza-se a variável `threadIdx`, a qual é um vetor com até três dimensões. Cada bloco dentro do *grid* fornece seu índice identificador, como em `blockIdx.x` e `blockIdx.y`. Além disso, os índices das dimensões dos blocos de *threads* são acessíveis através da variável `blockDim`. Por fim, é possível acessar os valores das dimensões do *grid* através da variável `gridDim`.

O CUDA introduz ainda um conjunto de funcionalidades que se assemelham as chamadas de funções, mas que são funcionalidades intrínsecas ao próprio hardware. Um bom exemplo é a chamada `__syncthreads()` que invoca sincronização das *threads* por hardware do próprio GPU – *barrier synchronization* – ao invés de cada multiprocessador. Segundo o fabricante com esta função são eliminados os *deadlocks*¹⁰.

Na Figura 2.21 ilustra-se o fluxo de processamento de um programa escrito em CUDA. Basicamente, o programa inicia copiando os dados a serem processados da memória RAM para a memória principal da GPU. Em seguida, configura-se um *grid* e é realizada a chamada do *kernel* para ser executado na GPU. Por fim, os dados resultantes são copiados para a memória RAM para posterior processamento pela CPU.

¹⁰ Ocorrem por exemplo, quando determinada *thread* espera por um resultado de outra *thread*, e vice-versa. Como uma *thread* está esperando a outra, nenhuma segue em frente para fornecer o resultado para outra, e o programa fica parado.

Figura 2.21 – Fluxo de execução de um programa CUDA.

Desta forma, como já mencionado anteriormente o ciclo de execução de uma aplicação utilizando a tecnologia CUDA alterna entre execuções na CPU e na GPU. Na Figura 2.22 representamos esta execução, na qual se inicia com uma *thread* mestre (*master thread*) no *host*, em seguida é invocado a execução no *device* onde são criadas varias *threads* para executar a rotina paralelamente, observamos a organização das *threads* em blocos e os blocos em um *grid*. Após a execução do *kernel* ou é chamado outro *kernel* para execução de outra rotina em paralelo no *device* ou a *master thread* termina a execução do programa no *host*.

Figura 2.22 – Caminho de execução de uma aplicação CUDA (NVIDIA, 2008).

Um aspecto importante das placas da NVIDIA foi sua divergência com relação à norma IEEE-754, a qual define os formatos adequados para representar números em pontos flutuantes de precisão simples (32 bits) e de precisão dupla (64 bits). Nas placas da série 8 e 9, por exemplo, em operações de multiplicação seguidas de adição frequentemente são combinadas em uma única instrução de hardware denominada de FMAD. Esta instrução trunca os valores intermediários das multiplicações, o que pode resultar em perda de precisão. Para suprir esta perda de precisão o CUDA fornece implementações via software para realizar essas operações de forma a seguir o padrão IEEE-754, no entanto ocorre uma perda de desempenho nessas implementações.

Em novembro de 2006, a NVIDIA introduziu no mercado sua primeira GPU dedicada à computação de propósito geral destinada exclusivamente ao processamento de dados, não tendo como característica a utilização como placa de vídeo, denominada Tesla (NVIDIA, 2007c). A partir desta série dedicada a NVIDIA começou a suprir a limitação do hardware perante o padrão IEEE. Inicialmente introduziu suporte para o padrão IEEE 754-1985

somente para precisão simples e posteriormente para versões mais recentes de Tesla introduziu o suporte para precisão dupla seguindo o padrão IEEE 754-1985.

As placas da última geração da NVIDIA, denominadas Fermi (NVIDIA, 2009a) como Tesla C2050/C2070 já adotam o padrão IEEE 754-2008 tanto para precisão simples como dupla. Segundo a fabricante nesta nova geração as placas não apresentam perda de desempenho na utilização de precisão dupla, e uma novidade interessante é a possibilidade de executar *kernels* em paralelos (paralelizar a execução de códigos em paralelo) como representado pela Figura 2.23 e outras novidades que a tornam muito atraentes para área científica.

Figura 2.23 – Execução simultânea do kernel (NVIDIA, 2009a).

2.3.4 OpenCL

O OpenCL (Open Computing Language) (KHROS, 2010) é um *framework* que permite escrever programas que executam em plataformas heterogêneas como CPUs, GPUs, e outros processadores de múltiplos núcleos interligando-se de maneira integrada. O OpenCL foi inicialmente desenvolvido pela Apple Inc. em colaboração com as fabricantes de processadores AMD, Intel e NVIDIA. Em 2008 foi formado um consórcio gerenciado pelo *Khronos Compute Working Group* com representantes de companhias de fabricantes de CPU, GPU e software. O OpenCL inclui uma linguagem baseada em C para escrever núcleos (funções que executam em dispositivos OpenCL), mais APIs que são usadas para definir e controlar a plataforma heterogênea como pode ser observado na Figura 2.24, permitindo a programação paralela usando ambos paralelismo de tarefas e de dados.

Figura 2.24 – OpenCL: plataforma heterogênea CPU e GPU (KHRONOS, 2010).

O modelo de programação OpenCL é bastante semelhante ao modelo de programação CUDA da NVIDIA. Este padrão tem a tendência de em pouco tempo substituir a programação em CUDA, de forma que a menos do compilador, específicos de cada fabricante, os códigos para GPU serão perfeitamente intercambiáveis.

2.3.5 Comparativo entre as ferramentas de programação paralela

O objetivo de utilizar programação paralela é melhorar o desempenho da aplicação, executando-a através dos diversos núcleos e/ou processadores existentes na mesma máquina.

A seguir se faz uma avaliação das ferramentas de programação paralela apresentadas anteriormente com base nos critérios qualitativos propostos pela APSTC (*Asia Pacific Science & Technology Center*) em 2008 (KASIM et al., 2008). Os critérios são apresentados a seguir:

- Arquitetura do sistema: compreende a memória compartilhada que se refere a sistemas na qual os processadores compartilham a mesma área de memória (compartilham os mesmos endereços de acesso a memória), ou de memória distribuída que se refere aos casos em que cada nó de processamento tem seu próprio espaço de memória não compartilhado com os outros.
- Metodologia de programação: entende-se como metodologia da programação ao conjunto de normas, métodos e anotações que nos indicam a forma de programar

de maneira a utilizar os recursos de paralelismo, como por exemplo: API, nova linguagem e diretivas especiais.

- c) Gerenciamento de trabalho: o paralelismo pode ocorrer através de processos ou de *threads*. Se o programador precisa cuidar da criação e gerenciamento de tarefas diz-se que o gerenciamento de trabalho é explícito, caso contrário ele é implícito, bastando especificar a seção de código que vai rodar em paralelo.
- d) Esquema de particionamento da carga de trabalho: A carga de trabalho que será executada é dividida em pequenas porções chamadas tarefas, sendo que no caso implícito o programador precisa apenas especificar qual carga de trabalho irá ser processada em paralelo sem se preocupar em gerenciá-la. No caso explícito o programador é responsável em gerir manualmente como essa carga de trabalho será dividida.
- e) Mapeamento entre tarefa e a *thread*/processo: no critério implícito o programador não precisa especificar qual *thread*/processo é responsável pela tarefa. No caso explícito gerenciar isso é responsabilidade do programador.
- f) Sincronização: define em que sequência as *threads*/processos acessam os dados que compartilham. Na sincronização implícita não há esforço do programador, ou este é mínimo bastando especificar que naquele trecho do código ocorrerá uma sincronização. Na sincronização explícita o programador é responsável por gerenciar como se dará o acesso dos processos e *threads* nesta área compartilhada.
- g) Modelo de comunicação: foca-se no paradigma de comunicação utilizado por um modelo: espaço de memória compartilhada, troca de mensagens, partição do espaço de endereço de memória global ou espaço de memória global.

Analisando as quatro ferramentas para programação paralela em arquitetura de memória compartilhada apresentadas se obtém a seguinte tabela:

Tabela 2.1 – Ferramentas de programação em arquitetura de memória compartilhada.

Critério	Pthreads	OpenMP	CUDA	OpenCL
Execução	Thread	Thread	Thread	Thread
Metodologia de programação	API	API, Extensão	API, Extensão ao C	API
Gerenciamento de trabalho	Explícito	Implícito	Implícito	Implícito
Particionamento da carga de trabalho	Explícito	Implícito/Explícito	Explícito	Explícito
Mapeamento entre tarefa e a <i>thread</i>	Explícito	Implícito	Explícito	Explícito
Sincronização	Explícito	Implícito/Explícito	Explícito	Explícito
Modelo de comunicação	Espaço de memória compartilhado	Espaço de memória compartilhado	Espaço de memória compartilhado	Espaço de memória compartilhado

A Tabela 2.1 tem o intuito de facilitar a comparação entre as ferramentas para programação em arquiteturas de memória compartilhada, por este motivo as comparações são feitas abordando o modo mais simplista de programação entre as ferramentas de programação, pois dependendo a ferramenta é possível utilizá-la tendo maior controle sobre as ações.

CAPÍTULO 3 PROBLEMA ABORDADO E SUA MODELAGEM

Neste capítulo será apresentada uma descrição do processo de tomografia térmica e a formulação do problema, juntamente com as suas vantagens e dificuldades geradas na solução de problemas inversos de convecção de calor.

3.1 Tomografia Térmica

A tomografia de processos industriais é uma técnica de determinação de variáveis operacionais através de medidas externas não intrusivas. O princípio de funcionamento da tomografia térmica geralmente se baseia na aplicação de um fluxo de calor na superfície externa do processo e na medição de temperaturas resultantes nessa mesma superfície. A técnica é robusta, econômica e, o mais importante, tem uma ótima resposta com medidas experimentais simples (CAMPOS, 2004) e pode ser aplicada na fabricação de peças fundidas no mapeamento de imperfeições (incrustações, trincas, vazios, etc.). Outros exemplos de aplicações, as principais vantagens e desvantagens da utilização da tomografia térmica podem ser observadas em Brandi (2010). Muitas destas dificuldades surgem da não linearidade das variáveis e do mau condicionamento intrínseco, apresentando-se extremamente sensível à presença de erros, tanto de origem experimental nos dados de entrada quanto de truncamento nas operações numéricas dos procedimentos computacionais.

Genericamente, mapear o interior de um objeto a partir de sinais periféricos com auxílio de técnicas tomográficas envolve resolver um problema inverso. Problemas inversos são característicos de um grande número de problemas matematicamente mal condicionados das ciências físicas, médicas e outras. Este mau condicionamento pode sob certas condições afetar a qualidade das soluções, principalmente com relação à visualização tomográfica de escoamentos multifásicos (SELEGHIM JR; MILIOLI, 2001).

Devido a esta instabilidade, a solução dos problemas inversos deixa de ser simples e passa a ser complexa, motivo este de constantes estudos na área. Os problemas inversos

constituem uma área multidisciplinar que une a análise matemática dos problemas aos dados experimentais.

Apesar da abordagem em termos de estímulos e respostas ser simples, problemas inversos, matematicamente pertencem à classe de problemas mal postos. Segundo Hadamard (1902) um problema é bem posto se as três condições forem satisfeitas:

- i. Existência – o problema deverá apresentar solução;
- ii. Unicidade – a solução deverá ser única;
- iii. Estabilidade – a solução deverá exibir dependência contínua e suave em relação aos dados de entrada;

Durante muito tempo admitiu-se que estas condições eram indispensáveis para a solução matemática de um problema físico. Os problemas que não satisfaziam estas condições eram ditos mal postos ou impropriamente-postos e considerados irrelevantes para fins práticos.

Dentre as três condições o atributo da estabilidade é tido como o mais importante, já que é praticamente impossível convergir para a solução sem considerar os efeitos degenerativos do ruído aditivo sobre os dados, o operador, ou até mesmo as limitações impostas pelo caráter iterativo do processamento numérico.

Assim, os problemas inversos violam a terceira condição de Hadamard por serem intrinsecamente mal condicionados. Ser bem ou mal condicionado é uma propriedade intrínseca do problema inverso que se pretende avaliar. A avaliação é dita bem condicionada quando um pequeno erro no ponto de avaliação não afeta fortemente o valor da função. Já, quando esta é mal condicionada o valor da função é afetado significativamente. Um erro introduzido em um cálculo, por meio de um dado experimental ou de um arredondamento, pode produzir diversos efeitos no resultado final como redução, permanecer pequeno ou, ainda, ampliação (SILVA NETO; MOURA NETO, 2005).

Portanto, o problema de tomografia térmica é intrinsecamente mal condicionado e, desse modo, extremamente sensível na presença de erros, tanto de natureza experimental nas medidas quanto de arredondamento/truncamento nos dados do procedimento computacional. Esta sensibilidade é comprovada por Seleglim e Milioli (2001) na determinação de histogramas de diâmetros a partir do histograma de cordas obtido de uma sonda de detecção de fase imersa em escoamento a bolhas.

Devido ao mau condicionamento, as técnicas de reconstrução tradicionais são inadequadas, podendo destacar, resolução pobre, baixa distinguibilidade e produção de

imagens espúrias (artefatos) que em geral, violam grandezas físicas como frações de vazio menores do que 0 (zero) ou maiores do que 1 (um) (LÁNYI, 1998).

3.2 Métodos de Solução de Problemas Inversos e Técnicas de Regularização

A resolução do problema inverso pode ser formulada como um problema de otimização em que a função incógnita, por exemplo, a distribuição do coeficiente de convecção, é refinada até que o funcional de erro atinja seu mínimo global. Porém é extremamente importante ressaltar que, relativamente à construção da solução numérica, não existem algoritmos de uso geral capazes de lidar com as particularidades de cada problema. Ao contrário, a escolha de uma rotina deve se basear no tipo de otimização (condicionada ou não), na disponibilidade de expressões analíticas para as derivadas, na quantidade de informações a priori (valor inicial), esforço computacional, memória e assim por diante.

Os métodos de solução de um problema inverso têm sido motivo de intensas pesquisas científicas encontrando espaço nas mais diversas áreas do conhecimento. Novas metodologias surgem continuamente como resposta à dificuldade na solução do problema inverso. Entre as mais conhecidas podemos citar os métodos de otimização, decomposição em valores singulares, mínimos quadrados e variantes, métodos variacionais, métodos de regularização, molificação, métodos bayesianos, filtros digitais, redes neurais e outras novas metodologias (CAMPOS VELHO, 2001; OLIVEIRA, 2006; VOGEL, 2002; ZABARAS; YANG, 1997).

Alguns dos métodos de solução de problemas inversos mais utilizados são: Método da Descida Máxima (HUANG; YUAN; AY, 2003), Método de Newton (SHARMA, 2002), Método de Levenberg-Maquardt (SILVA NETO; ÖZISIK, 1995), Método dos Gradientes Conjugados (BOKAR; ÖZISIK, 1995), Algoritmos Genéticos (ROLNIK, 2003), Método de Decomposição em Valores Singulares (LAGIER; LEMONNIER; COUTRIS, 2004) e Métodos Bayesianos (WANG; ZABARAS, 2004).

Independente da técnica de resolução do problema inverso ou da abordagem variacional ou funcional para reconstrução da distribuição do coeficiente de convecção, as dificuldades são intrínsecas uma vez que o problema é mal posto e, provavelmente, sem solução na ausência de uma estratégia de regularização.

Uma primeira idéia para trabalhar com o mau condicionamento de um problema foi apresentada por Tikhonov e Arsenin (1977), chamada de regularização ou método de regularização. O método de regularização consiste na determinação da solução aproximada mais suave compatível com os dados observados, para certo nível de ruído. A busca da

solução mais suave (regular) é uma informação adicional, que transforma o problema mal posto em bem posto como representado na Figura 3.1 (CAMPOS VELHO, 2001).

Figura 3.1 – Idéia básica do método de regularização.

Portanto, ao invés de resolver um problema inverso diretamente, resolve-se um problema levemente alterado por um parâmetro de regularização. Diz-se que este novo problema levemente alterado é uma regularização do problema original. Entretanto, existe o problema de escolher o melhor parâmetro de regularização, sabendo que para resolver problemas mal condicionados, informações adicionais e de boa qualidade contribuem para uma melhor determinação da solução do problema.

Uma forma de avaliar se o parâmetro de regularização é ótimo é calculando um funcional de erro entre o problema original e o problema regularizado. Quando este funcional se aproximar de zero é que a solução do problema regularizado se aproxima da solução do problema original. Deve-se, portanto, buscar o parâmetro de regularização ótimo, de forma a se ter à mínima alteração do problema original, mas com a desejada estabilidade da solução.

De acordo com a teoria de problemas inversos, a reconstrução numérica deste problema é tratada como um problema de minimização global. Assim, a técnica de regularização proposta por Brandi (2010), baseia-se na construção de uma pseudo-inversa da matriz Hessiana, do método de Newton, a partir da truncagem de sua Decomposição em Valores Singulares (TSVD), para a determinação do coeficiente de convecção, a partir de medidas não intrusivas (BRANDI; ANSONI; SELEGHIM JR, 2009).

3.3 Modelagem do Problema de Condução de Calor

Na tomografia térmica a relação entre a distribuição espacial da condutividade térmica k e a temperatura θ pode ser derivada a partir do princípio de conservação de energia. Após

algumas simplificações, a equação que governa a condução de calor bidimensional pelo potencial térmico θ no domínio é dada por

$$\vec{\nabla} \cdot (-k \vec{\nabla} \theta) = 0 \text{ em } \Omega, \quad (3.1)$$

em que Ω denota o domínio do problema.

A interação entre Ω e o ambiente externo ocorre por meio do seu contorno $\partial\Omega$ e é definida pelas relações entre excitação e resposta térmica. Em termos práticos isso pode ser feito aplicando-se um fluxo de calor sobre o contorno e medindo a distribuição de temperaturas resultantes. Como em Brandi (2010) consideram-se as seguintes condições de contorno

$$\theta = \theta_0 \text{ em } \partial\Omega_1, \quad (3.2)$$

$$-k \frac{\partial \theta}{\partial n} = h\theta + q \text{ em } \partial\Omega_2, \quad (3.3)$$

em que n é o vetor normal h é o coeficiente de convecção e q é o fluxo de calor imposto no contorno. As equações (3.1)-(3.3) se possuírem parâmetros totalmente conhecidos formam um problema matematicamente bem posto, podendo ser resolvido por técnicas numéricas convencionais (BORCEA, 2002). No caso da tomografia térmica isso não ocorre: um ou mais parâmetros normalmente são desconhecidos em partes do contorno. A estratégia para a solução consiste em suprir esta deficiência de informação através da medição de condições de contorno redundantes nas partes acessíveis do contorno do domínio.

A equação (3.2) impõe a condição de Dirichlet, ou seja, temperaturas prescritas no contorno. A equação (3.3) impõe a condição de contorno mista, ou seja, o acoplamento entre a condução e a convecção de calor.

O problema direto é formulado por uma equação diferencial parcial do tipo estacionária e por condições de contorno definidas em todos os pontos do contorno. Considerando o fato de que não existe solução analítica quando, por exemplo, a distribuição da condutividade térmica não é constante, faz-se necessário a resolução do problema de modo numérico. Para tanto, o método dos elementos finitos é adotado para a discretização do domínio e das derivadas parciais.

A aplicação do método dos elementos finitos (MEF) (ASSAN, 1999; BECKER; CAREY; ODEN, 1981; MENDONÇA, 2005) foi escolhida para discretizar as equações do

problema por mostrar vantagens expressivas em relação ao método das diferenças finitas (MDF) (CUNHA, 2000; FORTUNA, 2000). Algumas dessas vantagens são:

- i) Domínio particionado em células (elementos);
- ii) A solução de interesse é a solução de alguma forma integral da equação diferencial parcial.
- iii) A matriz do sistema algébrico gerada pelo método é simétrica, positiva definida, esparsa e dependendo da numeração dos nós e elementos podem ter banda.

Os passos detalhados para a obtenção do sistema linear mediante o uso do MEF utilizados neste trabalho encontram-se em Brandi (2010). A seguir será apresentada uma breve formulação, omitindo os detalhes dos cálculos.

De acordo com a equação governante (3.1), o resíduo R de uma solução aproximada $\theta(x, y)$, é definido por

$$R(x, y) = -\bar{\nabla} \cdot [k(x, y) \bar{\nabla} \theta(x, y)]. \quad (3.4)$$

Para testar o resíduo em sub-regiões arbitrárias, multiplica-se R por uma função de teste suficientemente suave v , em seguida Rv é integrado em cada domínio no qual esse produto é suave. As equações são geradas fixando-se a média ponderada resultante de R igual a zero, ou seja,

$$\int_{\Omega} [-\bar{\nabla} \cdot (k \bar{\nabla} \theta)] v dx dy = 0. \quad (3.5)$$

A maior vantagem da equação integral é que a ordem das derivadas dentro da integral pode ser reduzida de dois para um pela integração por partes. Usando a regra da diferenciação do produto (BECKER; CAREY; ODEN, 1981), temos

$$\int_{\Omega} \left\{ \bar{\nabla} \cdot [vk \bar{\nabla} \theta] - k \bar{\nabla} \theta \cdot \bar{\nabla} v \right\} dx dy = 0. \quad (3.6)$$

A integral (3.6) pode ser transformada numa integral de contorno usando o teorema do divergente. Assim,

$$\int_{\Omega} \bar{\nabla} \cdot [vk \bar{\nabla} \theta] dx dy = \int_{\partial\Omega} k \frac{\partial \theta}{\partial n} v ds, \quad (3.7)$$

em que $\partial\theta/\partial n = \vec{\nabla}\theta \cdot \vec{n}$.

Reescrevendo a equação (3.7) e como $\partial\Omega = \partial\Omega_1 + \partial\Omega_2$ tem-se

$$\int_{\Omega} k \vec{\nabla}\theta \cdot \vec{\nabla}v dx dy - \int_{\partial\Omega_1} k \frac{\partial\theta}{\partial n} v ds - \int_{\partial\Omega_2} k \frac{\partial\theta}{\partial n} v ds = 0. \quad (3.8)$$

Como a única restrição sobre v é que seja suave, pode-se arbitrar $v = 0$ em $\partial\Omega_1$.

Substituindo a condição de contorno (3.3) na integral de contorno em (3.8), assume-se

$$\int_{\Omega} k \vec{\nabla}\theta \cdot \vec{\nabla}v dx dy + \int_{\partial\Omega_2} [h\theta + q]v ds = 0. \quad (3.9)$$

Adota-se um domínio discretizado em subdomínios formados por elementos que contenham os nós. A equação (3.9) será aplicada tantas vezes quantos forem os nós cujas temperaturas forem desconhecidas gerando assim um número suficiente de equações para que sejam determinadas. Em particular, no método dos elementos finitos a função de ponderação v tem uma forma especial assumindo valor unitário no nó cuja temperatura é desconhecida e zero nos demais nós que o circundam. A formulação pormenorizada pode ser encontrada em Brandi (2010).

3.4 Medição não intrusiva do coeficiente de convecção formulada como um problema térmico inverso

As equações (3.1)-(3.3), se possuírem parâmetros totalmente conhecidos, formam um problema matematicamente bem posto, podendo ser resolvido por técnicas numéricas convencionais (BORCEA, 2002). No caso da tomografia térmica isso não ocorre: um ou mais parâmetros normalmente são desconhecidos em partes do contorno ou do domínio.

A estratégia para a solução consiste em suprir esta deficiência de informação através da medição de condições de contorno redundantes nas suas partes acessíveis. Neste trabalho será modelado pela subdivisão do contorno $\partial\Omega_2$ em três partes, isto é,

$$\partial\Omega_2 = \partial\Omega_{2,1} \cup \partial\Omega_{2,2} \cup \partial\Omega_{2,3}, \quad (3.10)$$

em que $\partial\Omega_{2,1}$ representa uma região externa acessível, sobre a qual medidas experimentais podem ser feitas, e regiões internas inacessíveis, porém onde os parâmetros da equação (3.3) são totalmente conhecidos a priori ($\partial\Omega_{2,2}$) ou onde eles devem ser determinados ($\partial\Omega_{2,3}$). Em

suma, o perfil do coeficiente de convecção ocorrendo em $\partial\Omega_{2,3}$, deve ser determinado a partir de condições de contorno redundantes (excitação e resposta) sobre $\partial\Omega_{2,1}$, além de informações a priori sobre $\partial\Omega_{2,2}$. Na Figura 3.2 temos a geometria do problema simulado numericamente.

Figura 3.2 – Geometria do problema simulado numericamente

O problema inverso é formulado como um problema de minimização de um funcional de erro quantificando a diferença entre as temperaturas medidas sobre $\partial\Omega_{2,1}$, denotadas θ_{real} , e temperaturas θ_{num} obtidas de um modelo prospectivo. Estas temperaturas são calculadas a partir da solução de (3.1) com as mesmas condições sobre a parte conhecida do contorno, porém impondo-se um perfil prospectivo para o coeficiente de convecção, denotado h_{num} , na parte inacessível do contorno $\partial\Omega_{2,3}$. Para tanto, considere dois modelos distintos: um correspondendo à própria montagem experimental (modelo de referência) e o outro correspondendo à implementação numérica das equações (3.1)-(3.3) em um computador digital (modelo prospectivo). Matematicamente este funcional de erro pode ser expresso como

$$e(h) = \|\theta_{real} - \theta_{num}\|, \quad (3.11)$$

em que as temperaturas θ_{real} e θ_{num} referem-se apenas a região acessível do contorno onde medidas podem ser realizadas.

Usualmente na equação (3.11) adota-se a norma euclidiana, ou seja,

$$e(h) = \sqrt{\sum (\theta_{real} - \theta_{num})^2}. \quad (3.12)$$

Neste trabalho como realizado no trabalho de Brandi (2010), as temperaturas medidas, ou sintéticas, são geradas numericamente da mesma forma que θ_{num} , porém com o perfil correto h_{real} imposto sobre $\partial\Omega_{2,3}$ na equação (3.3). Assim, um procedimento numérico capaz de produzir refinamentos sucessivos em h_{num} de maneira que o erro seja minimizado, conduz, hipoteticamente, à solução do problema, fazendo com que $h_{num} = h_{real}$.

3.5 Métodos de Otimização e Estratégia de Regularização

De uma maneira geral, os métodos numéricos de otimização podem ser classificados em dois grandes grupos: métodos determinísticos e estocásticos. Em ambos os casos procura-se estabelecer um procedimento iterativo que resulte no refinamento dos parâmetros do problema de forma que o funcional de erro convirja para seu mínimo global. Existem também os métodos híbridos, que combinam a estratégia de busca global dos métodos estocásticos com busca local dos métodos determinísticos.

Como mencionado anteriormente a função de erro depende do vetor de coeficientes de convecção especificados no contorno interno. Assim, a solução do problema é encontrada determinando os valores locais do coeficiente de convecção que minimizam a função de erro. Matematicamente isso pode ser expresso por uma expressão que constitui um sistema de equações impondo que a derivada do erro em relação a cada um dos valores do coeficiente de convecção seja nula, isto é,

$$\frac{\partial e}{\partial h_i} = 0. \quad (3.13)$$

Estas equações são não-lineares porque a dependência entre h_i e a função de erro e se dá através da equação diferencial do problema (3.1) e (3.3). Assim, as equações (3.13) devem ser resolvidas com o auxílio de algum método numérico.

Os métodos determinísticos, quando aplicados a este tipo de problemas, geralmente determinam um procedimento iterativo que, depois de um certo número de iterações supostamente converge para o mínimo da função de erro. O procedimento iterativo pode ser escrito da seguinte forma geral (COLAÇO; ORLANDE; DULIKRAVICH, 2004)

$$h_{k+1} = h_k + \lambda_k d_k, \quad (3.14)$$

em que λ_k corresponde ao passo de refinamento e pode ser determinado com o auxílio de um método de busca unidirecional (COLAÇO; ORLANDE; DULIKRAVICH, 2004), d_k representa a direção de refinamento e o índice k indica o k -ésimo ciclo de otimização.

Normalmente os métodos determinísticos de otimização diferem pela forma de se calcular a direção de refinamento. Por exemplo, o método da Descida Máxima (NOCEDAL; WRIGHT, 1999), que é o mais básico método de gradiente adota a direção de refinamento contrária ao vetor gradiente, isto é,

$$d_k = -\vec{\nabla}e(h_k), \quad (3.15)$$

em que $\vec{\nabla}e$ corresponde ao vetor gradiente associado com a superfície de erro e calculado na k -th iteração.

A principal característica deste método é a capacidade de convergir para a solução mesmo que a estimativa inicial esteja longe do mínimo global. Uma desvantagem importante é a facilidade com a qual a trajetória de convergência é aprisionada por um mínimo local.

Uma técnica de otimização interessante é o método de Newton (NOCEDAL; WRIGHT, 1999) que, além das derivadas de primeira ordem do funcional de erro, conforme utilizadas no método da Descida Máxima e dos Gradientes Conjugados fazem uso também da derivada de segunda ordem para acelerar a convergência. Este método adota a direção de refinamento dada por

$$d_k = -H(h_k)^{-1} \vec{\nabla}e(h_k), \quad (3.16)$$

em que H corresponde a matriz Hessiana contendo as derivadas de segunda ordem da função de erro em relação aos valores locais do coeficiente de convecção, ou seja,

$$H_{i,j} = \frac{\partial^2 e}{\partial h_i \partial h_j}. \quad (3.17)$$

Substituindo a direção de refinamento (3.16) na equação geral (3.14), o método de Newton pode ser escrito da seguinte forma

$$h_{k+1} = h_k - \lambda_k H(h_k)^{-1} \vec{\nabla} e(h_k). \quad (3.18)$$

O cálculo da matriz Hessiana, em geral, é o processo computacionalmente mais custoso, podendo-se usar técnicas numéricas para o cálculo desta matriz. Como em Brandi (2010), o método das diferenças finitas foi adotado para o cálculo da matriz Hessiana, no qual suas derivadas foram aproximadas por diferenças centrais de segunda ordem.

Este método é muito eficiente se a matriz do problema de otimização é bem condicionada. Como o problema é inverso e, portanto, intrinsecamente mal condicionado, é esperado que H seja problemática em algum sentido. Mais detalhes sobre o condicionamento da matriz Hessiana do problema podem ser vistos no trabalho de Brandi (2010).

O método de Newton tem uma convergência quadrática, porém pode não convergir, dependendo se a estimativa inicial estiver fora de uma região de convergência (região próxima ao máximo/mínimo) ou devido ao mau condicionamento da matriz Hessiana.

Considerando que o funcional de erro não foi regularizado através de aplicação de técnicas específicas, como o método de Tikhonov, por exemplo, algum procedimento equivalente deve ser aplicado na matriz Hessiana. Assim, a técnica de regularização adotada no trabalho (BRANDI, 2010) baseou-se na construção de uma pseudo-inversa da matriz Hessiana a partir da truncagem de sua Decomposição em Valores Singulares (TSVD), no qual os autovetores correspondentes aos autovalores singulares ou quase singulares de H são truncados. Ou seja, isto corresponde a descartar uma combinação linear do conjunto de equações que se está tentando resolver. A explicação é que se descarta justamente a combinação de equações que, de tão corrompida por erros de arredondamento, atrai fortemente a solução para uma direção próxima daquela do espaço nulo.

CAPÍTULO 4 IMPLEMENTAÇÃO NA ARQUITETURA PARALELA

Como visto no capítulo anterior, o princípio de funcionamento da tomografia térmica geralmente se baseia na aplicação de um fluxo de calor na superfície externa do processo e na medição das temperaturas resultantes nessa mesma superfície. Dentre as principais vantagens da tomografia térmica estão o seu baixo custo e robusteza, o que a torna extremamente interessante para aplicações industriais. No entanto, o problema da tomografia térmica é altamente custoso, em termos computacionais, requerendo considerável tempo de processamento na CPU, principalmente para o cálculo e regularização da matriz Hessiana do método de otimização de Newton.

Para o cálculo da matriz Hessiana foi adotado o método das diferenças finitas, no qual suas derivadas foram aproximadas por diferenças centrais de segunda ordem. Para a solução dos sistemas lineares adotou-se o método dos Gradientes Biconjugados Pré-Condicionados (PBCG – *Preconditioned Biconjugate Gradient*) (BRANDI, 2010).

Na Tabela 4.1, observa-se o número de vezes que o método do PBCG é chamado e a porcentagem de tempo gasto no cálculo e regularização da matriz Hessiana em uma iteração do método de Newton na solução de um problema térmico inverso estudado.

Tabela 4.1 – Número de vezes que o método PBCG é chamado e porcentagem de tempo gasto na montagem da matriz Hessiana

Método de Newton	Cálculo e regularização da matriz Hessiana	Porcentagem de tempo (s)
3869	3813	98%

Ao observar a tabela, nota-se que a solução do problema térmico inverso proposta por Brandi (2010) é dominada, quase na sua totalidade pelo cálculo e montagem da matriz Hessiana, no que se refere a tempo de processamento.

Desta forma constatou-se que o grande gargalo de processamento do problema consiste no cálculo e montagem da matriz Hessiana, na qual vários Gradientes Biconjugados Pré-condicionados devem ser utilizados de maneira a resolver os vários sistemas lineares independentes gerados.

A partir desta orientação, buscou-se desenvolver algoritmos paralelos que busquem tirar o máximo proveito dos processadores *multi-cores* em arquiteturas de memória compartilhada.

Em um primeiro estudo, procurou-se tirar proveito do poder computacional apresentado pelas GPUs modernas, desenvolvendo algoritmos de modo que seja possível transferir vários sistemas lineares esparsos para a memória global da placa gráfica e resolvê-los de forma independente.

Em um segundo estudo implementou-se rotinas buscando utilizar a capacidade de processamento dos processadores *multi-cores* das CPUs modernas, para acelerar a montagem da matriz Hessiana.

Este capítulo apresenta as estruturas de dados desenvolvidas para possibilitar a montagem e regularização da matriz Hessiana de forma paralela utilizando as ferramentas de programação paralela *multi-core* em arquitetura de memória compartilhada NVIDIA CUDA e Pthreads.

4.1 Implementação na GPU – NVIDIA CUDA

O processo desenvolvido por Brandi (2010) que se pretende acelerar envolve fundamentalmente operações de cálculo vetorial e matricial. O método iterativo utilizado para a resolução dos sistemas lineares de equações gerados é dominado fundamentalmente por operações de multiplicação entre matrizes esparsas e vetores densos.

Devido ao custo computacional destas operações, torna-se interessante a sua paralelização, de forma a acelerar o cálculo, sendo a GPU uma unidade de processamento paralelo otimizado para operações sobre matrizes e vetores, tornando-se atraente a idéia de executar todo este processamento na GPU.

Vários trabalhos vêm sendo publicados na literatura buscando tirar proveito do poder de processamento da GPU na resolução de sistemas lineares esparsos (BOLZ et al., 2003; BUATOIS; CAUMON; LÉVY, 2008; CEVAHIR; NUKADA; MATSUOKA, 2009; KRÜGER; WESTERMANN, 2003; WIGGERS et al., 2007). O GPU moderno é um processador altamente paralelo e com enorme largura de banda, cujos processadores

respeitam um modelo SIMD. Quando utilizado para computação genérica, o GPU atua como um co-processador ao CPU (Figura 4.1).

Figura 4.1 – Esquema típico de um sistema que use o GPU como um co-processador.

Desta forma busca-se neste trabalho o desenvolvimento de um algoritmo, recorrendo a este modelo de utilização da GPU como um co-processador, para resolução de sistemas lineares de equações, especialmente desenhados para que possam ser executados em paralelo no processador gráfico sem a intervenção do processador central (CPU), evitando dessa forma comunicações desnecessárias entre CPU e GPU devido ao gargalo (*bottleneck*) apresentado pela largura de banda da PCI-Express como pode ser observado na Figura 4.1.

O que se pretende desenvolver neste trabalho é algo diferente do que foi proposto na literatura, na qual somente algumas operações são efetuadas na GPU e devolvidas à CPU para que este termine a execução do método iterativo. Uma destas operações é a multiplicação da matriz esparsa por um vetor denso que é a operação que demanda mais tempo de processamento na resolução do método iterativo.

Como no problema térmico inverso é necessário a resolução de vários sistemas lineares independentes na montagem e regularização da matriz Hessiana, surgiu a idéia de implementar o método iterativo, mais precisamente o método dos Gradientes Conjugados Pré-Condicionados (PCG) inteiramente no espaço da memória da placa gráfica, de modo que seja possível a resolução de vários PCGs na sua totalidade em paralelo na GPU.

Torna-se determinante agora, a escolha das estruturas de dados apropriadas para a implementação das matrizes esparsas, de maneira que seja possível a montagem de N sistemas em CPU e que sejam transferidos para GPU de modo que ofereçam suporte às operações necessárias que serão executadas na placa gráfica. Também se apresenta a seguir o

método iterativo utilizado para a resolução dos sistemas lineares implementado no espaço da memória da GPU.

4.1.1 Armazenamento da matriz esparsa

O armazenamento dos dados requer considerável atenção no momento em que se lida com matrizes esparsas e de grande porte. A minimização do espaço de memória obtida por desconsiderar elementos nulos, a preservação de estruturas da matriz, como simetria ou banda de elementos não nulos, e a importância de operações com matriz são particularidades de diferentes técnicas para armazenamento de matrizes esparsas.

Os algoritmos que permitem trabalhar diretamente com matrizes esparsas são mais complexos e pesados que os algoritmos utilizados no tratamento de matrizes cheias. Esta complexidade resulta não só da necessidade de se evitarem seqüências de operações redundantes, mas também do tipo de armazenamento utilizado, onde o acesso à informação é efetuado de uma forma indireta.

O método mais simples para armazenamento de matrizes esparsas é conhecido como armazenamento por coordenadas (*Coordinate Format*) (SAAD, 2003) o qual utiliza três vetores para efetuar o armazenamento dos elementos não nulos e nos outros dois restantes são armazenados respectivamente o índice correspondentes as linhas e colunas dos elementos não nulos.

O armazenamento compacto por coluna (*Compressed Sparse Row - CSR*) (BARRETT et al., 1994; SAAD, 2003) é uma estrutura de dados eficiente e amplamente utilizada, guardando em um vetor apenas os elementos diferentes de zero existente na matriz, linha a linha. Este formato utiliza o endereçamento indireto, e é tido como o método mais popular para armazenar matrizes esparsas e muito utilizada na paralelização de operações envolvendo a matriz esparsa em GPU (CEVAHIR; NUKADA; MATSUOKA, 2009; WIGGERS et al., 2007).

O armazenamento compacto em blocos (*Block Compressed Row Storage – BCRS*) analisado foi desenvolvido no trabalho *Concurrent Number Cruncher – CNC* de BUATOIS; CAUMON; LÉVY, 2008 para representar de uma forma genérica matrizes esparsas respeitando uma estrutura ligeiramente diferente do formato CRS, com o objetivo de obter uma estrutura de dados mais adequada ao processamento paralelo em GPU. O BCRS implementado no CNC agrupa os coeficientes não-nulos da matriz em blocos, de dimensões

2x2 ou 4x4, por forma a maximizar o uso da largura de banda da memória gráfica dos GPUs e a reduzir o numero de leituras de memória global reduzindo a sua latência.

Entretanto como se busca transferir várias matrizes esparsas para a memória global da GPU foi necessário desenvolver uma estrutura de dados para o armazenamento das matrizes esparsas do problema de modo a permitir a montagem de N sistemas de uma vez e sua transferência para a memória global da placa gráfica.

A primeira etapa para o desenvolvimento do algoritmo paralelo foi a criação de uma estrutura de dados capaz de armazenar os elementos não-nulos das N matrizes esparsas. Essa estrutura consiste em um vetor de estruturas (vetor de *structs*) como mostrado a seguir:

Algoritmo 4: Estrutura de dados para armazenamento das N matrizes esparsas

```
typedef struct __align__(16){
    float elemento[tam];
    int coluna[tam];
    int size;
} armazenamentoMatrizEsparsa

armazenamentoMatrizEsparsa m[nSistMAX];
```

Onde `__align__(16)` é uma diretiva para estruturas do CUDA para garantir que em estruturas superiores a 16 *bytes*, o compilador gere o número mínimo de instruções e `tam` seja o número máximo de elementos não-nulos por linha da matriz esparsa.

Em seguida alocamos um vetor do tipo `armazenamentoMatrizEsparsa` de modo que as N primeiras posições do vetor representem as linhas da primeira matriz esparsa armazenada e as posições $N + 1$ até $2N$ sejam para a segunda matriz, assim por diante, até que todas as matrizes sejam armazenadas.

A seguir apresentamos um exemplo de algoritmo para montagem da matriz esparsa na estrutura de dados desenvolvida, considerando matrizes quadradas de dimensões $N \times N$ e `nSist` represente o número de sistemas armazenados.

Algoritmo 5: Armazenamento da matriz esparsa

```
for (i = 0; i < N; i++){
    m[i + (nSist*N)].size = 0;
    for (j = 0; j < N; j++){
        if (MatrizEsparsa[i][j] != 0){
            m[i+(nSist*N)].coluna[m[i+(nSist*N)].size] = j;
            m[i+(nSist*N)].elem[m+(nSist*N).size]=MatrizEsparsa[i][j]
            m[i+(nSist*N)].size ++;
        }
    }
}
```

Na Figura 4.2 temos a representação do armazenamento na estrutura desenvolvida para duas matrizes, as quais podem ser observadas o armazenamento dos elementos não-nulos sendo realizado por linhas.

Figura 4.2 – Armazenamento das matrizes esparsas.

Assim, as estruturas de dados necessárias para a implementação do método de resolução dos sistemas lineares esparsos, devem oferecer suporte adequado a estas matrizes de ocupação esparsa suportando simultaneamente algoritmos eficientes para as operações matemáticas necessárias tais como o produto entre matrizes esparsas e vetores densos ou a adição de matrizes.

4.1.2 Método dos Gradientes Conjugados na GPU

O método dos Gradientes Conjugados é um método amplamente utilizado para a solução iterativa de sistemas definidos positivos de equações lineares (HESTENES; STIEFEL, 1952; SHEWCHUK, 1994), sendo adequado à resolução de sistemas esparsos de grandes dimensões.

O método dos Gradientes Conjugados consiste em um método iterativo de busca do mínimo local da função. Desta forma, geram-se aproximações para a solução e em cada

iteração do método, dois produtos internos são realizados para que se calculem dois escalares definidos de forma que a sequência obedeça a condições de ortogonalidade.

Dados as entradas A , b , um valor inicial x , um preconditionador M , um número máximo de iterações $i_{\max}$ e um valor de tolerância de erro $\varepsilon < 1$, o sistema linear expresso como $Ax = b$ pode ser resolvido utilizando o algoritmo do Gradiente Conjugado Pré-condicionado descrito como:

Algoritmo 6: Pseudocódigo para o algoritmo dos Gradientes Conjugados Pré-condicionado (PCG)

```

 $r_0 := b - Ax_0$ 
 $z_0 := M^{-1}r_0$ 
 $p_0 := r_0$ 
 $k_0 := 0$ 
repeat
 $\alpha := \frac{r_k^T z_k}{p_k^T A p_k}$ 
 $x_{k+1} := x_k + \alpha p_k$ 
 $r_{k+1} := r_k + \alpha A p_k$ 
if  $r_{k+1}$  sufficiently small then exit loop end if
 $z_{k+1} := M^{-1}r_{k+1}$ 
 $\beta := \frac{r_{k+1}^T z_{k+1}}{r_k^T z_k}$ 
 $p_{k+1} := z_{k+1} + \beta p_k$ 
 $k := k + 1$ 
end repeat
The result is  $x_{k+1}$ 

```

Para $M = I$, onde I é a matriz identidade o algoritmo resulta no método dos Gradientes Conjugados sem pré-condicionador. Neste trabalho foi utilizado o algoritmo com o pré-condicionador de Jacobi ($M = \text{diag}(A)$).

Em nível operacional, o método dos Gradientes Conjugados envolve, para cada iteração, dois produtos entre uma matriz esparsa e um vetor denso, três operações sobre vetores (soma, subtração ou multiplicação por um escalar) e dois produtos internos.

Dado o custo computacional destas operações, torna-se interessante a sua paralelização sendo o GPU uma unidade de processamento paralelo otimizado para operações sobre matrizes e vetores, de forma a acelerar o cálculo e montagem da matriz Hessiana.

O desenvolvimento de um algoritmo, recorrendo ao poder computacional da GPU como um co-processador genérico, para a resolução de sistemas lineares esparsos especialmente desenhados para que possa ser executado em paralelo no processador gráfico com o mínimo de intervenção do processador central CPU.

Como todas as operações que dominam o método dos Gradientes Conjugados são operações facilmente paralelizáveis, estamos perante um algoritmo com um padrão embaraçosamente paralelo. Torna-se possível a perspectiva de executar a totalidade do algoritmo no espaço de memória do GPU, evitando ao máximo a utilização do bus PCI-Express, que representa o maior gargalo na utilização do GPU como um co-processador ao CPU.

Como mencionado anteriormente buscou-se uma estratégia diferenciada para a implementação do PCG utilizando o poder de processamento da GPU. Em trabalhos encontrados na literatura, principalmente pelo apresentado por BUATOIS; CAUMON; LÉVY, 2008, o método PCG é utilizado para resolver sistemas lineares esparsos acelerados pela paralelização em GPU. Entretanto essa paralelização recai nas operações de multiplicação da matriz esparsa por um vetor denso utilizando um pacote desenvolvido por BELL; GARLAND, 2008, operações sobre vetores (soma, subtração ou multiplicação por um escalar) e dois produtos internos.

Neste trabalho como é necessário que se resolva vários sistemas lineares esparsos independentes para a montagem da matriz Hessiana, buscou-se implementar todo o método do PCG no espaço da memória da placa gráfica, de modo que possam ser resolvidos vários sistemas lineares esparsos paralelamente respeitando a capacidade da memória da GPU.

O algoritmo foi desenvolvido de maneira a tirar o máximo proveito possível da memória compartilhada (*shared memory*) da GPU, nos quais vetores e variáveis auxiliares que somente serão utilizados dentro do método são armazenados nesta memória ultra-rápida.

Para que estes cálculos sejam possíveis é necessário transferir para a memória global da placa gráfica os sistemas lineares esparsos, utilizando a estrutura de dados desenvolvida. Após a execução da resolução dos sistemas na GPU é retornado para a memória RAM os vetores contendo a solução dos sistemas lineares para que possam ser utilizados pela CPU.

Na Figura 4.3 observa-se a representação da utilização da GPU como um co-processador. O programa inicia a sua execução com uma *thread* executada na CPU (*host*),

após a montagem dos sistemas lineares é chamado o *kernel* para a execução em GPU (*device*) dos métodos dos Gradientes Conjugados Pré-condicionados (PCGs). A execução na GPU é executada por várias *threads* organizadas em blocos o qual é alocado a um multiprocessador da GPU que será responsável pela execução de um PCG. Após a execução dos PCGs no *device* o programa termina sua execução com uma *thread* no *host*.

Figura 4.3 – Execução do método do PCG na GPU

No APÊNDICE A pode ser observado a implementação do método do PCG na GPU desenvolvido neste trabalho.

Cabe mencionar que todo o algoritmo implementado em GPU utiliza precisão simples, devido a restrições apresentada na placa gráfica NVIDIA GeForce 9800 GT que proporciona suporte apenas a esta precisão. Na seção seguinte são apresentados testes realizados de modo a verificar se o algoritmo desenvolvido em GPU apresenta uma convergência satisfatória, sendo apto para a aplicação no problema tomográfico inverso que é intrinsecamente mal condicionado.

Como conclusão dessa seção cabe salientar a grande dificuldade em trabalhar com matrizes esparsas na tecnologia CUDA, sendo citada em vários trabalhos científicos (BOLZ et al., 2003; BUATOIS; CAUMON; LÉVY, 2008; CEVAHIR; NUKADA; MATSUOKA, 2009).

O CAPÍTULO 5 trata da avaliação do desempenho obtido para a resolução dos sistemas esparsos do problema térmico inverso.

4.1.3 Verificação do modelo numérico e análise de convergência do Código em GPU

Para a verificação da convergência do algoritmo desenvolvido em GPU foram realizados testes de problemas simples, ou seja, foi analisado um problema direto de transferência de calor que é bem posto e preciso, pois apresenta solução analítica que possibilita a análise da convergência do algoritmo.

Para validação do código existente o problema bidimensional de condução de calor foi considerado (BRANDI, 2010). Soluções analíticas e numéricas com condições de contorno diferentes foram simuladas com o propósito de avaliar a precisão do código utilizando o processamento em GPU.

Os testes numéricos para validação do código desenvolvido foi executado em um computador contendo uma placa gráfica NVIDIA GeForce 9800 GT com 1 Gb de memória. Nesta primeira análise focou-se o estudo da convergência do método executado na GPU, devido ao fato dela trabalhar somente com precisão simples.

O primeiro teste trata-se da solução do problema de condução bidimensional em regime permanente em uma placa de lados unitários, com temperatura prescrita (MALISKA, 1995) e $\theta_{\infty} = 0$ em todo o contorno conforme mostra a Figura 4.4, cuja equação diferencial é

$$\nabla^2 \theta = 0. \quad (4.1)$$

A solução analítica desse problema é dada por

$$\theta(x, y) = \frac{\sinh\left(\frac{\pi y}{a}\right)}{\sinh\left(\frac{\pi b}{a}\right)} \sinh\left(\frac{\pi x}{a}\right). \quad (4.2)$$

Figura 4.4 – Geometria do problema de condução bidimensional em uma placa.

Três malhas foram consideradas para simulação desse problema: a malha grossa (5×5 pontos, 25 nós, $\delta x = \delta y = 0.25$ e 32 elementos triangulares); a malha intermediária (9×9 pontos, 81 nós, $\delta x = \delta y = 0.125$ e 128 elementos triangulares) e a malha fina (17×17 pontos, 289 nós, $\delta x = \delta y = 0.0625$ e 512 elementos triangulares). Os resultados destas simulações são apresentados nas Figuras (4.5 – 4.7). Pode-se observar destas figuras que as soluções numéricas simuladas utilizando a GPU com precisão simples estão qualitativamente de acordo com a solução analítica do problema em questão.

Figura 4.5 – Comparação malha grossa (25 nós) entre soluções: (a) numérica e (b) analítica.

Figura 4.6 – Comparação malha intermediária (81 nós) entre as soluções: (a) numérica e (b) analítica.

Figura 4.7 – Comparação malha fina (289 nós) entre as soluções: (a) numérica e (b) analítica.

Os resultados numéricos foram analisados de maneira a mostrar a convergência do método numérico apresentado. O erro relativo (E_2) na norma L_2 entre as soluções numéricas e a solução analítica é apresentado pela equação

$$E_2 = \frac{\sum (\theta_{analítica} - \theta_{numérica})^2}{\sum (\theta_{analítica})^2}. \quad (4.3)$$

Como visto em Brandi (2010) foi realizada a análise utilizando a norma L_∞ entre as soluções numéricas e a solução analítica. A norma L_∞ é mais severa em relação à norma L_2 , segundo a literatura, a solução numérica deve convergir na média para a solução exata do problema conforme a malha é progressivamente refinada, isso significa que pode haver um número finito de pontos nos quais o erro não tende a zero. Matematicamente a norma L_∞ é representada pela seguinte equação

$$E_\infty = \frac{\max |\theta_{analítica} - \theta_{numérica}|}{|\theta_{analítica}|}. \quad (4.4)$$

Uma síntese dos erros relativos nas três malhas é apresentada na Tabela 4.2. Estes erros quantificam as observações consideradas nas Figuras (4.5 – 4.7). Nota-se na Tabela 4.2 que os erros decrescem com o refinamento da malha confirmando a convergência do método numérico para este problema. Na Figura 4.8 observa-se que conforme o refinamento progressivo das malhas os erros estão diminuindo, mostrando o comportamento de convergência do método apresentado.

Tabela 4.2 – Erros para o problema de calor definido na Figura 4.4.

Malha	Erro E_2	Erro E_∞
Grossa	4,22E-04	3,66E-02
Intermediária	4,02E-05	9,50E-03
Fina	3.08E-06	2,90E-03

Figura 4.8 – Erro relativo na norma L_∞ para o problema de condução de calor.

4.2 Implementação na CPU – Pthreads

O programa para solução do problema inverso tomográfico (BRANDI, 2010) também foi acelerado utilizando o poder de processamento das CPUs *multi-cores*. A ferramenta de programação paralela utilizada foi a Pthreads.

Essa API de manipulação de *threads* é utilizada em programação paralela com o objetivo de aumentar o desempenho das aplicações. Esse aumento de desempenho se deve ao fato de múltiplas *threads* terem um custo de criação e de comunicação menor que o custo existente em múltiplos processos (SILBERSCHATZ; GALVIN; GAGNE, 2002).

Para a implementação do código em paralelo utilizando Pthreads partiu-se de um código seqüencial na linguagem C desenvolvido por Brandi (2010). A seguir são descritos a forma de armazenamento da matriz esparsa e o método de resolução dos sistemas lineares esparsos utilizados em Brandi (2010) e que foi paralelizado de modo a acelerar a montagem da matriz Hessiana do problema.

4.2.1 Armazenamento da matriz esparsa

O armazenamento ideal da matriz esparsa ocupando um espaço mínimo de memória e a solução do sistema linear consumindo o menor tempo possível é capaz de afetar significativamente a eficiência da resolução do problema tomográfico. Visto que se trata de fatores fortemente relacionados, ou seja, uma resolução ágil do sistema linear depende do acesso aos elementos da matriz e, conseqüentemente, de como foram armazenados.

De acordo com as características da discretização do domínio e com a preocupação em minimizar o espaço de memória e facilitar o acesso aos coeficientes da matriz armazenada, o

processo de obtenção do sistema linear percorrendo os nós incógnitas ao invés de elementos possibilitou um armazenamento otimizado da matriz (BRANDI, 2010). Em outras palavras, o algoritmo obtém uma linha do sistema linear a cada nó incógnita percorrido e só armazena os coeficientes não nulos em um vetor de pontos flutuantes, sa , e os índices da coluna dos respectivos coeficientes em um vetor de inteiros, ija . Desta forma, não é preciso armazenar as matrizes elementares e nem realizar a sobreposição de matrizes elementares na obtenção do sistema linear, procedimentos estes indispensáveis na abordagem em que os elementos são percorridos. Esta técnica de armazenamento de matrizes esparsas é conhecida como *row-indexed sparse storage mode* (PRESS et al., 1992).

No APÊNDICE B apresenta-se uma explicação mais detalhada e um pseudocódigo para se armazenar matrizes esparsas quadradas utilizando o *row-indexed sparse storage mode*.

4.2.2 Resolução do sistema linear esparso

A seguir apresenta-se o método de resolução do sistema linear esparso utilizado por Brandi (2010), o qual é utilizado para montagem da matriz Hessiana do problema térmico inverso.

Para a paralelização utilizando Pthreads utilizamos as mesmas ferramentas de armazenamento e resolução dos sistemas lineares esparsos que Brandi (2010) abordou em seu trabalho. Um dos motivos para manter as mesmas estruturas de dados deve-se ao fato de poder analisar a convergência do programa paralelizado utilizando precisão dupla, pois como mencionado anteriormente na paralelização utilizando a placa gráfica os algoritmos desenvolvidos utilizaram precisão simples e devido aos bons resultados apresentados em termos de precisão obtidos em outros trabalhos com essas rotinas (ROLNIK, 2003).

O método dos Gradientes Conjugados é extremamente eficiente no caso em que a matriz do sistema é simétrica e positiva definida (BARRETT et al., 1994) como já mencionada anteriormente. Entretanto, um mau condicionamento da matriz pode impedir que a solução seja efetivamente obtida.

Assim sendo o método utilizado no problema consistiu do método dos Gradientes Conjugados Biconjugados Pré-Condicionados (PBCG), uma variação do método dos Gradientes Conjugados, que tem a característica de solucionar quaisquer sistemas lineares desde que a matriz não seja singular. O PBCG foi desenvolvido de maneira que a sequência de resíduos ortogonais utilizados no CG fosse substituída por seqüências mutuamente ortogonais, porém, com a desvantagem de não mais garantir a minimização do resíduo. Este é

um ponto muito importante, pois nos métodos do CG e PCG, desconsiderando o erro de arredondamento, estes métodos garantem que se a matriz dos coeficientes é simétrica e positivo-definida então a norma do vetor resíduo tende a zero. O método dos Gradientes Biconjugados, e suas derivações, não se baseiam na teoria de otimização, e sim na teoria de Lanczos.

O método consiste de uma matriz pré-condicionadora $\tilde{A}$ que satisfaça a condição $\tilde{A}^{-1}A \approx I$, ou seja, $\tilde{A}$ próxima de A . A intenção é gerar sistemas lineares auxiliares de fácil resolução que permitam uma convergência acelerada do algoritmo (PRESS et al., 1992).

A matriz $\tilde{A}$ deve ser definida de maneira que sua inversa seja previamente conhecida e pode, por exemplo, ser definida como sendo simplesmente a diagonal de A (interessante se esta for diagonal dominante). Entretanto, em alguns casos, $\tilde{A}$ e sua inversa podem ser definidas a partir de imagens obtidas da solução do problema tomográfico em tempos anteriores. Em outras palavras, em um esquema dinâmico em que o problema tomográfico é resolvido para produzir varias imagens por segundo, gerando assim uma visualização dinâmica, $\tilde{A}$ em um determinado instante pode ser definida como a matriz A do instante anterior.

O uso da matriz pré-condicionadora, como sendo a inversa da diagonal de A , o que corresponde ao pré-condicionador de Jacobi, do PBCG resultou em soluções satisfatórias e juntamente com o armazenamento adotado evidenciou um bom desempenho no que diz respeito à eficiência da resolução do problema tomográfico paralelo.

No APÊNDICE C está descrito a obtenção do PBCG a partir do CG.

4.2.3 Paralelização em Pthreads

Na paralelização do código utilizando a API Pthreads partiu-se diretamente do código sequencial para a resolução do problema térmico inverso implementado na linguagem C. O Pthreads permite a criação de *threads*. Para cada *thread* criada pode ser atribuída uma função, bem como os respectivos argumentos

A implementação na CPU incidiu em paralelizar o cálculo e a montagem da matriz Hessiana do problema térmico inverso, que consiste no grande gargalo na resolução do problema.

Na Figura 4.9 vemos o modelo do código o qual foi paralelizado utilizando os procedimentos da API Pthreads. O código parte de uma *thread* mestre que executa a parte

serial do programa. Ao chegar na função do cálculo e montagem da matriz Hessiana são criadas *threads* de modo que cada *thread* criada seja responsável pelo cálculo e montagem de uma linha da matriz Hessiana. Ao término do cálculo e montagem da matriz Hessiana a execução procede de modo serial através da *thread* mestre. Como o modelo de programação paralela adotada utiliza memória compartilhada foi necessário o cuidado de não ocorrer condição de corrida nas variáveis.

Figura 4.9 – Execução do código utilizando Pthreads.

Para o cálculo da matriz Hessiana é necessário a discretização das equações diferenciais parciais do problema utilizando o método dos elementos finitos (MEF) e a resolução do sistema linear esparso.

No capítulo seguinte são apresentados os tempos de execução e a convergência obtida com os códigos paralelizados.

CAPÍTULO 5 TESTES NUMÉRICOS E RESULTADOS

O trabalho desenvolvido nesta dissertação visa a aceleração do tempo de processamento do trabalho desenvolvido por Brandi (2010), recorrendo ao modelo de programação paralela em arquiteturas de memória compartilhada utilizando o modelo de hardware e de programação fornecido pelo CUDA (GPU) e pelo Pthreads (CPU).

Neste capítulo são apresentados os resultados obtidos para validação do modelo numérico paralelizado e a análise de convergência do código na GPU e na CPU *multi-core*. O estudo de caso, as trajetórias de convergência, os tempos de processamentos e os *speedups* obtidos também são apresentados.

5.1 Infraestrutura para a realização dos testes

A seguir apresentam-se a infraestrutura disponível para a realização dos testes. Como houve a paralelização utilizando tanto o poder computacional das GPUs modernas e a das CPUs *multi-cores*, cabe aqui a descrição detalhada dos hardwares empregados para tal processamento como as versões do *kernel* do Linux utilizados (todos os testes foram realizados no ambiente Linux).

Na Tabela 5.1 é possível observar as especificações das GPUs utilizadas para os testes, sendo que a GPU 1 foi adquirida pelo laboratório do Núcleo de Engenharia Térmica e Fluidos (NETeF) no início do desenvolvimento do projeto para possibilitar os primeiros passos no entendimento da programação paralela de propósito geral em GPU. A GPU 2 é uma placa gráfica desenvolvida pela NVIDIA dedicada exclusivamente ao processamento científico altamente paralelo que transforma a estação de trabalho em um computador de alta performance, superior à de um pequeno cluster. A GPU 2 utilizada foi adquirida pelo Laboratório de Computação de Alto Desempenho (LCAD) do Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo em São Carlos o qual autorizou a realização dos testes na devida GPU.

Tabela 5.1 – Características das GPUs utilizadas no desenvolvimento do trabalho.

	GPU 1	GPU 2
Processador	NVIDIA GeForce 9800 GT	NVIDIA TESLA C1060
Número de núcleos	112	240
Frequência de clock (GHz)	1.5	1.3
Memória – off chip	1 GB GDDR3	4 GB GDDR3
Largura de banda (GB/s)	57.6	102
Versão do kernel do Linux	Ubuntu 8.04 Karmic Koala (2.6.24-23) 32 bits	Ubuntu 9.10 Karmic Koala (2.6.31-14) 64 bits
Versão do CUDA toolkit	2.1	2.1

Figura 5.1 – GPU 1 – GeForce 9800 GT.

Figura 5.2 – GPU 2 – TESLA C1060.

A Tabela 5.2 apresenta as especificações das CPUs *multi-cores* utilizadas para o processamento em serial e paralelo utilizando a API Pthreads.

Tabela 5.2 – Característica das CPUs utilizadas no desenvolvimento do trabalho.

	CPU 1	CPU 2	CPU 3
Processador	Intel Core 2 Duo (E4400)	Intel Core 2 Quad (Q8200)	Intel Core i7 975 Extreme Edition
Número de núcleos	2	4	4
Threads	2	4	8
Frequência de clock (GHz)	2	2.5	3.33
Memória RAM	1 GB DDR2	4 GB DDR2	12 GB DDR3
Versão do kernel do Linux	Ubuntu 9.10 Karmic Koala (2.6.31-20) 32 bits	Debian Lenny (2.6.26-2-amd64) 64bits	Ubuntu 9.10 Karmic Koala (2.6.31-14) 64 bits

Figura 5.3 – CPU 1 – Core 2 Duo.

Figura 5.4 – CPU 2 – Core 2 Quad.

Figura 5.5 – CPU 3 – Core i7 975.

O compilador utilizado para os códigos desenvolvidos em CUDA foi o *nvcc* - 2.1. Este compilador fornecido pela NVIDIA é regido pela invocação de diversas ferramentas para diferentes estágios de compilação. A etapa inicial de compilação começa pelo tratamento que é dado aos marcadores introduzidos pelo CUDA, que estão presentes em variáveis e funções

ao longo do código e são tratados de maneira especial pelo compilador, que em uma primeira etapa, separa os códigos específicos de cada arquitetura, neste caso da GPU (*device*) e do CPU (*host*). O resultado dessa separação é código C puro para o *host* e um objeto **cubin** (formato binário) para o *device* (NVIDIA, 2007a).

O compilador para os códigos desenvolvidos para a CPU foi o *gcc 4.3* e os códigos que utilizam a API Pthreads incluem o comando *-lpthread*.

Todos os tempos medidos correspondem ao tempo total de processamento do programa após 10 iterações do método de Newton, e foram cronometrados com a função *time* do Linux.

5.2 Aplicação em um problema térmico inverso

Esta seção visa a verificação dos códigos desenvolvidos aplicados ao problema tomográfico térmico inverso e sua devida comparação de desempenho apresentada com o programa serial desenvolvido por Brandi (2010).

Considerando um domínio bidimensional quadrado de lados unitários com $k = 1$ em todo o domínio e os outros parâmetros conforme mostra a Figura 5.6. A equação (3.1) em coordenadas cartesianas foi discretizado pelo método de elementos finitos. A condição de contorno mista (3.3) foi assumida em todos os lados do quadrado. O domínio foi discretizado em uma malha computacional de 31×31 pontos, gerando 961 nós igualmente espaçados ($\delta x = \delta y = 0,033$) e 1800 elementos triangulares. Ambas as variações de temperatura e condutividade térmica foram assumidas lineares no interior dos elementos.

Figura 5.6 – Geometria do problema simulado numericamente

O problema de referência foi definido a partir da distribuição de h_{real} no lado inferior do quadrado ($y=0$) a ser reconstruída através da minimização da função de erro. Mais especificamente, ao fixar h_{real} obtêm-se as temperaturas que seriam medidas do lado superior ($y=1$) como resposta a aplicação do fluxo de calor no mesmo lado. Na medida em que o erro (3.11) diminui durante o processo de otimização as temperaturas do problema de referência e, conseqüentemente, o h_{num} converge para o h_{real} .

Como analisado em Brandi (2010) o problema inverso é intrinsecamente mal condicionado e o problema de otimização correspondente é patológico. Mais especificamente, partindo das perturbações de h_{real} , a sequência dada pela fórmula de recorrência de Newton (3.18) convergiria monotonicamente se a superfície de erro fosse convexa em toda a parte. Mas como apresentado em Brandi (2010), as superfícies de otimização são patológicas, ou seja, elas contêm estruturas como vale estreito, múltiplos mínimos locais, platôs, etc., e, conseqüentemente, é esperado que as trajetórias de convergência correspondentes sejam erráticas.

Para o estudo da convergência foram construídas três trajetórias de convergência iniciando de uma versão perturbada de h_{real} e, progressivamente, reduzindo-se a intensidade da perturbação. Matematicamente isto pode ser expresso por

$$h_{inicial} = h_{real} + \delta, \quad (5.1)$$

em que δ corresponde a um vetor de perturbação aleatórias uniformes centrados em torno do zero, e um desvio padrão definido previamente. Isto garante que a estimativa inicial em (3.18) é tomada progressivamente mais próxima da solução correta de h_{real} .

O primeiro caso analisado foi para h_{real} constante e igual a 5 e as perturbações são geradas de acordo com as seguintes fórmulas,

$$h = 5 + \begin{cases} \delta \in [-5.0, +5.0], & \text{caso 1.1;} \\ \delta \in [-2.0, +2.0], & \text{caso 1.2;} \\ \delta \in [-0.15, +0.15], & \text{caso 1.3;} \end{cases} \quad (5.2)$$

No segundo caso, h_{real} é uma função cossenoidal e as perturbações são geradas de acordo com as seguintes fórmulas,

$$h = 1 + \cos(2\pi x) + \begin{cases} \delta \in [-2.0, +2.0], & \text{caso 2.1;} \\ \delta \in [-1.0, +1.0], & \text{caso 2.2;} \\ \delta \in [-0.5, +0.5], & \text{caso 2.3;} \end{cases} \quad (5.3)$$

No terceiro caso, h_{real} é uma função triangular e as perturbações são geradas de acordo com as seguintes fórmulas,

$$h = \begin{cases} 2 - 2(x/15), & x \leq 15 \\ -2 + 2(x/15), & x > 15 \end{cases} + \begin{cases} \delta \in [-2.0, +2.0], & \text{caso 3.1;} \\ \delta \in [-1.0, +1.0], & \text{caso 3.2;} \\ \delta \in [-0.5, +0.5], & \text{caso 3.3.} \end{cases} \quad (5.4)$$

Nas seções seguintes são apresentados os testes realizados com os casos descritos com os códigos paralelizados.

5.3 Aplicação com o NVIDIA CUDA

A seguir apresentam-se os resultados obtidos com o código utilizando o processamento paralelo das GPUs. A fim de ser mais preciso todos os programas tiveram seu tempo total de processamento medido dez vezes (ao final de 10 iterações do método de Newton) com a função *time* do Linux, todas as vezes compilado do mesmo modo (para um

programa escrito em C para CUDA utilizando o compilador *nvcc*), afim de se extrair o tempo médio e o desvio padrão do tempo que cada programa levou para executar.

Os programas paralelos foram executados nas GPU 1 e GPU 2 como descrito anteriormente e foram comparados com o programa serial executado na CPU 1. Como o grande gargalo de processamento do problema térmico inverso analisado encontra-se no cálculo e montagem da matriz Hessiana (cerca de 98% do tempo de processamento) considerou-se o tempo de processamento da parte serial dos programas executados nas CPUs como desprezíveis.

É importante ressaltar que os códigos desenvolvidos para a GPU utilizam precisão simples, e como se analisou na seção 4.1.3 não apresentou problemas na convergência do problema térmico direto em questão. Entretanto na resolução do problema térmico inverso a matriz do problema é mal condicionada, mas ao observar as figuras dos casos estudados nota-se que a precisão utilizada nos cálculos não apresentou problemas se compararmos com o gráfico do código serial que utiliza precisão dupla.

Nos códigos implementados em CUDA foram montados um total de 123 sistemas lineares esparsos com dimensões 961×961 utilizando a estrutura de dados desenvolvida e transferidos para a memória global das placas gráficas, possibilitando dessa maneira a resolução dos sistemas lineares esparsos totalmente no espaço da memória da GPU. A resolução dos 123 sistemas lineares esparsos possibilita a montagem de uma linha da matriz Hessiana do problema analisado sendo que ela tem dimensão 31×31 . Os gráficos apresentados a seguir para a análise da convergência do código correspondem a resolução do problema térmico inverso após 1000 iterações do método de Newton.

Figura 5.7 – Estimativas iniciais em GPU e solução real do coeficiente de convecção, conforme a equação (5.2) caso 1.1.

Figura 5.8 – Estimativas iniciais em GPU e solução real do coeficiente de convecção, conforme a equação (5.2) caso 1.2.

Figura 5.9 – Estimativas iniciais em GPU e solução real do coeficiente de convecção, conforme equação (5.2) caso 1.3.

Figura 5.10 – Estimativas iniciais em GPU e solução real do coeficiente de convecção, conforme a equação (5.3) caso 2.1.

Figura 5.11 – Estimativas iniciais em GPU e solução real do coeficiente de convecção, conforme a equação (5.3) caso 2.2.

Figura 5.12 – Estimativas iniciais em GPU e solução real do coeficiente de convecção, conforme a equação (5.3) caso 2.3.

Figura 5.13 – Estimativas iniciais em GPU e solução real do coeficiente de convecção, conforme a equação (5.4) caso 3.1.

Figura 5.14 – Estimativas iniciais em GPU e solução real do coeficiente de convecção, conforme a equação (5.4) caso 3.2.

Figura 5.15 – Estimativas iniciais em GPU e solução real do coeficiente de convecção, conforme a equação (5.4) caso 3.3.

Analisando as figuras dos casos testes observa-se que as soluções dos sistemas lineares no espaço da memória da GPU utilizando precisão simples apresentaram uma convergência satisfatória se comparado com o programa serial original utilizando precisão dupla.

A Figura 5.16 apresenta os tempos de processamento obtidos com os processamentos em GPU, e na Figura 5.17 temos a análise dos *speedups* alcançados com o uso das placas gráficas. Também cabe aqui ressaltar a dificuldade em se trabalhar com a resolução de sistemas lineares de matrizes esparsas em GPU, como discutidas anteriormente os resultados obtidos se encontram em uma mesma faixa que o observado na literatura (BUATOIS; CAUMON; LÉVY, 2008).

No APÊNDICE D encontra-se a tabela com todos os tempos de processamento e os desvios padrões dos casos analisados.

Figura 5.16 – Tempo de processamento dos casos analisados serial e paralelo em GPU.

A Figura 5.17 mostra os *speedups* obtidos com o processamento em paralelo nas GPUs do método dos PCGs em comparação com a execução do programa desenvolvido por Brandi (2010) em um computador com a CPU 1 serial.

Figura 5.17 – *Speedups* obtidos com o processamento em GPU.

Como se pode observar nas figuras acima através da implementação do método PCG em GPU, constatou-se ganhos nos tempos de execução no processo de resolução do problema térmico inverso e com uma boa convergência se comparado ao programa original desenvolvido em Brandi (2010) tornando a aplicação desenvolvida interessante para trabalhos futuros.

Também vale mencionar a escalabilidade obtida com o código desenvolvido, podendo ser utilizada em placas gráficas da NVIDIA com a tecnologia CUDA com a quantidade memória adequada. Ao utilizar o código desenvolvido na GPU 1 na GPU 2 (sem nenhuma alteração no código) obteve-se um ganho de desempenho em torno de 1,5 vezes, isto se deve ao fato da GPU 2 como discutido ser uma placa gráfica dedicada ao processamento científico com maior poder de processamento.

A fim de analisar se a utilização de precisão simples na resolução dos sistemas lineares esparsos na GPU afetou o número de iterações para a convergência do algoritmo do método PCG, comparou-se o número de iterações necessário para o método empregado no espaço da memória global da GPU utilizando precisão simples com o método utilizado originalmente por Brandi (2010) no seu trabalho (método do PBCG com precisão dupla).

Os números de iterações referem-se à resolução de um sistema linear esparsa utilizado no cálculo da matriz Hessiana do problema.

Figura 5.18 – Números de iterações para convergência de um sistema linear esparsa utilizando os métodos PBCG (precisão dupla em CPU) e o PCG (precisão simples em GPU).

Analisando a Figura 5.18 constatou-se que o método PCG implementado em GPU converge para a solução do sistema com menos iterações que o método PBCG, isso se deve ao fato que o método do PBCG ter a característica de resolver qualquer sistema linear (para mais detalhe sobre o método PBCG utilizado consultar o APÊNDICE C) e o método PCG resolver apenas sistemas lineares simétricos e positivo-definidos que é o caso dos sistemas lineares do problema. A Figura 5.19 apresenta a diferença obtida da resolução de um sistema linear esparsa do problema utilizando precisão simples e dupla ficando em torno da quinta ou sexta casa decimal.

Figura 5.19 – Diferença da resolução de um sistema linear esparso do problema utilizando precisão simples e dupla.

5.4 Aplicação com o Pthreads

A seguir apresentam-se os resultados obtidos com o código utilizando o poder de processamento paralelo das CPUs *multi-cores*. Para a implementação do código paralelizado utilizando Pthreads partiu-se de um código seqüencial na linguagem C desenvolvido no trabalho de Brandi (2010).

A fim de ser mais justo todos os programas tiveram seu tempo de processamento medido dez vezes (ao final de 10 iterações do método de Newton) com a função *time* do Linux, com todas as vezes compilado do mesmo modo (para um programa escrito em C utilizando o compilador *gcc*, com o objetivo de se extrair o tempo médio e o desvio padrão do tempo que cada programa levou para executar).

Os códigos paralelos foram executados nas CPU 1, CPU 2 e CPU 3 todas com 20 *threads* e foram comparados com a execução do código serial executados respectivamente nos mesmos processadores a fim obter o ganho de performance com a utilização da paralelização.

Os gráficos para a análise da convergência do código correspondem a resolução do problema térmico inverso após de 1000 iterações do método de Newton.

Figura 5.20 – Estimativas em Pthreads e solução real do coeficiente de convecção, conforme a equação (5.2) caso 1.1.

Figura 5.21 – Estimativas em Pthreads e solução real do coeficiente de convecção, conforme a equação (5.2) caso 1.2.

Figura 5.22 – Estimativas em Pthreads e solução real do coeficiente de convecção, conforme a equação (5.2) caso 1.3.

Figura 5.23 – Estimativas iniciais em Pthreads e solução real do coeficiente de convecção, conforme a equação (5.3) caso 2.1.

Figura 5.24 Estimativas iniciais em Pthreads e solução real do coeficiente de convecção, conforme a equação (5.3) caso 2.2.

Figura 5.25 – Estimativas iniciais em Pthreads e solução real do coeficiente de convecção, conforme a equação (5.3) caso 2.3.

Figura 5.26 – Estimativas iniciais em Pthreads e solução real do coeficiente de convecção, conforme a equação (5.4) caso 3.1.

Figura 5.27 – Estimativas iniciais em Pthreads e solução real do coeficiente de convecção, conforme a equação (5.4) caso 3.2.

Figura 5.28 – Estimativas iniciais em Pthreads e solução real do coeficiente de convecção, conforme a equação (5.4) caso 3.1.

Analisando os gráficos de convergência dos casos testes observa-se que as soluções dos sistemas lineares utilizando a paralelização em Pthreads apresentaram uma convergência satisfatória se considerar o programa serial original desenvolvido por Brandi (2010).

Figura 5.29 – Tempo de processamento dos casos analisados serial e paralelo em CPU.

No APÊNDICE E encontra-se a tabela com todos os tempos de processamento e os desvios padrões dos casos analisados utilizando a paralelização em Pthreads.

Na Figura 5.30 é possível observar os *speedups* obtidos com o processamento em paralelo nas CPUs *multi-cores* do cálculo e montagem da matriz Hessiana do problema em comparação com a execução do programa serial desenvolvido por Brandi (2010).

Figura 5.30 – *Speedups* obtidos com o processamento paralelo Pthreads.

Como observado a paralelização utilizando Pthreads se apresentou eficiente chegando a oferecer um *speedup* de quase 4 vezes se comparada com a versão sem paralelização,

atingindo o limite da quantidade do número de *cores* físicos dos processadores e com uma convergência da solução do problema extremamente satisfatória.

No entanto ainda é possível melhorar ainda mais o desempenho de processamento nas CPUs *multi-cores* com a utilização de *flags* de otimização que são usadas para customizar e otimizar aplicações quando as construímos pelas fontes. Como se utiliza o compilador *gcc* (GNU *Compiler Collection*) foram empregadas as *CFLAGS* que são *flags* do compilador C. *CFLAGS* são geralmente usadas para especificar a arquitetura de seu computador, bem como o tipo de CPU usada e qualquer outra opção especial que se queira habilitar ou desabilitar. Esta informação é importante para o GCC que diz a ele exatamente como customizar as instruções de montagem que ele cria através do código-fonte da aplicação. Tecnicamente o GCC pode rodar sem nenhuma *CFLAG*, já que são opções extras.

Como o objetivo deste trabalho é otimizar o tempo de processamento do problema térmico inverso, foi utilizado a *flag* de otimização *-O2*. Este é o nível de otimização mais usado para pacotes e distribuições no mundo Linux e para o kernel Linux ela é considerada segura não apresentando alterações na convergência do problema.

Na Figura 5.31 figura observa-se os tempos de processamento dos testes realizados o qual é possível notar o ganho de tempo obtido com as versões paralelizadas com *flags* de otimização.

A fim de obter os tempos de processamento do problema paralelizado e com o auxílio da *flag* de otimização foram realizados os mesmos testes que os descritos anteriormente a menos dos testes de convergência que não apresentaram alterações significativas.

Figura 5.31 – Tempo de processamento dos casos analisados serial e paralelo com auxílio de *flags* de otimização.

A seguir na Figura 5.32 nota-se os *speedups* obtidos nos casos analisados utilizando Pthreads com o auxílio da *flag* de otimização.

Figura 5.32 – *Speedups* obtidos com o processamento paralelo Pthreads com *flag* de otimização.

Como se pode observar a paralelização utilizando Pthreads com o auxílio da *flag* de otimização se apresentou extremamente eficiente chegando a oferecer um *speedup* de quase

12 vezes se comparada com a versão serial sem nenhum tipo de otimização e com uma convergência aceitável da solução do problema.

Existe ainda a possibilidade de melhorar o tempo de processamento do problema térmico com o uso de *flags* de otimização mais agressivas e que explorem mais das características da CPU e também com a utilização de novos compiladores próprios como o desenvolvido pela Intel, pois os processadores utilizados são fabricados por ela.

A paralelização utilizando Pthreads também possibilitou verificar a escalabilidade apresentada pelos processadores utilizados. As CPU 1, CPU 2 e CPU 3 foram executadas com uma *thread* (versão serial) e foram comparadas, possibilitando o ganho obtido apenas mudando o processador.

Figura 5.33 – Escalabilidade das CPUs com o processamento com uma *thread*.

O fato de apenas aceder o poder de processamento das CPUs utilizando CPUs mais modernas e com paralelismos nativos possibilitou um *speedup* de até 2,5 vezes se comparando o tempo de processamento do caso 2.2 na CPU 1 e na CPU 3 como observado na Figura 5.33.

CAPÍTULO 6 CONCLUSÕES E PERSPECTIVAS

A idéia principal que serviu de impulso para o início dos trabalhos desta dissertação foi acelerar a resolução de um problema térmico inverso desenvolvido por Brandi (2010) tirando proveito da capacidade computacional da GPU e de CPUs *multi-cores*.

O GPU moderno pode ser encarado como um co-processador programável, para processamento genérico, com enorme capacidade de computação e revela-se especialmente adequado para a implementação de operações paralelas que tenham alta intensidade aritmética. Deste modo, pode-se encarar a programação genérica em GPU como uma boa abordagem para a aceleração de algoritmos computacionalmente muito exigentes, através de fases cujas operações sejam paralelizáveis, segundo o modelo SIMD do hardware do GPU.

A plataforma CUDA simplifica muito a aprendizagem da programação em GPU, bem como permite tirar bom proveito do poder computacional do GPU. O fato do seu modelo de programação consistir em uma extensão à linguagem C, familiar a uma grande quantidade de programadores torna seu aprendizado mais simples.

Os *speedups* apresentados no Capítulo 5 referentes à implementação em CUDA são relativamente pequenos se comparados a outros trabalhos utilizando processamento em GPU, mas vale salientar que cada problema abordado permite estratégias diferentes que produzem *speedups* diferentes. O trabalho com matrizes esparsas em GPU é uma tarefa extremamente árdua, devido ao seu baixo grau de intensidade aritmética, ou seja, trabalhar com matrizes esparsas implica em ter uma grande quantidade de acesso a memória.

Comparando-se os *speedups* obtidos com a GPU neste trabalho com outros trabalhos encontrados na literatura (BOLZ et al., 2003; BUATOIS; CAUMON; LÉVY, 2008; CEVAHIR; NUKADA; MATSUOKA, 2009; WIGGERS et al., 2007) nota-se que se encontram em uma mesma faixa de ganho de processamento, tornando-se interessante a sua aplicação em outros problemas, pois para problemas de ordem de execução de dias com esse *speedup* apresentado este poderia cair para horas.

Outro fator importante é que a implementação em GPU ainda pode ser otimizada através de novos *layouts* de manipulação da memória, principalmente se utilizar memória de

texturas e aperfeiçoar o uso de memória através da minimização de transferência de dados *host/device* e do acesso coalescido à memória global da GPU.

Com respeito a convergência ambas as implementações foram satisfatórias comparado ao código original desenvolvido por Brandi (2010), observando que a implementação em GPU utilizou precisão simples em todos os seus testes e que o método do PCG com o pré-condicionador de Jacobi apresentou uma convergência mais rápida para a solução do sistema comparado como o método do PBCG.

A paralelização utilizando Pthreads proporcionou resultados extremamente satisfatórios sendo sua aplicação interessante para simulações futuras com uma malha mais refinada e problemas mais complexos que os utilizados no estudo de caso desta dissertação. A utilização de *flags* de otimização e a de compiladores apropriados possibilitam um ganho substancial na redução do tempo computacional do problema.

É importante salientar que os resultados obtidos neste trabalho não esgotam as possibilidades de exploração da tecnologia CUDA sendo este apenas um primeiro passo na incessante busca por desempenho computacional. A utilização de máquinas multiprocessadas – com múltiplas GPUs – representa um enorme potencial a ser explorado. Além disso, a utilização do CUDA com a tecnologia MPI parece bastante promissora ou até mesmo a utilização da nova geração das placas gráficas da NVIDIA que permitem a execução de *kernels* em paralelo, ganhando assim, mais um nível de paralelismo. Certamente, este é um caminho a ser explorado por trabalhos futuros.

Outra opção importante é a utilização de códigos implementados em OpenCL que executariam na GPU e na CPU *multi-cores*.

A ampliação da capacidade computacional e redução de custos associados a estas tecnologias são fatores importantes que podem impulsionar os grupos de pesquisas de diversas áreas em um salto qualitativo e quantitativo em seus trabalhos.

A seguir apresentam-se as perspectivas futuras que estão divididas em três categorias:

➤ Melhorias nos códigos já implementados

- Desenvolver uma estrutura de dados para o armazenamento da matriz esparsa que favoreça o acesso coalescido de memória (GPU);
- Utilizar memória de textura (GPU);
- Testar novos pré-condicionadores no método do PCG.
- Substituir o método do PBCG pelo PCG (CPU);
- Programação em OpenCL (GPU) (CPU);

- Testar desempenho dos códigos em OpenCL nas placas gráficas da ATI Radeon e na NVIDIA Fermi;
- Infraestrutura de comunicação
 - Implementação de duas ou mais GPUs em um computador;
 - Implementação de um cluster heterogêneo com troca de mensagens por MPI (CUDA - MPI) (JACOBSEN; THIBAUT; SENOCAR, 2010);
- Demandas Específicas
 - Implementação de Algoritmos Genéticos (AG) na GPU (HARDING; BANZHAF, 2009; YU; CHEN; PAN, 2005);
 - Desenvolvimento de algoritmos híbridos com métodos estocásticos (AG) e determinísticos acelerados em GPU e CPUs *multi-cores*;
 - Implementação de redes neurais na GPU (NISSEN, 2010).

REFERÊNCIAS BIBLIOGRÁFICAS

- ALMASI, G.S.; GOTTLIEB, A. (1990). Review of Highly parallel computing **IBM Syst. J.**, v.29, n.1, p.165-166.
- _____. (1994). **Highly parallel computing**. 2ed. Redwood City: Benjamin/Cummings.
- AMDAHL, G.M. (1967). Validity of single processor approach to achieving large scale computing capability. **AFIPS Conference Proceedings**, v.30, p.483-485.
- ASSAN, R.B. (1999). **Métodos dos Elementos Finitos: Primeiros passos**. Campinas: Ed. Unicamp.
- BARNEY, B. (2010a). **Introduction to Parallel Computing**. Disponível em: <https://computing.llnl.gov/tutorials/parallel_comp/>. Acesso em: 04/13/2009.
- _____. (2010b). **OpenMP**. Disponível em: <<https://computing.llnl.gov/tutorials/openMP/>>. Acesso em: 02/03/2010.
- _____. (2010c). **POSIX Thread programming**. Disponível em: <<https://computing.llnl.gov/tutorials/pthreads/>>. Acesso em: 06/03.
- BARRETT, R. et al. (1994). **Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods**. 2 ed. Philadelphia: SIAM.
- BECKER, E.B.; CAREY, G.F.; ODEN, J.T. (1981). **Finite Elements: An Introduction**. Englewood Cliffs: Prentice-Hall.
- BELL, N.; GARLAND, M. (2008). Efficient Sparse Matrix-Vector Multiplication on CUDA. **NVIDIA Technical Report**, n.NVR-2008-004.
- BHANDARKAR, M. et al. (2010). **NAMD Scalable Molecular Dynamics**. Disponível em: <<http://www.ks.uiuc.edu/Research/namd/>>. Acesso em: 04/01/2010.
- BOKAR, J.C.; ÖZISIK, M.N. (1995). An inverse analysis for estimating the time-varying inlet temperature in laminar flow inside a parallel plate duct. **International Journal of Heat and Mass Transfer**, v.38, n.1, p.39-45.
- BOLZ, J. et al. (2003). Sparse matrix solvers on the GPU: Conjugate Gradients and Multigrid. **ACM Transactions on Graphics**, v.22, n.3, p.917-924.
- BORCEA, L. (2002). Eletrical impedance tomography. **Inverse Problems**, v.18, n.6, p.R99-R136.
- BRANDI, A.C. (2010). **Desenvolvimento de uma Técnica não Intrusiva de Medição do Coeficiente de Convecção - Solução do Problema Térmico Inverso**. 143 pp. (Doutorado) - Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2010.

- BRANDI, A.C.; ANSONI, J.L.; SELEGHIM JR, P. (2009). Reconstruction of the convection coefficient from non-intrusive measurements: Regularization of the inverse problem by the truncated singular value decomposition method. In: 20th International Congress of Mechanical Engineering, 2009, Gramado, RS, Brazil. Gramado, RS, Brazil: ABCM. p.1-8
- BROOKWOOD, N. (2010). **AMD Fusion™ Family of APUs: Enabling a Superior, Immersive PC Experience** Disponível em: <http://sites.amd.com/us/Documents/48423B_fusion_whitepaper_WEB.pdf>. Acesso em: 04/04/2010.
- BUATOIS, L.; CAUMON, G.; LÉVY, B. (2008). Concurrent number cruncher - A GPU implementation of a general sparse linear solver. **International Journal of Parallel, Emergent and Distributed Systems**, p.1-18.
- BUCK, I. et al. (2004). Brook for GPUs: Stream computing on graphics hardware. In: International Conference on Computer Graphics and Interactive Techniques, 2004, New York. New York: ACM New York. p.777-786
- BUTENHOF, D.R. (1997). **Programming with POSIX Threads**. Boston: Addison-Wesley Longman Publishing.
- CAMPOS, G.P. (2004). **Contribuição ao desenvolvimento de técnicas de visualização térmica para monitoração de processos envolvendo fluidos multifásicos**. 90 pp. (Doutorado) - Universidade de São Paulo, São Carlos, 2004.
- CAMPOS VELHO, H.F. (2001). Problemas inversos: conceitos básicos e aplicações. In: ENCONTRO DE MODELAGEM COMPUTACIONAL, 2001, Nova Friburgo. Nova Friburgo: p.17
- CEVAHIR, A.; NUKADA, A.; MATSUOKA, S. (2009). Fast Conjugate Gradients with Multiple GPUs. In: ICCS '09: Proceedings of the 9th International Conference on Computational Science, 2009, Baton Rouge, LA. Baton Rouge, LA: Springer-Verlag. p.893-903
- CHAPMAN, B.; JOST, G.; PASS, R.V.D. (2008). **Using OpenMP: Portable Shared Memory Parallel Programming**. Cambridge: The MIT Press.
- COLAÇO, M.J.; ORLANDE, H.R.B.; DULIKRAVICH, G.S. (2004). Inverse and optimization problems in heat transfer. In: BRAZILIAN CONGRESS OF THERMAL SCIENCES AND ENGINEERING, 2004, Rio de Janeiro. Rio de Janeiro: Anais...
- CORRIGAN, A. et al. (2009). Running Unstructured Grid Based CFD Solvers on Modern Graphics Hardware. In: 19th AIAA Computational Fluid Dynamics, 2009, San Antonio, Texas. San Antonio, Texas: American Institute of Aeronautics and Astronautics. p.Paper 2009-4001
- CULLER, D.; SINGH, J.P.; GUPTA, A. (1998). **Parallel Computer Architecture: A Hardware / Software Approach**. San Francisco: Morgan Kaufmann.
- CUNHA, M.C.C. (2000). **Métodos numéricos**. Campinas: Ed. Unicamp.

- FAYAD, M.E.; SCHMIDT, D.C. (1997). Object-oriented Application frameworks. **Communications of the ACM**, v.40, n.10, p.10.
- FLYNN, M.J. (1972). Some computer organizations and their effectiveness. **IEEE Transactions on Computers**, v.C-21, n.9, p.948-960.
- FORTUNA, A.O. (2000). **Técnicas computacionais para dinâmicas dos fluidos: conceitos básicos e aplicações**. São Paulo: EDUSP.
- FRIEDMAN, R. (2010). **The OpenMP API specification for parallel programming**. Disponível em: <<http://openmp.org/wp/>>. Acesso em: 02/12/2010.
- GUSTAFSON, J. (1988). Reevaluating Amdah's law. **Communications of the ACM**, v.31, n.5, p.532-533.
- HADAMARD, J. (1902). Sur les problemes aux derives partielles et leur signification physique. **Princeton University Bulletin**, v.13, p.49-52.
- HALFHILL, T.R. (2008). Parallel processing with CUDA: Nvidia's High-Performance Computing Platform Uses Massive Multithreading. **Microprocessor Report**, v.January 2008, p.1-8.
- HARDING, S.L.; BANZHAF, W. (2009). Distributed genetic programming on GPUs using CUDA. In: Workshop on Parallel Architectures and Bioinspired Algorithms, 2009, Raleigh, USA. Raleigh, USA: BADS.
- HARRIS, M.J. (2002). **General-Purpose Computation on Graphics Hardware**. Disponível em: <<http://gpgpu.org/>>. Acesso em: 03/05/2010.
- _____. (2003). **Real-Time Cloud Simulation and Rendering**. 173 pp. (Doctor) - University of North Carolina at Chapel Hill Chapel Hill, 2003.
- HESTENES, M.R.; STIEFEL, E. (1952). Methods of Conjugate Gradients for solving linear systems. **Journal of Research of the National Bureau of Standards**, v.49, n.6, p.409-436.
- HUANG, C.H.; YUAN, I.C.; AY, H. (2003). A three-dimensional inverse forced convection problem in estimating surface heat flux by conjugate gradient method. **International Journal of Heat and Mass Transfer**, v.46, n.19, p.3629-3638.
- JACOBSEN, D.A.; THIBAUT, J.C.; SENOCAR, I. (2010). An MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters. In: 48th AIAA Aerospace Sciences Meeting, 2010, Orlando, Florida. Orlando, Florida: New Horizons Forum and Aerospace. p.1-17
- JANUSZEWSKI, M.; KOSTUR, M. (2009). **Accelerating numerical solution of Stochastic Differential Equations with CUDA**. Disponível em: <<http://arxiv.org/abs/0903.3852v2>>. Acesso em: 05/02/2010.
- KASIM, H. et al. (2008). Survey on Parallel Programming Model. **NPC '08: Proceedings of the IFIP International Conference on Network and Parallel Computing**, v.5245, p.266--275.

- KHRONOS, G. (2010). **OpenCL: Parallel Computing for Heterogeneous Devices**. Disponível em: <<http://www.khronos.org/opencl/>>. Acesso em: 04/02/2010.
- KRÜGER, J.; WESTERMANN, R. (2003). Linear Algebra Operators for GPU Implementation of Numerical Algorithms. **ACM Transactions on Graphics**, v.22, p.908-916.
- LAGIER, G.L.; LEMONNIER, H.; COUTRIS, N. (2004). A numerical solution of the linear multidimensional unsteady inverse heat conduction problem with the boundary element and the singular value decomposition. **International Journal of Thermal Sciences**, v.43, n.2, p.145-155.
- LÁNYI, S. (1998). Analysis of linearity errors of inverse capacitance position sensors. **Measurement Science and Technology**, v.9, n.10, p.1757-1764.
- LEJDFORS, C.; OHLSSON, L. (2006). Implementing an embedded GPU language by combining translation and generation. In: ACM symposium on Applied computing, 2006, ACM New York. p.1610-1614
- LEWIS, B.; BERG, D.J. (1996). **PThreads Primer: A Guide to Multithreaded Programming**. Mountain View: SunSoft Press.
- MALISKA, C.R. (1995). **Transferência de Calor e Mecânica dos Fluidos Computacional**. Rio de Janeiro: Livros Técnicos e Científicos.
- MARCARI JUNIOR, E. (2002). **Classificação e comparação de ferramentas para análise de desempenho de sistemas paralelos**. 113 pp. (Mestrado) - Universidade Estadual Paulista, Ilha Solteira, 2002.
- MENDONÇA, P.D.T.R. (2005). **Transferência de Calor pelo MEF**. Disponível em: <<http://www.grante.ufsc.br/download/TrnsCal.pdf>>. Acesso em: 15/10/2008.
- NICKOLLS, J. et al. (2008). Scalable Parallel Programming with CUDA. **ACM Transactions on Graphics**, v.6, n.2, p.1-16.
- NISSEN, S. (2010). **Neural Networks on the GPU**. Disponível em: <http://leenissen.dk/fann/html_latest/files2/gpu-txt.html>. Acesso em: 06/07/2010.
- NOCEDAL, J.; WRIGHT, S.J. (1999). **Numerical optimization**. New York: Springer.
- NOËL, P.B. et al. (2008). Clinical Evaluation of GPU-Based Cone Beam Computed Tomography. In: MICCAI Workshop: High-Performance Medical Image Computing and Computer Aided Intervention, 2008, New York. New York: HP-MICCAI.
- NVIDIA (2007a). **The CUDA Compiler Driver: NVCC**. Disponível em: <<http://sbel.wisc.edu/Courses/ME964/2008/Documents/nvccCompilerInfo.pdf>>. Acesso em: 03/05/2008.
- _____. (2007b). **NVIDIA CUDA - Programming Guide 1.1**. Disponível em: <http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf>. Acesso em: 05/05/2008.

- _____. (2007c). **NVIDIA Tesla: GPU computing Technical Brief**. Disponível em: <http://www.nvidia.com/docs/IO/43395/tesla_technical_brief.pdf>. Acesso em: 03/03/2010.
- _____. (2008). **NVIDIA CUDA - Programming Guide 2.2** Disponível em: <http://developer.nvidia.com/object/gpu_programming_guide.html>. Acesso em: 04/02/2009.
- _____. (2009a). **NVIDIA's Next Generation CUDA Compute Architecture: Fermi**. Disponível em: <http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf>. Acesso em: 12/12/2009.
- _____. (2009b). **NVIDIA CUDA C Programming: Best Practices Guide** Disponível em: <http://www.serc.iisc.ernet.in/ComputingFacilities/systems/Tesla_Doc/NVIDIA_CUDA_Best_Practices_Guide_2.3.pdf>. Acesso em: 09/06/2009.
- _____. (2010). **NVIDIA CUDA C - Programming Guide 3.1**. Disponível em: <http://developer.nvidia.com/object/cuda_3_1_downloads.html>. Acesso em: 07/10/2010.
- OLIVEIRA, J. (2006). **Desenvolvimento de um sensor de temperatura inteligente - compensação em tempo real dos efeitos de convecção, acumulação e radiação**. 82 pp. (Doutorado) - Universidade de São Paulo, São Carlos, 2006.
- PREIS, T. et al. (2009). GPU accelerated Monte Carlo simulation of the 2D and 3D Ising model. **Journal of Computational Physics**, v.228, n.12, p.4468-4477.
- PRESS, W.H. et al. (1992). **Numerical Recipes in C: The Art of Science Computing**. New York: Cambridge University Press.
- ROLNIK, V.P. (2003). **Contribuição ao desenvolvimento de uma técnica de reconstrução tomográfica para sondas de visualização direta**. 128 pp. (Doutorado) - Universidade de São Paulo, São Carlos, 2003.
- SAAD, Y. (2003). **Iterative methods for sparse linear systems**. 2. Philadelphia: SIAM.
- SEILER, L. et al. (2008). Larrabee: A Many-Core x86 Architecture for Visual Computing. **ACM Transactions on Graphics**, v.27, n.3, p.18:1-18:15.
- SELEGHIM JR, P.; MILIOLI, F.E. (2001). Improving the determination of bubble size histograms by employing wavelet de-noising techniques. **Powder Technology**, v.115, n.2, p.114-123.
- SHARMA, M.D. (2002). Group velocity along general direction in a general anisotropic medium. **International Journal of Solids and Structures**, v.39, n.12, p.3277-3288.
- SHEWCHUK, J.R. (1994). **An introduction to the conjugate gradient method without the agonizing pain**. Disponível em: <<http://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf>>. Acesso em: 07/03/2008.
- SILBERSCHATZ, A.; GALVIN, P.B.; GAGNE, G. (2002). **Operating System Concepts**. New York: John Wiley & Sons, Inc.

SILVA NETO, A.J.; MOURA NETO, F.D. (2005). **Problemas inversos: conceitos fundamentais e aplicações**. Rio de Janeiro: Editora da Universidade do Estado do Rio de Janeiro.

SILVA NETO, A.J.; ÖZISIK, M.N. (1995). An inverse problem of simultaneous estimation of radiation phase function, albedo and optical thickness. **Journal of Quantitative Spectroscopy and Radiative Transfer**, v.53, n.4, p.397-409.

STALLINGS, W. (2002). **Arquitetura e organização de computadores: projeto para o desempenho**. 5 ed. São Paulo: Prentice Hall.

STERLING, T.L. (2001). **Beowulf cluster computing with Linux**. Cambridge: MIT Press.

TANENBAUN, A.S.; GOODMAN, J.R. (1999). **Structured computer organization**. 4ed. Upper Saddle River, N.J.: Prentice Hall.

TIKHONOV, A.N.; ARSENIN, V.Y. (1977). **Solutions of ill-posed problems**. Washington: Winston & Sons.

TILERA (2010). **Tilera processors**. Disponível em: <<http://www.tilera.com/products/processors.php>>. Acesso em: 05/20/2010.

VOGEL, R.C. (2002). **Computational methods for inverse problems**. Philadelphia: SIAM.

WANG, J.; ZABARAS, N. (2004). A Bayesian inference approach to the inverse heat conduction problem. **International Journal of Heat and Mass Transfer**, v.47, n.17/18, p.3927-3941.

WIGGERS, W.A. et al. (2007). Implementing the conjugate gradient algorithm on multi-core systems. **Proceedings of the International Symposium on System-on-Chip (SoC 2007)**, p.11-14.

WIKILIVROS (2010). **Programação Paralela em Arquiteturas Multi-Core/Programação em Pthreads**. Disponível em: <http://pt.wikibooks.org/wiki/Programa%C3%A7%C3%A3o_Paralela_em_Arquiteturas_Multi-Core/Programa%C3%A7%C3%A3o_em_Pthreads>. Acesso em: 03/06/2010.

WILLIAMS, S. et al. (2006). The Potential of the Cell Processor for Scientific Computing. In: 3rd Conference on Computing frontiers, 2006, New York. New York: ACM New York. p.9-20

YU, Q.; CHEN, C.; PAN, Z. (2005). Parallel Genetic Algorithms on Programmable Graphics Hardware In: LNCS - Lecture Notes in Computer Science, 2005, Berlin Berlin Springer Berlin / Heidelberg. p.1051-1059

ZABARAS, N.; YANG, G.Z. (1997). A functional optimization formulation and implementation of an inverse natural convection problem. **Computer Methods in Applied Mechanics and Engineering**, v.144, n.3/4, p.245-274.

ZHAO, Y. (2008). Lattice Boltzmann based PDE solver on the GPU. **The Visual Computer**, v.24, n.5, p.323-333.

APÊNDICE A PCG NA GPU – NVIDIA CUDA

```

/*
 * File:   KERNEL_JACOBI CG_GPU.h
 * Author: Jonas L. Ansoni
 */
#ifndef _KERNEL_JACOBI CG_GPU_H
#define _KERNEL_JACOBI CG_GPU_H
/*GLOBAL FUNCTION -- PRECONDITIONED CONJUGATE GRADIENT METHOD --
PRECONDICIONADOR DE JACOBI*/
//=====
// ----- KERNEL -- CUDA CONJUGATE GRADIENT -----
//=====
//----- DEVICE - SPARSE MULTIPLICATION MATRIX x VECTOR -----
__device__ void Mat_Vet_Mul(t_lista *device_lista, float *device_vector,
int i, int cont, float *device_mulmatvet){
    float sum = 0.0f;
    if(i>= nNodes) return;
    for(int cont2 = i; cont2 < nNodes; cont2 = cont2+blockDim.x){
        for(int j = 0; j < device_lista[cont2 + (cont*nNodes)].size;
j++){
            sum += device_lista[cont2 + (cont*nNodes)].elem[j] *
device_vector[device_lista[cont2+(cont*nNodes)].coluna[j]+cont*nNodes];
        }
        device_mulmatvet[cont2]=sum;
        sum = 0.0f;
    }
}
//----- DEVICE - DIAGONAL INVERSA MATRIZ x VECTOR -----
__device__ void diagInv_sparse_mult(t_lista *device_lista, float
*device_vector, float *device_mulmatvet, int i, int cont){
    int j;
    float inverse, sum = 0.0f;
    if(i>= nNodes) return;
    for(int cont2 = i; cont2 < nNodes; cont2 = cont2 + blockDim.x){
        for(j = 0; j < device_lista[cont2 + (cont*nNodes)].size; j++){
            if(device_lista[cont2 + (cont*nNodes)].coluna[j] ==
cont2){
                inverse = 1/device_lista[cont2 +
(cont*nNodes)].elem[j];
                sum += inverse * device_vector[device_lista[cont2 +
(cont*nNodes)].coluna[j]];
            }
            inverse= 0.0f;
        }
        device_mulmatvet[cont2]=sum;
        sum = 0.0f;
    }
}
//----- DEVICE - MULTIPLICATION VECTOR x VECTOR -----
__device__ void Vet_Vet_Mul(float *device_vector_1, float *device_vector_2,
int cont, float *device_mulvetvet){
    *device_mulvetvet = 0.0f;
    for(int j = 0; j < nNodes; j ++ ){
        *device_mulvetvet += device_vector_1[j] * device_vector_2[j];
    }
}

```



```

}
//----- DEVICE - MULTIPLICATION CONST x VECTOR -----
__device__ void Mul_Vet_Const(float *device_vet, float device_const, int
cont, float *device_mulconstvet){
    for(int j = 0; j < nNodes; j ++ ){
        device_mulconstvet[j] = device_const * device_vet[j];
    }
}
//----- DEVICE - SUBTRACTION VECTOR x VECTOR -----
__device__ void Vet_Vet(float *device_vector_1, float *device_vector_2,
float device_const, int i, int cont, float *device_vetvet, int flag){
    if(i >= nNodes) return;
    if(flag){ //se flag == 1 faz subtração senao add
        for(int cont2 = i; cont2 < nNodes; cont2 = cont2+blockDim.x){
            device_vetvet [cont2] = device_vector_1[cont2] -
device_const * device_vector_2[cont2];
        }
    }
    else{
        for(int cont2 = i; cont2 < nNodes; cont2 = cont2+blockDim.x){
            device_vetvet [cont2] = device_vector_1[cont2] +
device_const * device_vector_2[cont2];
        }
    }
}
//----- DEVICE - COPY VECTOR -----
__device__ void Copy_Vet(float *device_vector_ORIGINAL, float
*device_vector_COPY, int i, int cont){
    if(i >= nNodes) return;
    for(int cont2 = i; cont2 < nNodes; cont2 = cont2+blockDim.x){
        device_vector_COPY [cont2] = device_vector_ORIGINAL[cont2];
    }
}
//===== GLOBAL =====
__global__ void Kernel_jacobiConjGradient(t_lista *kernel_lista, float
*kernel_vector_x, float *kernel_vector_b, int *its_atual, float *epsilon,
int nSist){
    int i = threadIdx.x;
    int k = blockIdx.x;
    int cont;
    __shared__ float gpu_r[nNodes], gpu_p[nNodes], gpu_z[nNodes],
gpu_u[nNodes];
    __shared__ float gpu_aux1, gpu_aux2;
    __shared__ float sum;
    __shared__ float beta, alpha;
    __shared__ int its[nSistmax];
    float tol=1e-6f;
    for(cont = k; cont < nSist; cont = cont+(gridDim.x-1)){
        // Quantidade de sistemas para ser resolvido
        // z = A*x
        Mat_Vet_Mul(kernel_lista, kernel_vector_x, i, cont, gpu_z);
        //SPARSE MATRIX MULTIPLICATION
        __syncthreads();
        // r = b - A*x = b - z
        Vet_Vet(kernel_vector_b, gpu_z, 0, i, cont, gpu_r, 1);
        //VECTOR SUBTRACTION
        __syncthreads();
        //z = inv(matriz_diag(A))*r
        diagInv_sparse_mult1(kernel_lista, gpu_r, gpu_z, i, cont);
        __syncthreads();
        // p = z

```



```

Copy_Vet (gpu_z, gpu_p, i, cont);          //COPY VECTOR
__syncthreads();
for(its[cont] = 0; its[cont] < 100000; its[cont]++){
    // u = A*p
    Mat_Vet_Mul2(kernel_lista, gpu_p, i, cont, gpu_u);
//SPARSE MATRIX MULTIPLICATION
    __syncthreads();
    Vet_Vet_Mul(gpu_r, gpu_z, cont, &gpu_aux1);
//VECTOR x VECTOR MULTIPLICATION
    __syncthreads();
    Vet_Vet_Mul(gpu_p, gpu_u, cont, &gpu_aux2);
//VECTOR x VECTOR MULTIPLICATION
    __syncthreads();
    if (i==0)
        alpha = gpu_aux1 / gpu_aux2;
    __syncthreads();
    // x = x + alpha * p
    Vet_Vet(kernel_vector_x, gpu_p, alpha, i, cont,
kernel_vector_x, 0);          //VECTOR ADD
    __syncthreads();
    //beta2 = r*z
    Vet_Vet_Mul(gpu_r, gpu_z, cont, &gpu_aux2);
//VECTOR x VECTOR MULTIPLICATION
    __syncthreads();
    // r = r - alpha*u --> r = r - alpha*A*p
    Vet_Vet(gpu_r, gpu_u, alpha, i, cont, gpu_r, 1);
//VECTOR SUBTRACTION
    __syncthreads();
    //sum = r*r
    if(i==0) Vet_Vet_Mul(gpu_r, gpu_r, cont, &sum);
//VECTOR x VECTOR MULTIPLICATION
    __syncthreads();
    //Euclidian Norm
    if(i==0)
        *epsilon = sqrtf(sum);
    __syncthreads();
    if(*epsilon < tol) break;//verificação da convergência
    __syncthreads();
    if(i==0)
        its_atual[cont] = its[cont];
    __syncthreads();
    //z = inv(matriz_diag(A))*r
    diagInv_sparse_mult(kernel_lista, gpu_r, gpu_z, i, cont);
    //beta1 = r(k+1) *z(k+1)
    Vet_Vet_Mul(gpu_r, gpu_z, cont, &gpu_aux1);
//VECTOR x VECTOR MULTIPLICATION
    __syncthreads();
    // beta = beta1 / beta2
    if(i==0)
        beta = gpu_aux1 / gpu_aux2;
    __syncthreads();
    // s = r + beta * s
    //Vet_Vet_Add2(gpu_r, gpu_aux_s, i, cont, gpu_s);
//VECTOR ADD
    Vet_Vet(gpu_z, gpu_p, beta, i, cont, gpu_p, 0);
//VECTOR ADD
    __syncthreads();
}
}
}
#endif          /* _KERNEL_JACOBI CG_GPU_H */

```


APÊNDICE B ALGORITMO PARA ARMAZENAMENTO DA MATRIZ ESPARSA – PTHREADS

A seguir apresenta-se a estrutura lógica para o armazenamento utilizando a técnica *row-indexed sparse storage mode* (PRESS et al., 1992) de uma matriz quadrada esparsa. Ade dimensões $N \times N$:

- 1) Preparam-se dois vetores: o primeiro sa , para armazenar os valores dos elementos da matriz, e o segundo, ija , para armazenar os índices de direcionamento sobre o vetor sa .
- 2) As primeiras N posições do vetor sa são ocupadas pelos valores dos elementos da diagonal da matriz A , em ordem. Os valores dos elementos da diagonal devem ser armazenados mesmo que sejam iguais a zero.
- 3) As primeiras N posições do vetor ija são ocupadas pelos índices de armazenamento em sa do primeiro valor diferente de zero fora da diagonal, para cada linha da matriz A . Se não houver valor diferente de zero fora da diagonal em uma linha da matriz A , ija deve assumir o valor do índice do último elemento armazenado em sa acrescido de uma unidade.
- 4) A primeira posição de ija deve armazenar o valor $N + 2$.
- 5) A posição $N + 1$ do vetor ija deve armazenar o índice de sa correspondente ao valor do último elemento diferente de zero da última linha da matriz A . O valor de $ija[N+1]$ pode ser lido para determinar o número de elementos diferentes de zero da matriz A ou o número de elementos nos vetores ija e sa . A posição $N + 1$ de sa não tem utilidade, podendo assumir qualquer valor.
- 6) As posições acima de $N + 1$ em sa devem armazenar os valores diferentes de zero dos elementos da matriz A fora da diagonal, ordenados pelas linhas e, em cada linha, ordenados pelas colunas.
- 7) As posições acima de $N + 1$ em ija devem armazenar as posições das colunas correspondentes aos elementos armazenados em sa .

Segue abaixo um exemplo de armazenamento de uma matriz esparsa encontrado no livro *Numerical Recipes in C* (PRESS et al., 1992):

$$\begin{bmatrix} 3 & 0 & 1 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 \\ 0 & 7 & 5 & 9 & 0 \\ 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 6 & 5 \end{bmatrix} \quad (\text{B.1})$$

Na técnica *row-indexed sparse storage mode* a matriz (B.1) é representada por dois vetores unidimensionais de tamanho 11 como abaixo:

Tabela B.1 – Armazenamento *row-indexed sparse storage mode*

Índice k	1	2	3	4	5	6	7	8	9	10	11
<i>ija[k]</i>	7	8	8	10	11	12	3	2	4	5	4
<i>sa[k]</i>	3	4	5	0	5	X	1	7	9	2	6

A seguir temos o código utilizando a linguagem C de programação para o armazenamento de uma matriz quadrada A de dimensões $N \times N$, representada por $a[1..n][1..n]$, empregando-se dois vetores *sa* e *ija* de dimensões $ija[ija[1]-1]-1$ (como explicado anteriormente).

Algoritmo 7: *Row-indexed sparse storage mode*

```
void sprsin(float **a, int n, float thresh, unsigned long nmax, float sa[],
           unsigned long ija[])
{
    int i, j;
    unsigned long k;
    for (j=1; j<=n; j++) sa[j]=a[j][j];
    ija[1]=n+2;
    k=n+1;
    for (i=1; i<=n; i++) { Loop over rows.
        for (j=1; j<=n; j++) { Loop over columns.
            if (fabs(a[i][j]) >= thresh && i != j) {
                if (++k > nmax) nerror("sprsin: nmax too small");
                sa[k]=a[i][j];
                ija[k]=j;
            }
        }
        ija[i+1]=k+1;
    }
}
```

APÊNDICE C MÉTODO DOS GRADIENTES BICONJUGADOS PRECONDICIONADOS

O método dos gradientes conjugados é um método poderoso para solucionar sistemas de equações do tipo

$$Ax = b. \quad (\text{C.1})$$

Seja A uma matriz simétrica e positiva definida e considere uma forma quadrática auxiliar

$$f(x) = \frac{1}{2} x^T A x - b^T x + c \quad (\text{C.2})$$

que transforma vetores em números reais. Como a forma é quadrática, há apenas um vetor que minimiza a função (C.2) e é o ponto crítico, solução de $\nabla f(x) = 0$, e neste caso corresponde a

$$\nabla f(x) = \frac{1}{2} (A^T + A)x - b = Ax - b, \quad (\text{C.3})$$

assim se o ponto mínimo for encontrado, ele será solução do sistema linear (C.1). A minimização é realizada gerando-se uma sucessão de direções p_k (direções de descida) que minimize x_k .

Considerando métodos iterativos de otimização do tipo

$$x_{k+1} = x_k + \alpha_k p_k, \quad (\text{C.4})$$

de forma que haja uma descida, ou seja, $f(x_{k+1}) < f(x_k)$. A cada iteração p_k e x_k são encontrados de maneira que x_{k+1} seja o minimizador de (C.2). Após N iterações espera-se que seja encontrado todo o espaço vetorial solução de (C.1).

Vamos agora generalizar estes procedimentos para encontrar a solução de sistemas lineares, mas não necessariamente positivos definidos ou simétricos. Uma importante ferramenta é o método dos Gradientes Biconjugados. Esse método, em geral, não tem uma conexão simples com a função de minimização. Ele constrói uma sequência de quatro vetores,

$r_k, \tilde{r}_k, p_k$ e $\tilde{p}_k, k=1, 2, 3, \dots$, assumindo que r_1 e $\tilde{r}_1$ como sendo vetores iniciais e definindo que $p_1 = r_1$ e $\tilde{p}_1 = \tilde{r}_1$. Então se obtém a seguinte recursividade:

$$\begin{aligned}\alpha_k &= \frac{\tilde{r}_k \cdot r_k}{\tilde{p}_k \cdot A \cdot p_k} \\ r_{k+1} &= r_k - \alpha_k A \cdot p_k \\ \tilde{r}_{k+1} &= \tilde{r}_k - \alpha_k A^T \cdot \tilde{p}_k \\ \beta_k &= \frac{\tilde{r}_{k+1} \cdot r_{k+1}}{\tilde{r}_k \cdot r_k} \\ p_{k+1} &= r_{k+1} + \beta_k p_k \\ \tilde{p}_{k+1} &= \tilde{r}_{k+1} + \beta_k \tilde{p}_k\end{aligned}\tag{C.5}$$

Esta sequência de vetores satisfaz a condição de biortogonalidade

$$\tilde{r}_i \cdot r_j = r_i \cdot \tilde{r}_j = 0, j < i\tag{C.6}$$

e a condição de dupla conjugação

$$\tilde{p}_i \cdot A \cdot p_j = p_i \cdot A^T \cdot \tilde{p}_j = 0, j < i.\tag{C.7}$$

Da mesma forma, a mútua ortogonalidade é satisfeita pela equação seguinte

$$\tilde{r}_i \cdot p_j = r_i \cdot \tilde{p}_j = 0, j < i\tag{C.8}$$

Considere agora o caso particular do método das direções conjugadas, em que elas são obtidas através do gradiente. No caso linear o gradiente é dado pelo resíduo

$$r_k = b - A \cdot x_k.\tag{C.9}$$

A equação (C.4) assegura que r_{k+1} proveniente da recursão é de fato o resíduo de $b - A \cdot x_{k+1}$ correspondente a x_{k+1} . Uma vez que $r_{m+1} = 0$, x_{m+1} é a solução da equação (C.1).

Até agora não dizemos nada sobre o grau de convergência do método. O método dos Gradientes Conjugados apresenta uma boa convergência se a matriz do problema é bem condicionada. Uma sugestão para acelerar a convergência do método dos Gradientes Biconjugados é utilizar pré-condicionadores na equação (C.1)

$$(\tilde{A}^{-1} \cdot A) \cdot x = \tilde{A}^{-1} \cdot b.\tag{C.10}$$

A idéia é resolver mais facilmente o sistema linear com o auxílio de alguma matriz pré-condicionadora $\tilde{A}$ que satisfaça a condição $\tilde{A}^{-1}A \approx 1$, ou seja, $\tilde{A}$ próxima de A . A intenção é gerar sistemas lineares auxiliares de fácil resolução que permita uma convergência acelerada do algoritmo (PRESS et al., 1992). Desta forma temos o método dos Gradientes Biconjugados Pré-Condicionado (PBCG).

A matriz $\tilde{A}$ deve ser definida de maneira que sua inversa seja previamente conhecida e pode, por exemplo, ser definida como sendo simplesmente a diagonal de A (interessante se esta for diagonal dominante). Entretanto, em alguns casos, $\tilde{A}$ e sua inversa podem ser definidas a partir de imagens obtidas da solução do problema tomográfico em tempos anteriores. Em outras palavras, em um esquema dinâmico em que o problema tomográfico é resolvido para produzir varias imagens por segundo, gerando assim uma visualização dinâmica, $\tilde{A}$ em um determinado instante pode ser definida como a matriz A do instante anterior.

Considere um conjunto de vetores adicionais z_k e $\tilde{z}_k$ definidos por

$$\tilde{A}.z_k = r_k \text{ e } \tilde{A}^T.\tilde{z}_k = \tilde{r}_k \quad (\text{C.11})$$

e modificamos as definições de α_k , β_k , p_k e $\tilde{p}_k$ na equação (C.5) da seguinte maneira:

$$\begin{aligned} \alpha_k &= \frac{\tilde{r}_k \cdot z_k}{\tilde{p}_k \cdot A.p_k} \\ \beta_k &= \frac{\tilde{r}_{k+1} \cdot z_{k+1}}{\tilde{r}_k \cdot z_k} \\ p_{k+1} &= z_{k+1} + \beta_k p_k \\ \tilde{p}_{k+1} &= \tilde{z}_{k+1} + \beta_k \tilde{p}_k \end{aligned} \quad (\text{C.12})$$

e finalmente tem-se o algoritmo do PBCG.

O pseudocódigo que se segue, foi adaptado do livro de Barret (BARRETT et al., 1994) que exemplifica a implementação do método do PBCG:

```

Calcule  $r_0 = b - A.x_0$  para algum  $x_0$  inicial.
Escolha  $\tilde{r}_0$  (por exemplo,  $\tilde{r}_0 = r_0$ ).
Para  $k = 1, 2, \dots$ 
  Resolva  $\tilde{A}.z_{k-1} = r_{k-1}$ 
  Resolva  $\tilde{A}^T.\tilde{z}_{k-1} = \tilde{r}_{k-1}$ 
       $\rho_{k-1} = z_{k-1}^T.\tilde{r}_{k-1}$ 
  se  $\rho_{k-1} = 0$ , o método falha
  se  $i = 1$ 
       $\rho_k = z_{k-1}$ 
       $\tilde{\rho}_k = \tilde{z}_{k-1}$ 
  senão
       $\beta_{k-1} = \frac{\rho_{k-1}}{\rho_{k-2}}$ 
       $p_k = z_{k-1} + \beta_{k-1}p_{k-1}$ 
       $\tilde{p}_k = \tilde{z}_{k-1} + \beta_{k-1}\tilde{p}_{k-1}$ 
  fim do se
   $q_k = A.p_k$ 
   $\tilde{q}_k = A^T.\tilde{p}_k$ 
   $\alpha_k = \frac{\rho_{k-1}}{\tilde{p}_k^T.\rho_k}$ 
   $x_k = \tilde{x}_{k-1} + \alpha_k p_k$ 
   $r_k = r_{k-1} - \alpha_k q_k$ 
   $\tilde{r}_k = \tilde{r}_{k-1} - \alpha_k \tilde{q}_k$ 
  Verifique a convergência; continue se necessário
fim

```

Figura C.1 – Pseudocódigo do método PBCG.

APÊNDICE D TABELA COM OS TEMPOS E OS DESVIOS PADRÕES DOS CASOS ANÁLISADOS EM GPU

Tabela D.1 – Tempos de processamento e desvio padrão dos códigos serial e paralelo em GPU.

		Tempo (s)	Desvio Padrão (s)
CPU 1 SERIAL	Caso 1.1	1428.8358	7.204392392
	Caso 1.2	1510.7934	9.490220958
	Caso 1.3	1435.3799	10.17700638
	Caso 2.1	1519.0324	7.069115774
	Caso 2.2	2313.2991	5.746692197
	Caso 2.3	1539.84	7.424080774
	Caso 3.1	1632.861	7.431076982
	Caso 3.2	1751.4873	10.92043715
	Caso 3.3	1545.3847	7.168286081
GPU 1	Caso 1.1	855.9842	9.106918015
	Caso 1.2	878.5648	2.078391515
	Caso 1.3	894.9893	5.528468666
	Caso 2.1	897.3611	3.743629743
	Caso 2.2	926.1009	1.885321632
	Caso 2.3	901.3482	8.846952568
	Caso 3.1	937.8928	7.78368864
	Caso 3.2	939.0509	3.030204777
	Caso 3.3	902.2433	16.21218723
GPU 2	Caso 1.1	552.2151	0.089991913
	Caso 1.2	562.2502	0.142469334
	Caso 1.3	579.3775	0.132982246
	Caso 2.1	599.2461	0.159468875
	Caso 2.2	581.4033	0.342741253
	Caso 2.3	579.5069	0.119119408
	Caso 3.1	596.752	0.090584522
	Caso 3.2	596.954	0.030643107
	Caso 3.3	577.04675	0.133247577

APÊNDICE E TABELA COM OS TEMPOS E OS DESVIOS PADRÕES DOS CASOS ANÁLISADOS EM PTHREADS

Tabela E.1 – Tempos de processamento e desvio padrão dos códigos serial e paralelo em CPU.

		Tempo (s)	Desvio Padrão (s)		Tempo (s)	Desvio Padrão (s)		Tempo (s)	Desvio Padrão (s)
CPU 1 SERIAL	Caso 1.1	1428,8358	7,204392392	CPU 1 PTHREADS	835,6915	0,28354982	CPU 1 PTHREADS + flag	381,3395	1,12422183
	Caso 1.2	1510,7934	9,490220958		754,287	1,24450793		388,9654	0,50186324
	Caso 1.3	1435,3799	10,17700638		917,0275	0,04737615		421,7505	5,80873875
	Caso 2.1	1519,0324	7,069115774		896,4715	0,01202082		464,372	1,43421802
	Caso 2.2	2313,2991	5,746692197		885,9195	0,44194174		458,3324	1,00272376
	Caso 2.3	1539,84	7,424080774		802,3055	3,04692312		405,8687	3,02304163
	Caso 3.1	1632,861	7,431076982		895,075	0,26162951		439,7023	0,64817831
	Caso 3.2	1751,4873	10,92043715		929,8895	1,22117341		441,4381	0,92119577
	Caso 3.3	1545,3847	7,168286081		794,191	0,81600123		405,142	2,43705852
CPU 2 SERIAL	Caso 1.1	1153,3101	0,339055043	CPU 2 PTHREADS	304,3395	0,2323625	CPU 2 PTHREADS + flag	97,6594	0,42535611
	Caso 1.2	1145,8132	0,312190166		301,85	0,39084652		97,3559	0,29102801
	Caso 1.3	1183,2594	0,235459456		332,420333	0,27769828		99,4775	0,19890548
	Caso 2.1	1426,6424	0,415320813		371,784	0,47471149		119,8854	0,23539197
	Caso 2.2	1381,865	0,365388682		360,348667	0,37943291		115,6551	0,31582395
	Caso 2.3	1230,0053	0,424534268		323,772667	0,36002268		104,2971	0,32935575
	Caso 3.1	1194,4289	0,302313469		337,107667	0,2008092		111,5378	0,21929422
	Caso 3.2	1265,6262	0,474303185		331,779667	0,37575834		106,6187	0,244619
	Caso 3.3	1219,3502	0,498309096		323,499333	3,08016515		105,5791	0,31418482
CPU 3 SERIAL	Caso 1.1	744,4229	0,130475583	CPU 3 PTHREADS	197,955333	1,24757458	CPU 3 PTHREADS + flag	71,9849	0,78626493
	Caso 1.2	797,7715	0,03662801		210,828	0,33656649		72,7509	1,26471266
	Caso 1.3	768,8111	0,060393984		203,054333	2,99846833		73,5604	0,8339005
	Caso 2.1	928,37	0,068572751		245,658667	0,50452882		85,1355	0,82823524
	Caso 2.2	898,9733	0,313317748		237,286667	1,10017377		81,9642	0,85258128
	Caso 2.3	799,0062	0,090247561		211,511	2,61973262		75,2755	1,297277
	Caso 3.1	863,2018	0,14421034		225,530667	1,4155297		79,3854	0,96787731
	Caso 3.2	822,6887	0,080822233		219,467	2,70939606		77,9392	0,60293206
	Caso 3.3	792,3851	0,334361232		210,751667	0,49721256		75,385	0,94630204