DESENVOLVIMENTO DE ALGORITMOS PARALELOS BASEADOS EM GPU PARA SOLUÇÃO DE PROBLEMAS NA ÁREA NUCLEAR

Adino Américo Heimlich Almeida

DISSERTAÇÃO SUBMETIDA AO PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA E TECNOLOGIA NUCLEARES DO INSTITUTO DE ENGENHARIA NUCLEAR DA COMISSÃO NACIONAL DE ENERGIA NUCLEAR COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA NUCLEAR PROFISSIONAL EM ENGENHARIA DE REATORES. Aprovada por:

Prof. Claudio Marcio do Nascimento Abreu Pereira, D.Sc.

Prof. Antonio Carlos de Abreu Mol, D.Sc.

Prof. Celso Marcelo Franklin Lapa, D.Sc.

Prof. Paulo Augusto Berquo de Sampaio, D.Sc.

Prof. Roberto Schirru, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
AGOSTO DE 2009

Almeida, Adino Américo Heimlich

Desenvolvimento de Algoritmos Paralelos Baseados em GPU para Solução de Problemas na Área Nuclear/Adino Américo Heimlich Almeida. – Rio de Janeiro: PPGIEN/CNEN, 2009.

XII, 80 p.: il.; 29, 7cm.

Orientadores: Claudio Marcio do Nascimento Abreu Pereira

Antonio Carlos de Abreu Mol

Dissertação (mestrado em engenharia de reatores) – PPGIEN/CNEN/Programa de Engenharia de Reatores, 2009.

Bibliografia: p. 72 – 80.

1. Métodos Computacionais . 2. Matemática Computacional. 3. GPU. 4. Computação Paralela. I. Abreu Pereira, Claudio Marcio do Nascimento *et al.*. II. Instituto de Engenharia Nuclear do Rio de Janeiro, PPGIEN/CNEN, Programa de Engenharia de Reatores. III. Título.

Aos meus amores Luciana,

Gabriel e Guilherme

Agradecimentos

Ao Prof. Dr. Claudio Márcio do Nascimento Abreu Pereira, orientador e amigo, pelo apoio durante a realização deste trabalho.

Ao Prof. Dr. Antonio Carlos de Abreu Mol, orientador e amigo, pelo incentivo e confiança durante a realização deste trabalho.

Ao grande mestre algebrista e amigo de todas as horas Prof. Dr. Adilson Gonçalves.

À Prof. Astréa Barreto pela grande amizade e percepção ímpar.

Ao grande mestre e amigo Prof. Luis Osório de Brito Aghina pela justa sabedoria e conhecimento.

Ao Dr. Marcel Waintraub pela amizade, pelo incentivo inabalável, durante a realização deste trabalho.

Ao Dr. Edson de Oliveira Martins Filho, pelo incentivo e amizade, durante a realização deste trabalho.

Agradeço ao apoio do Instituto de Engenharia Nuclear, na figura do Dr. Julio Cezar Suita, diretor, pelo apoio e disponibilização da infraestrutura necessária para a realização deste trabalho.

Resumo da Dissertação apresentada à PPGIEN/CNEN como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

DESENVOLVIMENTO DE ALGORITMOS PARALELOS BASEADOS EM GPU PARA SOLUÇÃO DE PROBLEMAS NA ÁREA NUCLEAR

Adino Américo Heimlich Almeida

Agosto/2009

Orientadores: Claudio Marcio do Nascimento Abreu Pereira

Antonio Carlos de Abreu Mol

Programa: Engenharia de Reatores

Unidades de processamento gráfico ou GPUs, são co-processadores de alto desempenho destinados inicialmente à melhorar ou prover de capacidade gráfica um computador. Desde que pesquisadores e profissionais perceberam o potencial da utilização de GPU para fins gerais, a sua aplicação tem sido expandida à outras áreas fora do âmbito da computação gráfica. O principal objetivo deste trabalho é avaliar o impacto da utilização de GPU em dois problemas típicos da área nuclear. O transporte de nêutrons utilizando simulação Monte Carlo e a resolução da equação do calor em um domínio bi-dimensional pelo método de diferenças finitas foram os problemas escolhidos. Para conseguir isso, desenvolvemos algorítimos paralelos para GPU e CPU nos dois problemas descritos anteriormente. A comparação demonstrou que a abordagem baseada em GPU é mais rápida do que a CPU em um computador com dois processadores *quad core*, sem perda de prcisão nos resultados encontrados.

Abstract of Dissertation presented to PPGIEN/CNEN as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

DEVELOPMENT OF PARALLEL GPU BASED ALGORITHMS FOR PROBLEMS IN NUCLEAR AREA

Adino Américo Heimlich Almeida

August/2009

Advisors: Claudio Marcio do Nascimento Abreu Pereira

Antonio Carlos de Abreu Mol

Department: Reactor Engineering

Graphics Processing Units (GPU) are high performance co-processors intended, originally, to improve the use and quality of computer graphics applications. Since researchers and practitioners realized the potential of using GPU for general purpose, their application has been extended to other fields out of computer graphics scope. The main objective of this work is to evaluate the impact of using GPU in two tipical problems of Nuclear area. The neutron transport simulation using Monte Carlo method and solve heat equation in a bi-dimensional domain by finite differences method. To achieve this, we develop parallel algorithms for GPU and CPU in the two problems described above. The comparison showed that the GPU-based approach is faster than the CPU in a computer with two quad core processors, without precision loss.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Este trabalho teve início em virtude da necessidade premente da área nuclear em encontrar novas soluções em processamento científico, que devem fazem jus aos novos problemas que enfrentaremos nos anos por vir. Problemas e soluções que devem e podem ser melhor explorados por uma estrutura computacional mais robusta e potente. Nesse trabalho vamos explorar uma nova tecnologia de processamento escalável, baseada em unidades de processamento gráfico (GPUs), que segundo os fabricantes de processadores INTEL [1], AMD [2], IBM [3], NVIDIA [4] impõem ganhos substanciais de velocidade em diversas aplicações em computação científica.

Isto se traduz em novas propostas de paralelismo computacional baseadas em GPU, que podem fornecer à área nuclear o poder computacional necessário à fazer frente à antigos problemas, tais como : recarga de reatores nucleares, projetos neutrônicos e termo hidráulicos, simulações probabilísticas por Monte Carlo, etc sob uma abordagem mais ampla; e ainda os novos, e custosos computacionalmente, problemas apresentados pelos novos reatores de quarta geração, as plantas de produção de hidrogênio e tantos outros que dependem de potência computacional. É claro que além disso, tal implementação deve ter custo de implementação e esforço de aprendizado em um nível aceitável.

Nossa avaliação se conduz através da simulação de dois problemas característicos da engenharia de reatores, contudo, faz-se necessária uma descrição da cronologia do desenvolvimento e da evolução dos sistemas computacionais desde o seu surgimento em meados da década de 1940 até o momento. Em torno do desenvolvimento dos artefatos nucleares que destruíram Hiroshima e Nagasaki, o projeto Manhatan

aglutinou as mentes mais brilhantes de seu tempo, e entre estas talvez John Von Neumman possa ser considerado o *pai* da computação científica ao criar o modelo básico de arquitetura do computador que conhecemos hoje; e este modelo possuía 3 características básicas conforme podemos observar na figura 1.1.

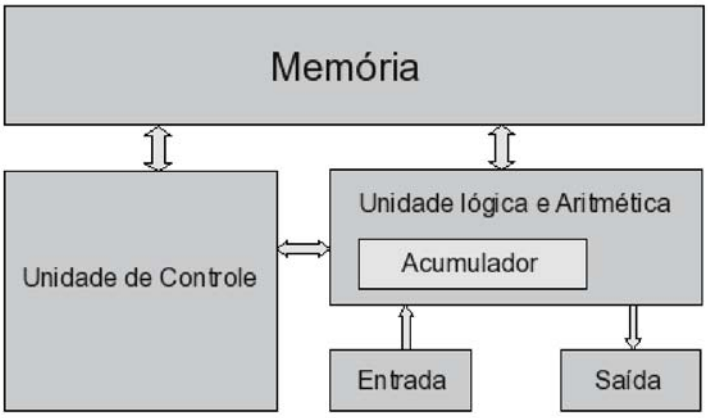


Figura 1.1: Arquitetura de Von Neumman

- As instruções são codificadas de uma forma possível de ser armazenada em base binária.
- Tais instruções e outras informações relevantes são armazenadas na memória.
- Quando processar o programa, buscar as instruções diretamente na memória.

Nasce assim, o computador programável que conhecemos hoje, onde o programa e os dados estão armazenados na memória e tal implementação ficou conhecida como *Arquitetura de von Neumann* [5]. Pois bem, a menos de uma geração, a utilização de sistemas computacionais dos mais variados tipos tem impulsionado áreas tão aparentemente dispares quanto física de partículas e ecologia; e a princípio não podemos hoje excluir nenhuma área do conhecimento humano do progresso científico e tecnológico advindo da utilização de computadores.

A evolução de tais sistemas viabiliza o aparecimento e desenvolvimento de novas áreas de pesquisa na matemática, física e engenharia, inviáveis de outra forma. Tomando como por exemplo, o cálculo da *queima* de um elemento combustível

no reator nuclear, e mesmo considerando muitas aproximações, não consideramos encontrar uma solução analítica em qualquer geometria.

Ao longo das duas ultimas décadas o desenvolvimento de tecnologias de paralelismo em memória distribuída e troca de informações entre computadores pessoais (PCs), MPI (*Message Passing Interface*) [6] e PVM (*Parallel Virtual Machine*)[7], possibilitou a organização destas máquinas em aglomerados de computadores de baixo custo [8] baseadas em sistemas operacionais abertos, tais como LINUX e FREEBSD [9]. Como exemplo desta arquitetura podemos citar o projeto da NASA denominado *Beowulf* [10].

Concomitantemente o *hardware* dos PCs evoluiu e incorporou características de sistemas maiores, que à época eram chamados de *mainframe*, vários processadores operando juntos em memória compartilhada ou SMP (*Symmetric Multiprocessing*), e como solução para manipular os processos e acessos à memória neste novo paradigma surgiram o OPENMP [11] e o *posix threads* [12].

Estas novas ferramentas computacionais ampliaram o interesse e a busca de novos algoritmos paralelos e novas maneiras de implementa-los nestas novas arquiteturas. Entre estas novas tecnologias podemos destacar o processamento em GPU(*Graphic Processor Unit*) [13], BSP (*Bulk synchronous parallel*) [14], PRAM (*Parallel Random Access Machine*) [15] e XMT (*Explicit Multi-Threading*) [16], que na verdade é a implementação em FPGA (*Field Programmable Gate Array*) do modelo idealizado na proposta da tecnologia PRAM.

Embora tecnologicamente diferentes todos estes modelos de processamento podem ser classificados pela taxionomia de Flynn [17]. Nosso objetivo neste trabalho é mostrar que a arquitetura computacional paralela das GPUs pode ser de grande valia aos pesquisadores na área nuclear. Vamos obter uma analise comparativa do desempenho computacional entre dois sistemas.

De um lado um computador com dois processadores XEON *quad core* executando o código sob um sistema operacional LINUX; e do outro, uma placa gráfica GTX-280 da NVIDIA. Uma visão aproximada do *hardware* empregado na GPU, que de maneira genérica faz parte de uma família de processadores denominada G200, pode ser visto na figura 1.2, possui 240 processadores denominados por *stream processor* e se conecta ao sistema pelo barramento PCI-e da placa mãe.

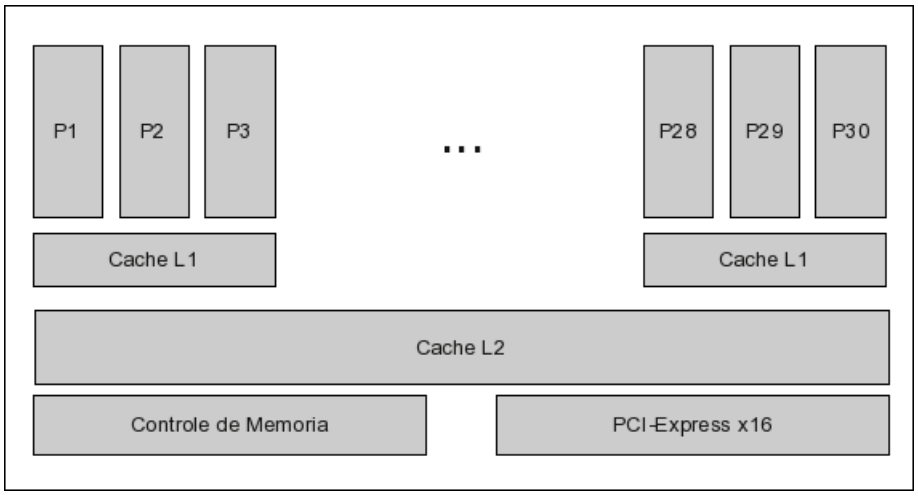


Figura 1.2: Arquitetura básica do chip G200

A partir desta orientação desenvolvemos nosso trabalho em função da busca de dois algoritmos representativos das necessidades da engenharia nuclear, e nossa escolha recaiu sobre o problema de transporte de nêutrons utilizando o método de Monte Carlo, empregado no cálculo de uma blindagem. E o outro a resolução do problema da transferencia de calor estacionário, com uma abordagem que resolve a equação de Poisson em um domínio retangular através do método das diferenças finitas.

Criamos os programas que resolvem as questões acima em linguagem C e os implementamos em *posix threads*, sobre um computador que possui dois processadores de quatro núcleos; além disso, criamos os programas correspondentes em CUDA, uma extensão da linguagem C/C++ que simplifica a programação genérica para GPUs da NVIDIA. Desde o seu lançamento diversos pesquisadores, cientistas e empresas tem apresentado novas soluções em áreas como : CFD (Dinâmica dos Fluidos Computacional) [18], resolução de integrais estocásticas [19], seqüenciamento de proteínas [20], tomografia [21] e dezenas de outras onde a velocidade de processamento é um item fundamental no processo de análise.

Os capítulos deste trabalho estão distribuídos da seguinte forma:

- No capítulo 2 analisaremos alguns aspectos da computação paralela de alto desempenho com referências à taxionomia de Flynn [22], a lei de Amdahl e os problemas relacionados à comunicação entre processos concorrentes. Além

disso uma breve introdução à programação na biblioteca *Posix Threads* [23] e a linguagem CUDA [4].

- No capítulo 3 apresentaremos os dois problemas nucleares abordados : O transporte de nêutrons e a transferência de calor; divididos em duas seções. Serão abordados também os métodos empregados na construção dos programas para a CPU e a GPU.

  - Na seção 3.1 vamos tecer considerações em relação as características probabilísticas das seções de choque de absorção e espalhamento e sua influencia no livre caminho médio; e com muitas simplificações, elaboramos uma solução para o cálculo de blindagem utilizando o método de Monte Carlo.

  - Na seção 3.2 vamos abordar o problema de transporte de calor, mais especificamente a solução da equação de Poisson estacionária. E sua resolução através da implementação dos algoritmos, paralelo e seqüencial, de Gauss-Seidel para o *stencil* de diferenças finitas de cinco pontos. Deve-se mencionar o seguinte fato: a proposta original do trabalho previa apenas a implementação do algoritmo de Jacobi em OPENMP, e sua implementação apesar de simples, mostrou-se demasiado lenta em relação ao *POSIX Threads*, em virtude do qual o adotamos. Além disso como o método de Jacobi é intrinsecamente paralelizável não nos fornece a dificuldade adicional do desacoplamento dos *stencils* necessário ao paralelizar o método de Gauss-Seidel.

- No capítulo 4 serão apresentados os resultados obtidos nas simulações e a sua análise.

- No capítulo 5 concluímos o trabalho com uma análise de desempenho das duas tecnologias e uma apresentação dos futuros trabalhos.

Capítulo 2

Modelos de Computação Paralela

2.1 Computação de Alto Desempenho

A simulação computacional surgiu como conseqüência direta da construção dos primeiros computadores nas décadas de 40 e 50 do século passado; podemos citar o ENIAC (*Electronic Numerical Integrator And Computer*) e seu sucessor EDIVAC (*Electronic Discrete Variable Automatic Computer*) como os primeiros computadores dedicados ao cálculo científico, sendo que este último já apresentava uma arquitetura de Von Neumman.

Na computação de alto desempenho desejamos obter o máximo possível de velocidade em cálculos científicos e outras tarefas onde a performance é um fator crítico. A cada geração de processadores e computadores as limitações físicas dos materiais e das técnicas de construção tem limitado o poder de processamento individual destes sistemas, mas as necessidades da comunidade científica demandam o emprego de técnicas de paralelismo, sejam de dados ou instruções, de modo a fazer frente a tal desafio. Tomando como égide a máxima de dividir para conquistar, na computação paralela grande problemas geralmente podem ser divididos em problemas menores, que então são resolvidos de maneira concorrente.

Observando o contexto atual do desenvolvimento das placas gráficas de última geração, que utilizam basicamente tecnologia SIMD (*Single Instruction Multiple Data*) nosso objetivo é verificar se seu potencial computacional pode ser utilizado com relativa facilidade na área nuclear e em especial na área de engenharia de reatores.

Existem diferentes técnicas de paralelismo e uma delas em particular, a distribuição de pequenos problemas em memória distribuída em *clusters* de computadores [24], a comunidade científica já utiliza desde a década de 90. Seu custo é relativamente baixo em relação à sistemas mais complexos como NUMA (*Non-Uniform Memory Access*) [25] e mais recentemente INFINIBAND [26].

Outro fator a levar em consideração é o custo relacionado ao suprimento de energia dos *clusters* de alto desempenho; pois além do custo da tarifa ao alimentar os computadores devemos levar em conta a compra e manutenção de dispendiosos sistemas de refrigeração e *nobreak*. Atualmente computadores em que o processador apresenta múltiplos núcleos em um substrato são relativamente comuns e estações de trabalho com mais de um processador multi-núcleo estão disponíveis no mercado; de forma que a base necessária para a programação paralela está lançada, mesmo para o pequeno usuário doméstico.

O desenvolvimento dos novos processadores baseados em dezenas e até centenas de núcleos é uma realidade que nos chega mais perto a cada dia [27]. Ora, basta observar agora que possuímos as técnicas e o hardware necessário mas nos falta o primordial ... os algorítimos.

Nossos algoritmos se adaptam à nova realidade ?

Com o intuído de melhor descrever os problemas enfrentados por escolhermos o modelo de paralelismo baseado em GPU e *posixthreads* devemos nos referenciar à taxionomia de Flynn [17], e os problemas relativos à comunicação e *speedup* entre os múltiplos processos concorrentes.

2.1.1 A taxionomia de Flynn

São quatro as classificações de uma arquitetura de processamento computacional, definidas por Flynn [17], e que tem na concorrência de processos, no sentido de concomitância, ou no controle de fluxo de dados em memória seus indexadores e podemos observa-la na figura 2.1. São elas:

Single Instruction, Single Data stream **(SISD)**

É a clássica taxionomia seqüencial, que não explora o paralelismo, seja por instruções ou fluxo de dados; podemos compara-la a computadores de núcleo

simples que possuem paralelismo de tarefas apenas por fatiamento de tempo processador

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

Descreve um sistema computacional que, processa múltiplos fluxos de dados simultaneamente a cada ciclo de uma instrução e pode executar operações que sejam naturalmente paralelizáveis; atualmente GPUs, FPGAs se encontram nesta categoria, bem como os grandes processadores vetoriais.

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

Múltiplas instruções operam em um fluxo de dados único. É uma estrutura pouco comum, mas geralmente utilizada por sistemas tolerantes à falha, garantindo alta disponibilidade.

***Multiple Instruction, Multiple Data streams* (MIMD)**

Múltiplos processadores autônomos processam simultaneamente instruções diferentes, em espaços de memória diferentes. Podemos associa-los a tecnologias de processamento distribuído tais como o PVM e o MPI.

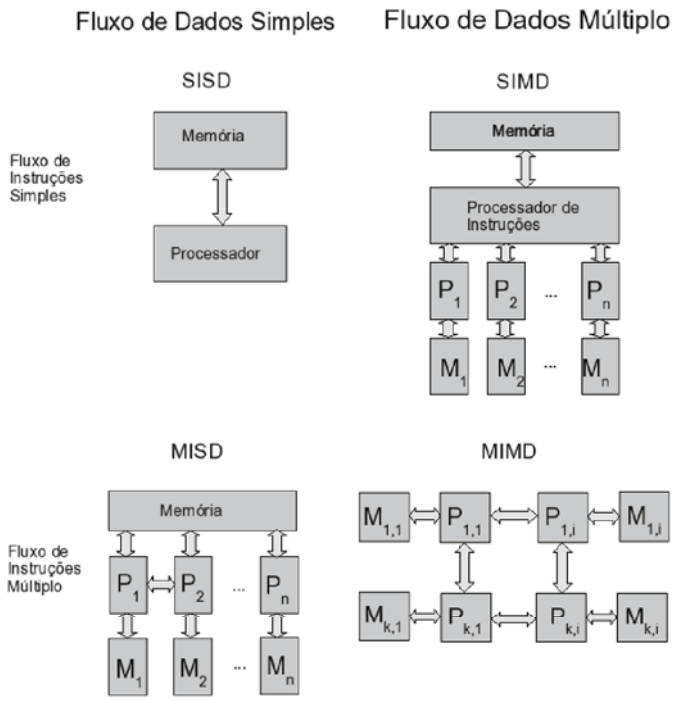


Figura 2.1: Taxionomia de Flynn

2.1.2 A Lei de Amdahl

A comunicação e a sincronização entre diferentes sub-tarefas é tipicamente uma das maiores barreiras para atingir grande desempenho em programas paralelos. O aumento da velocidade por resultado de paralelismo é dado pela lei de Amdhal [28].

Algoritmos paralelos em alguns casos são mais difíceis de programar que os seqüenciais pois a concorrência pela memória introduz diversos novos problemas, tal como a condição de corrida (*race conditions*); esta é uma falha produzida pelo fato de o resultado do processo ser inesperadamente dependente da seqüencia ou sincronia de outros eventos.

A lei de Amdahl é um modelo que relaciona o acréscimo de velocidade esperado em um algorítimo que possui parcelas seqüenciais e paralelas, ou seja, tomando um sistema computacional onde temos N processadores e se S é o tempo dispendido na parte serial do código e P é o tempo utilizado pelo código pra executar as partes paralelas, então a lei de Amdahl nos diz que:

$$Speedup = \frac{S + P}{S + \frac{P}{N}}. \tag{2.1}$$

2.1.3 A Influência da Comunicação

Uma das técnica de paralelismo mais difundidas entre programadores e cientistas é conhecida como *Divide and Conquer*, esta técnica é baseada em ramificações do problema onde cada ramo, ou seja o problema, seguinte é menor do o anterior de forma racional, no sentido de divisão por número inteiro. Esta técnica é sofisticada e de fato um dos principais desafios ao utiliza-la, reside em otimizar a divisão dos blocos de dados sob análise; por um lado existe a necessidade de comunicação entre os blocos e objetivo de manter a integridade da informação; e pelo outro lado acelerar a execução do código. Vamos ilustrar o problema com um exemplo.

Tomemos uma região quadrada e a dividamos em 16 blocos quadrados de igual tamanho, vamos presumir que os dados assumem posições pontuais no interior de cada bloco e por uma questão de simplicidade, cada bloco possui $n = \frac{N}{4}$ conjuntos de troca de informação em cada direção (norte, sul, leste e oeste) como podemos observar na figura 2.2.

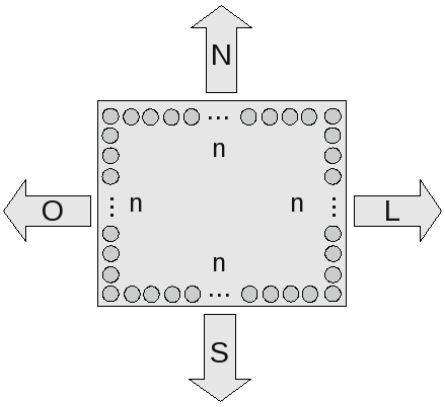


Figura 2.2: Detalhe da comunicação do bloco

Vamos também denotar por $M_{4\times4}$ o total de mensagens entre os blocos e por $D_{4\times4}$ o total de unidades de dados transferidos para cada bloco. Podemos observar esta partição na figura 2.3.

2	3	3	2
3	4	4	3
3	4	4	3
2	3	3	2

Figura 2.3: Comunicação Bidimensional em uma grade 4×4

Como podemos observar o número de regiões onde há troca de mensagens é de,

$$M_{4\times4} = 4 \times 2 + 3 \times 8 + 4 \times 4 = 48. \tag{2.2}$$

Ou seja, a equação 2.2 nos diz que uma região quadrada dividida em 16 blocos de igual tamanho possui um total de regiões onde deve ocorrer troca de informação

entre os blocos $M_{4\times4}$ composta por 4 blocos com 2 regiões de troca, 8 blocos com 3 regiões de troca e 4 blocos com 4 regiões de troca. Como cada bloco possui um conjunto de $\frac{N}{4}$ dados em cada direção, e a quantidade de informação que deve ser trocada neste modelo é proporcional à $M_{4\times4}$, temos que tomando $D_{4\times4}$ como este total,

$$D_{4\times4} = M_{4\times4} \times \frac{N}{4} = 12N. \tag{2.3}$$

Por outro lado podemos dividir a mesma região em 16 retângulos de mesmo tamanho em uma partição unidimensional onde cada bloco possui N dados em cada direção e vamos também denotar por $M_{1\times16}$ o total de mensagens entre os blocos e por $D_{1\times16}$ o total de unidades de dados transferidos para cada bloco. De acordo com a figura 2.4.

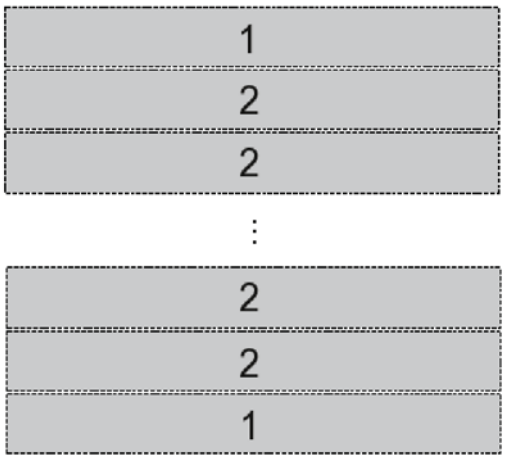


Figura 2.4: Comunicação Unidimensional em uma grade 1x16

E portanto podemos observar o número de regiões onde há troca de mensagens é de :

$$M_{1\times16} = 2 \times 1 + 14 \times 2 = 30. \tag{2.4}$$

e o número de unidades de dados que se comunicam é de:

$$D_{1\times 16} = M_{1x16} \times N = 30N. \tag{2.5}$$

De forma geral, podemos tomar um problema *grid* de tamanho $N_x \times N_y$ e particiona-lo em $I \times J$ sub-problemas que podem ser mapeados em $I \times J$ processadores. O número de mensagens, $M_{I\times J}$, e a quantidade de informação transferida, $D_{I\times J}$, em uma partição bidimensional de tamanho $I \times J$ é dada por,

$$M_{I\times J} = 4IJ - 2(P + Q) \tag{2.6}$$

$$D_{I\times J} = 2(N_y I + N_x J) - 2(N_x + N_y) \tag{2.7}$$

e em uma partição unidimensional $1 \times IJ$:

$$M_{1\times IJ} = 2(PQ - 1) \tag{2.8}$$

$$D_{1\times IJ} = 2N_x(PQ - 1) \tag{2.9}$$

O que implica que a partição bidimensional requer mais chamadas de comunicação porém requer uma menor largura de banda. Por outro lado na partição unidimensional o número de chamadas de comunicação é menor mas esta necessita de uma maior largura de banda, ou seja, apesar de um número menor de chamadas ser necessário, mais informação é transferida em cada comunicação.

Estes fatos tem conseqüências importantes na maneira como vamos estruturar nossos algoritmos, devido aos gargalos representados pela comunicação no desempenho geral do código; devemos lembrar que nem sempre é possível paralelizar a comunicação e pela lei da Amdahl o aumento do número de eventos seqüenciais tende a reduzir o paralelismo.

Neste trabalho vamos explorar vertentes de particionamento do problema por partição do domínio e através de algorítimos paralelos desenvolvidos em uma bibli-

oteca chamada *posix threads* e em CUDA.

2.2 *Posix Threads*

A biblioteca *Posix Threads* é um padrão para a programação de processos concorrentes [23] [29], que chamamos de *threads*, é baseada em uma interface de aplicação C/C++ e que é executável em sistemas operacionais LINUX ou SOLARIS. 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 *cores*.

O *Posix Threads* sobrecarrega menos a máquina do que outros métodos de concorrência, como *forking* e *spawning* [30], pois o sistema não inicia um novo espaço de memória virtual e ambiente para o processo; estes ganhos de desempenho podem ser mais expressivos em sistemas multi-processados. Por outro lado sistemas uni-processados também podem experimentar ganhos com a execução em *multi-threading* nos processos de entrada/saída e outros processos do sistema envolvendo altas latências.

As operações com *threads* incluem a criação, finalização, sincronização, agendamento, sinalização e interação com o processo *pai*. Um *thread* não pode manter uma lista de *threads*, todos os *threads* de um processo *pai* compartilham o mesmo espaço de endereçamento.

Os *threads* de um mesmo processo compartilham os descritores de arquivos abertos, os gerenciadores de sinalização, instruções do processo , dados e as variáveis de ambiente. Cada *thread* de um processo possui seu próprio conjuntos de registros e *stack pointer*, um *stack* para variáveis locais, ajuste de prioridade e retorna zero se não houver error na execução.

2.2.1 Sincronização

A biblioteca *Posix Threads* fornece 3 maneiras de efetuar a sincronização,

Mutexes

Mutual exclusion lock bloqueia o acesso à variáveis por outros *threads* im-

pondo acesso exclusivo à uma variável ou conjunto de variáveis. Mutexes são utilizados pra prevenir condições de corrida, que pode ocorrer quando dois ou mais *threads* necessitam operar na mesma região de memória. Mutexes só se aplicam aos *threads* do mesmo processo, tem a funcionalidade dos semáforos na sinalização entre os processos

Joins

Faz com que um *thread* espere até que todos os outros terminem.

Variáveis Condicionais

Uma variável condicional é uma variável do tipo

```
pthread_cond_t
```

2.2.2 Programando no Posix Threads

O exemplo a seguir cria dois processos concorrentes utilizando a biblioteca *POSIX THREADS*.

```
// O programa imprime duas strings de maneira concorrente utilizando
// threads

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

// declara a funcao que imprime a mensagem

void *imprime_mensagem( void *ptr );
main()
{
     pthread_t thread1, thread2;
     char *menssagem1 = "Ola Mundo 1";
     char *menssagem2 = "Ola Mundo 2";
     int  iret1, iret2;

     // Cria threads com apontadores independentes para a funcao
     // imprime_mensagem
     iret1 = pthread_create( &thread1, NULL, imprime_mensagem,
                             (void*) mensagem1);
     iret2 = pthread_create( &thread2, NULL, imprime_mensagem,
                             (void*) mensagem2);
```

```
     // espera ate que cada thread complete sua tarefa

     pthread_join( thread1, NULL);
     pthread_join( thread2, NULL);
     printf("Thread 1 retornou: %d\n",iret1);
     printf("Thread 2 retornou: %d\n",iret2);
     exit(0);
}
void *imprime_mensagem( void *ptr )
{
     char *mensagem;
     mensagem = (char *) ptr;
     printf("%s \n", mensagem);
}
```

2.3 Programação em CUDA

Com o aparecimento das bibliotecas de programação gráfica OPENGL e DIRECTX em meados da década de 90 os fabricante de GPUs tiveram a oportunidade de adicionar as suas capacidades *pixel shaders* programáveis, ou seja, estruturas de hardware onde cada pixel poderia ser processado em um pequeno programa e poderia incluir texturas como sua entrada. De maneira similar cada *geometric vertex* poderia ser processado por um pequeno programa antes de ser colocado no *buffer* de saída da GPU.

Ao longo do tempo outras linguagens de programação se tornaram disponíveis tais como **Sh** [31] e o **Brook** [32]. Estas novas linguagens aliadas a evolução gradual das GPUs tornaram a tarefa de programa-las mais acessível aos pesquisadores de diversas áreas do conhecimento. Além disso os desenvolvedores de algoritmos paralelos encontraram nas GPUs, que devido às suas estruturas de dezenas e até centenas de núcleos de processamento, máquinas nas quais podiam testar suas teorias.

O crescimento da comunidade de programação paralela baseada em GPUs desembocou em imensa produção científica [33] e o ganho em performance em alguns casos supera em mais de duas ordens de grandeza o desempenho de uma CPU [34].

O CUDA (*Compute Unified Device Architecture*) [4] é formado por um conjunto de extensões à linguagem C padrão, não constituindo um entrave à programação por qualquer programador em C com alguma experiência. O código de máquina gerado

por seu compilador, baseado no projeto OPEN64 [35], no qual a parte do código que executa na CPU é compatível com o *link* dos compiladores GNU e seu *debugger*.

A estrutura de desenvolvimento do CUDA distribuída pela NVIDIA consiste basicamente de um *driver* em uma camada de baixo nível para acesso aos dispositivos de *hardware*, uma biblioteca de *runtime* e duas outras bibliotecas de alto nível que fornecem funções em álgebra linear e FFT (*Fast Fourier Transform*).

A programação de uma GPU utilizando o CUDA é orientada à *threads*, ou linhas de execução, que é uma forma de um processo dividir a si mesmo em duas ou até mesmo centenas de tarefas, que podem ser executadas simultaneamente. Neste modelo, quando um *grid* com blocos de *threads* é criado, este consiste em centenas de processos unitários que podem ser identificados de maneira única pelo hardware, dependendo apenas de sua posição no bloco, como vemos na figura 2.5.

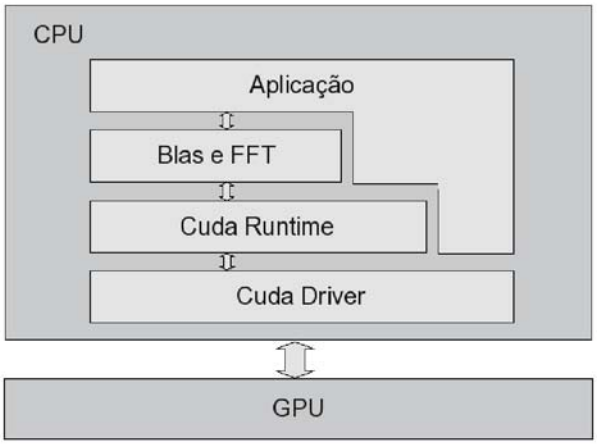


Figura 2.5: Estrutura CUDA

Atualmente cada *thread* pode executar instruções em precisão inteira e ponto flutuante de precisão simples e dupla, sobre funções que chamamos de textitkernels, onde cada uma destas funções é executada em cada bloco do *grid* de maneira independente. O *hardware* que utilizamos em nossos experimentos, a placa GTX-280 da NVIDIA, tem como base 240 *streaming processors*(SPs) agregados em grupos de processadores de textura (TPCs). Na figura 2.6 vemos em detalhe a SM.

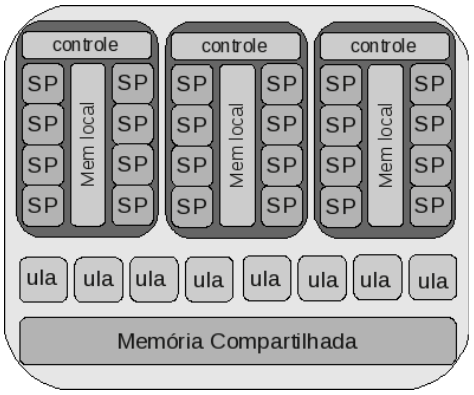


Figura 2.6: Estrutura CUDA

Temos ao todo dez TPCs organizados em grupos de três *streaming multiprocessors* (SMs). Cada SM possui oito SPs com 16Kb de memória compartilhada e dois *caches*, um para instruções e um para constantes, uma unidade para instruções *multi-thread*, duas unidades com funções especiais e uma unidade de ponto flutuante de 64 *bits*, conforme publicado por Lindholm et al [36]. Podemos observar a estrutura geral na figura 2.7

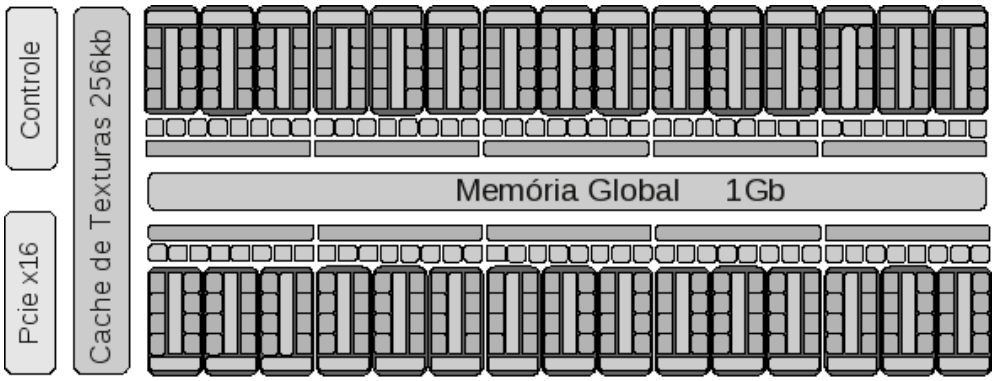


Figura 2.7: Estrutura CUDA

Cada SM pode manipular até 1024 *threads* e está estruturada em uma forma à reduzir o processo de decisão de quantos recursos estão disponíveis, de forma a diminuir a sobrecarga no acesso à memória e possibilitar o sincronismo entre os *threads* em poucos *clocks* de máquina. Esta é uma estrutura de blocos de *threads* denominada *warps* e em cada *warp* temos 32 *threads*. Estes *warps* são então executados em um manipulador denominado *Single Intruction Multiple Thread* e cada *thread* no *warp* possui seus próprios registros mas todos estão executando a mesma

instrução. Como vimos na taxionomia de Flynn, esta classifica esta topologia como uma máquina SIMD. Podemos observar um detalhe da SM em relação à estrutura geral na figura 2.8.

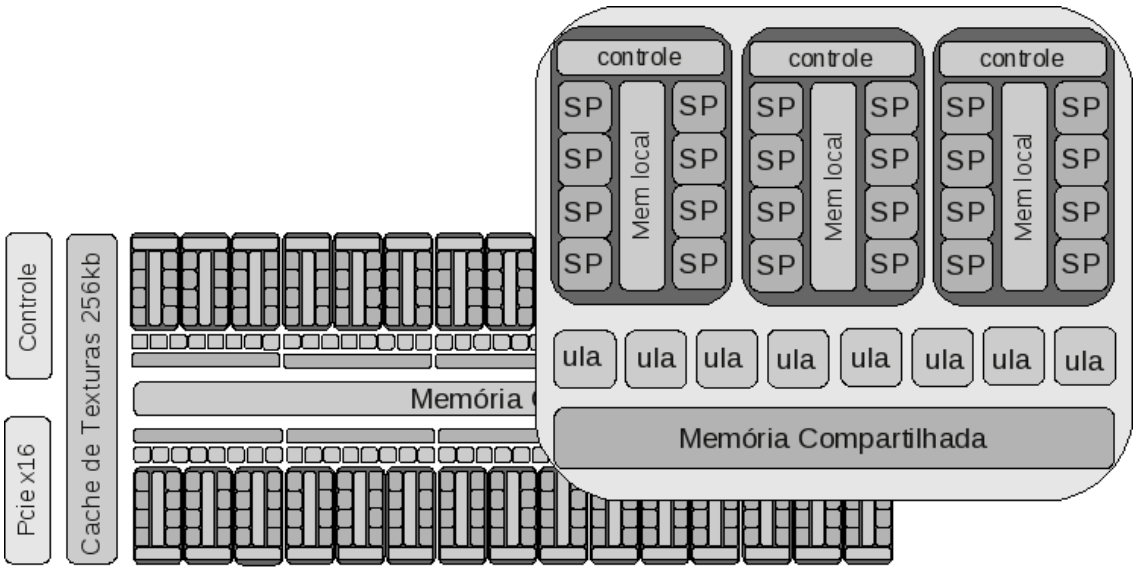


Figura 2.8: Estrutura CUDA

De forma à suportar uma possível divergência em cada *thread*, provocado por situações condicionais do tipo IF(), WHILE() ... , que podem ocorrer dentro do *warp* na qual alguns *threads* podem ficar inativos durante a execução das instruções. Neste caso diferentes ramificações do programa são serializadas em relação aos seus respectivos *threads*. De forma que o ideal é que os *threads* no mesmo *warp* estejam em sincronia. A implicação fundamental é que para obtermos uma performance ótima, todos os *threads* do *warp* devem seguir os mesmos passos; além disso este modelo baseado em *threads* de *kernels* só se justifica se utilizarmos em nossos programas uma enorme quantidades de *threads*.

Na solução proposta particionarmos grandes problemas em milhares de pequenos, que executam o mesmo código e transacionam entre si através da memória, mas em pequenos blocos. Isto não é evidentemente possível em qualquer problema, mas desenvolver algoritmos paralelos já é um ramo fascinante da ciência.

2.3.1 *Warps*

Um *warp* é formado por 32 *threads* e um *half-warp* por 16. Ao executar uma chamada de função na GPU os *threads* individuais de um *warp* são iniciados juntos no mesmo endereço de programa. Cada *warp* executa uma instrução comum à todos os seus *threads*, no caso do *thread* divergir em ramos por dependência de dados criados por situações condicionais, o *warp* executará em série o novo ramo, desabilitando o *thread* que não está alinhado e quando todo o ramo está completo os *threads* convergem de volta ao caminho anterior à ramificação

A divergência em ramos ocorre apenas dentro de um *warp*, *warps* diferentes podem executar independentemente de se tratar de execução comum ou disjunta de caminhos de código. A GPU pode ir buscar dados com 64 bytes ou 128 bytes em uma única transação. Se a memória não pode ser acessada de forma coalescente com o *half-warp*, uma transação em separado será emitida em seguida, o que não é desejável.

2.3.2 O Modelo de Programação

De forma à simplificar a manipulação de um grande números de *threads* o CUDA nos oferece o conceito de *grids* e de blocos de *threads* no qual nosso domínio computacional, os milhares de *threads*, são divididos em conjuntos que chamamos de *grids* de blocos, de maneira unidimensional ou bidimensional. Cada um destes blocos pode conter até 512 *threads* organizados em um *grid* tridimensional como mostrado na figura 2.9.

O blocos são mapeados nas SMs e o *driver* do CUDA oferece maneira de identificar tanto a posição do bloco no *grid* quanto a posição do *thread* no bloco, isto é feito através das variáveis de sistema *blockIdx* e *threadIdx* , as quais são vetores tri-dimensionais que apontam para o índice de seu *thread*; através destas variáveis é possível identificar e manipular os *threads* individualmente e isto é muito útil ao lidar com condições de contorno de um problema.

A sincronização entre os *threads* de um bloco é realizada através da chamada da primitiva *syncthreads()*, mas a sincronização entre *threads* de blocos diferentes não é

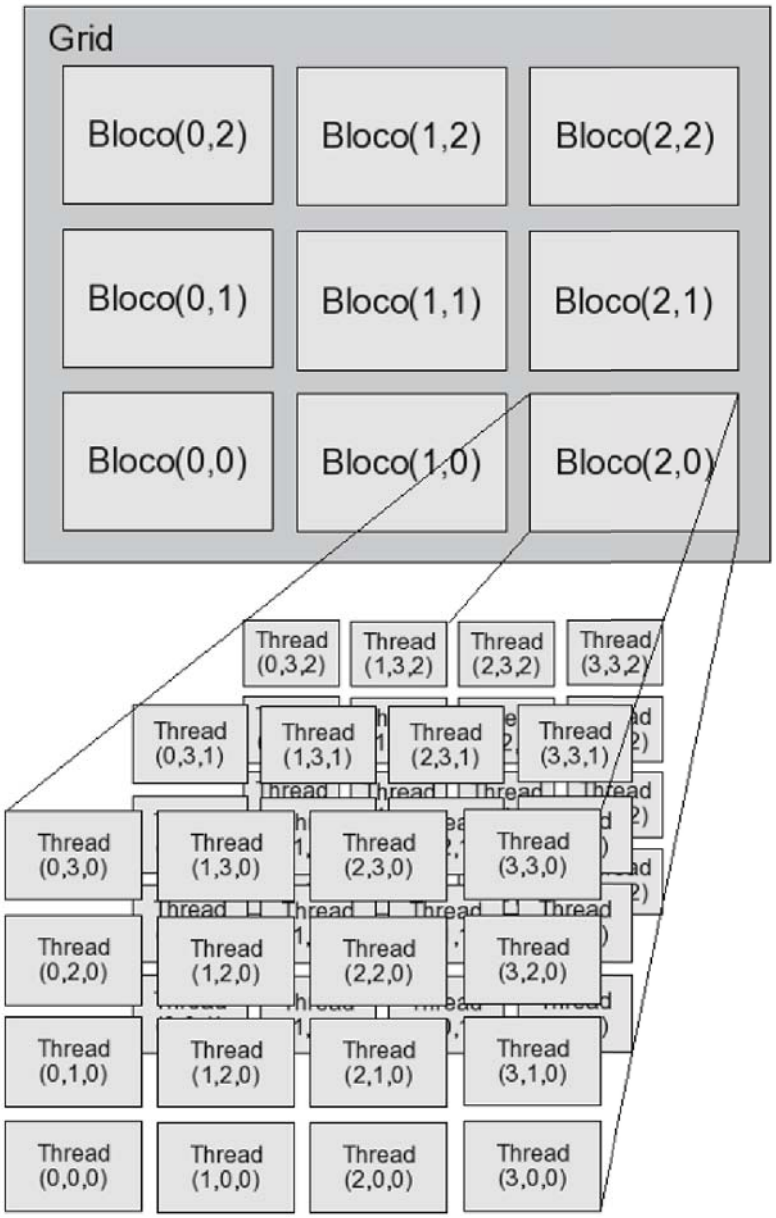


Figura 2.9: Estrutura de Grids e Blocos de Threads no CUDA

possível. É importante ressaltar que o programador não tem influência na seqüencia sob a qual um *thread* individual ou um bloco de *threads* é processado, o *hardware* é responsável por isso através de gerenciadores internos de fila.

2.3.3 O Modelo de Memória

Em computação, o modelo de memória descreve como os *threads* de processamento interagem com a própria; nas CPUs atuais o modelo de memória pode ser

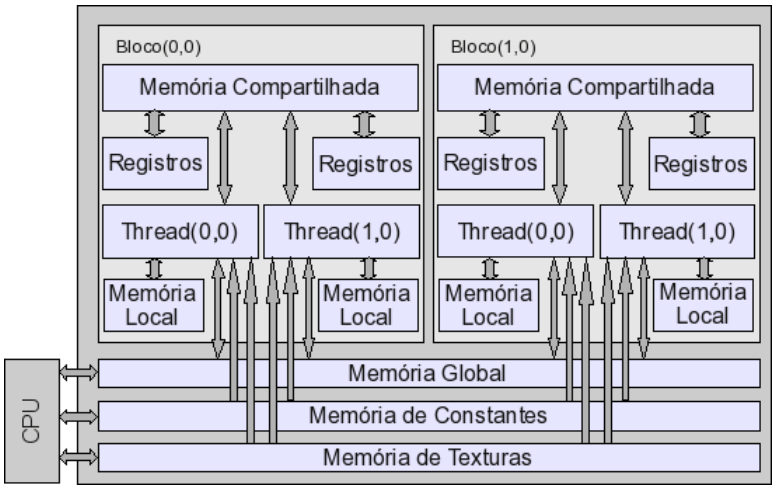


Figura 2.10: Modelo de Memória do CUDA

plano, ou *flat*; ou ainda segmentado, no caso de plataformas que utilizam memória paginada [37]. O modelo hierárquico de memória dos processadores gráficos utilizados pelo CUDA é um pouco diferente do modelo de memória de uma CPU comum, mas a diferença mais importante é a ausência completa de qualquer modelo de pilha para os *threads* de *hardware*, e de fato apenas as memórias de constantes e de texturas são *cached*. Podemos ver na figura 2.10 uma representação deste modelo de memória. Uma descrição detalhada da programação, acesso à memória e exemplos de utilização, pode ser encontrada nos manuais de programação da NVIDIA [38] e em seu SDK.

A memória da GPU é distribuída entre cinco tipos de acesso, cada um com características próprias:

Registros

As Sms possuem cada uma 64kB em memória de registros, são registros de 32 bits muito rápidos mas não podem ser acessados diretamente; o compilador é quem determina quantos registros por *thread* são necessários e os distribui. Devemos observar que o número de *threads* executando em uma SM é limitado pela quantidade de registros em uso.

Memória Local

Se a quantidade de registros em *hardware* não for suficiente, a memória local pode ser utilizada como uma armazenamento temporário. Por outro lado este tipo acesso à memória não é muito eficiente tanto em velocidade de transferên-

cia quanto em latência pois sua implementação foi feita em memória DRAM externa ao processador G92.

Memoria Compartilhada

De forma a habilitar a comunicação entre cada SP de uma SM, cada SM possui 16Kb de memória, que é implementada no *core* do processador G92, o que torna o seu acesso tão rápido quanto aos registros. A memória compartilhada ou *shared memory*, assim como a memória de registros só pode ser acessada internamente no lado do *device*, ou seja, não é possível acessar a memória compartilhada por instruções executando na CPU.

Memória Global

A *memória global* é principal banco de memória na GPU está situado em DRAM externa ao processador G92 e em nosso caso possui 1 Gbyte de espaço. A memória Global suporta regiões de: apenas leitura e leitura-escrita de dados, tanto da GPU quanto da CPU, bem como ao armazenamento de texturas. Ambos as regiões de apenas leitura e de texturas são *cached* pela GPU, mas o *hardware* não possui protocolos que detectem coerência de *cache*.

Memória de Constantes

Cada Sm possui 8kB em memória de constante *cached* e de apenas leitura. Para todos os *threads* de um *half-warp* a leitura da memória de constantes é mais rápida do que à leituras a memória de registros, pois todos os *threads* lêem apenas um endereço de memória.

Na tabela 2.1 observamos a relação entre os espaços de memória e seu tipo de acesso, a latência em ciclos de máquina e se esta memória fica no interior (INT) ou no exterior (EXT) do *chip*.

Tabela 2.1: Tipos de Acesso à Memória no CUDA

2.3.4 Programando em CUDA

Visando uma melhor compreensão da programação em CUDA, o exemplo a seguir efetua a soma de duas matrizes, B e C e coloca o resultado na matriz A;

Memória	Localização	*Thread*	Bloco	*Grid*	Latência
Local	EXT	leitura-escrita	x	x	200-300
Compartilhada	INT	leitura-escrita	leitura-escrita	x	= registro
Global	EXT	leitura-escrita	leitura-escrita	leitura-escrita	200-300
Texturas	INT cache	leitura-escrita	leitura-escrita	apenas-leitura	>100
Constantes	INT cache	leitura-escrita	leitura-escrita	apenas-leitura	= registro

```
1 void main(void)
2 {
3   int n =128, m=128, p=128
4   int pitch_a, pitch_b, pitch_c;
5
6 // Declara apontadores para a memória global da CPU
7   float *host_a, *host_b, *host_c;
8
9 // Declara apontadores para a memória global da GPU
10   __global__  float *dev_a, *dev_b, *dev_c;

// Reserva espaço de memória na CPU para as matrizes A,B e C
    host_a = (float *)malloc( m*n*sizeof(float));
    host_b = (float *)malloc( n*p*sizeof(float));
15  host_c = (float *)malloc( p*m*sizeof(float));

// Preenche com números aleatórios as matrizes B e C
    matrix_random(host_b);
    matrix_random(host_c);
20
// Reserva espaço de memória para as matrizes A, B e C no Dispositivo
// Matriz A n x m
    cudaMallocPitch( &dev_a, &pitch_a, n*sizeof(float), m );
// Matriz B n x p
25    cudaMallocPitch( &dev_b, &pitch_b, n*sizeof(float), p );
// Matriz C p x m
    cudaMallocPitch( &dev_c, &pitch_c, p*sizeof(float), m );

// Copia as matrizes da CPU para o dispositivo
30  cudaMemcpy2D( dev_b, pitch_b, host_b, n*sizeof(float), n*sizeof(float),
    p, cudaMemcpyHostToDevice );
    cudaMemcpy2D( dev_c, pitch_c, host_c, p*sizeof(float), p*sizeof(float),
                  m, cudaMemcpyHostToDevice );

35  dim3 threads(32);    // Define o número de threads (32) em cada bloco
                           // unidimensional
    dim3 grid(n/32, m);    // Define o número de Grids
// Chama a rotina na GPU

40  matrix_sum<<< grid, threads >>>( dev_a, dev_b, dev_c, pitch_a,
                                     pitch_b, pitch_c, n, m, p );

// Move os resultados da GPU para a CPU
    cudaMemcpy2D( host_a, n*sizeof(float), dev_a, pitch_a,
45                n*sizeof(float), m, cudaMemcpyDeviceToHost );
```

```
50
// Limpa a memória da CPU
   free(host_a);
   free(host_b);
   free(host_c);
55
// Limpa a memória do dispositivo
   cudaFree( dev_a );
   cudaFree( dev_b );
   cudaFree( dev_c );
60
}

__global__ void matrix_sum( float* a, float* b, float* c, int pitch_a,
                            int pitch_b, int pitch_c,  int n, int m,
65                          int p )
{
    int i = blockIdx.x*32+threadIdx.x;
    int j = blockIdx.y;
    float sum = 0.0;
70  for( int k = 0; k < p; ++k )
       sum += b[i+pitch_b*k] * c[k+pitch_c*j];
    a[i+pitch_a*j] = sum;
}
```

Observe no programa os seguintes passos,

- Declaramos as variáveis *array pointer* tanto para a CPU quanto para a GPU; linhas $6 \rightarrow 10$.
- Reservamos memória na CPU, que no contexto do CUDA é denominada *host*; nas linhas $12 \rightarrow 15$.
- Preenche as matrizes B e C com números quaisquer; linhas $18 \rightarrow 19$.
- Reservamos memória na GPU, que no contexto do CUDA é denominada *device*; nas linhas $21 \rightarrow 27$.
- Copia as matrizes da CPU para a GPU; nas linhas $30 \rightarrow 33$.
- Define o número de *threads* (32) em cada bloco; na linha 35.
- Define o número de grids de blocos; na linha 37.
- Chama a rotina que será executada na GPU; na linha 40 .

- Copia da memória da GPU para a CPU; nas linhas $44 \rightarrow 45$.
- Imprime o resultado; na linha 48.
- Retira a reserva dos espaços de memória da CPU e da GPU; nas linhas $51 \rightarrow 59$.
- Fim.

2.4 OpenCL

OpenCL (*Open Computing Language*) [39] é uma estrutura de programação que permite escrever códigos com execução em ambientes de *hardware* heterogêneos, tais ambientes compreendem os novos processadores de múltiplos núcleos bem como GPUs e FPGAs interligando-se de maneira integrada. O OpenCL é resultado de um consórcio gerenciado pelo *Khronos Group* que reúne os grandes fabricantes de *hardware* INTEL, AMD, NVIDIA, Apple e outros, e sua proposta é similar à outros padrões adotados tais como OpenAL [40] e OpenGL [41], padrões de áudio e gráficos 3D respectivamente. Este padrão em pouco tempo substituirá a programação em CUDA, de forma que à menos do compilador, específicos de cada fabricante, os códigos para GPU serão perfeitamente intercambiáveis. As principais vantagens da nova plataforma de desenvolvimento são:

- A utilização de todos os recursos computacionais do sistema.
- Paralelismo de dados e processos inerente à programação.
- Baseado na linguagem C.
- Abstração das características intrínsecas de cada plataforma de *hardware*.
- Computação em ponto flutuante e compatível com IEEE 754 [42].

Capítulo 3

Problemas Abordados e sua Modelagem

3.1 O Transporte de Nêutrons

O problema de transporte de nêutrons nos remete naturalmente à equação de transporte integro-diferencial de Boltzmman [43], e sua solução numérica nos fornece uma descrição completa do fluxo de nêutrons em um determinado problema. Uma solução do problema de transporte utilizando o método de Monte Carlo não possui tal refinamento, proporcionando ao invés, quantidades integrais, como por exemplo o fluxo médio em um determinado espaço.

Na busca de um problema representativo mas que possuísse a características de fácil implementação e depuração; e implementação de seu algoritmo, seqüencial ou paralelo, na CPU ou na GPU; optamos por investigar o problema da blindagem de nêutrons em um *slab*. Em nosso estudo de caso consideremos o fluxo de nêutrons através de uma placa infinita anisotrópica e de espessura h; além disso que o fluxo de nêutrons é mono-energético e uniforme, e atinge ortogonalmente a superfície da placa como vemos na figura 3.1.

Na interação do nêutron com a matéria este pode ser espalhado ou absorvido, e em seguida outros processos no interior do núcleo podem ocorrer como na figura 3.2. Os possíveis eventos da figura 3.2 vão depender das seções de choque de absorção,

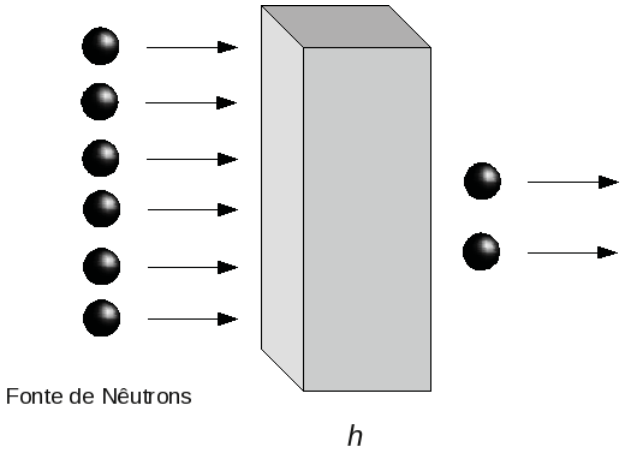


Figura 3.1: Slab geométrico do problema

espalhamento e de fissão do material. Vamos em nosso problema supor que cada partícula só pode ser absorvida ou ser espalhada; e além disso a partícula não pode ser espalhada para trás.

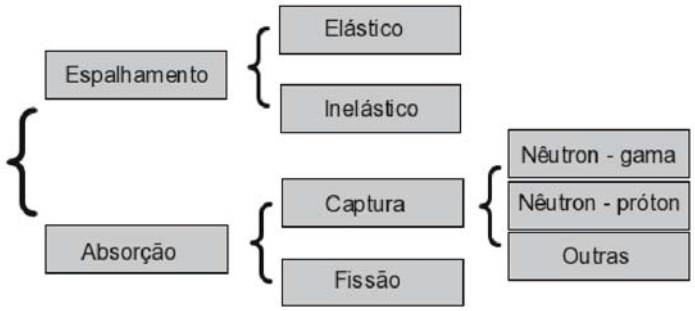


Figura 3.2: Possíveis interações do nêutron com a matéria

3.1.1 Reações Nucleares de Espalhamento

Uma reação nuclear de espalhamento pode ser imaginada como uma colisão entre o nêutron e o núcleo alvo, sendo o nêutron desviado de sua trajetória original, transferindo ao alvo, parte da energia que possuía antes da colisão. A energia perdida pelo nêutron depende da massa do núcleo alvo e do ângulo de colisão. De um modo geral, podemos dizer que um nêutron pode perder grande parte de sua energia se ele colidir com partículas com massa equivalente à sua, mas perderá somente uma pequena porção de sua energia se colidir com núcleos pesados. A perda de

energia do nêutron é também uma função do ângulo de colisão, pois em geral, há maior perda de energia em uma colisão frontal do que em uma colisão angular. O processo de espalhamento pode ser classificado como *espalhamento elástico* ou como *espalhamento inelástico*.

Espalhamento Elástico

Ocorre a partir da interação entre o nêutron e o núcleo causando uma variação da energia cinética do nêutron, energia esta que será transferida ao núcleo alvo na forma de energia de movimento, não alterando porém sua energia interna. O espalhamento elástico pode ser considerado uma reação (n, n), onde a energia interna do núcleo alvo `x` não é alterada, mas havendo em geral, uma troca de energia cinética entre o nêutron e o núcleo alvo.

Espalhamento Inelástico

O espalhamento inelástico pode ser considerado como uma reação $n \longrightarrow n'$ onde o nêutron é absorvido pelo núcleo alvo formando um núcleo composto que decai emitindo "outro nêutron"com menor energia. Grande parte da energia perdida pelo nêutron incidente aparece como energia de excitação do núcleo alvo, sendo emitida posteriormente, na forma de raios γ. O espalhamento inelástico predomina quando o meio é constituído por núcleos pesados e os nêutrons possuem alta energia (nêutrons rápidos). Por outro lado, o espalhamento elástico é predominante quando o meio é constituído por núcleos leves e os nêutrons são de baixa energia. Ambos os tipos de espalhamento (elástico e inelástico) produzem o mesmo resultado final que é a redução da energia dos nêutrons.

3.1.2 Seção de Choque

A seção de choque para um dado núcleo é definida como sendo a medida da probabilidade de que este núcleo sofra uma determinada reação com um nêutron. Quanto maior a seção de choque de um núcleo, maior será a probabilidade de sua reação com nêutrons. Devemos esperar que a probabilidade de que um nêutron sofra reação com um núcleo seja dependente do tamanho deste núcleo ou da área que o

mesmo apresente como alvo. Quanto maior a área apresentada pelo alvo, maior será a chance do nêutron em atingi-lo-lo, causando a reação.

Um núcleo pode sofrer vários tipos de reações nucleares, para cada tipo de reação nuclear existe uma seção de choque correspondente que é uma característica do núcleo alvo, sendo entretanto, função da energia do nêutron incidente. A seção de choque total de um núcleo alvo com nêutrons de determinada energia é a somatória das seções de choque individuais para cada tipo de reação nuclear. A seção de choque total é usualmente dividida em duas grandes componentes que são relativas às reações de absorção e reações de espalhamento. A seção de choque de espalhamento é, por outro lado, a somatória das seções de choque das reações de espalhamento elástico e de espalhamento inelástico. A seção de choque para um núcleo individual é expressa pelo símbolo σ, também denominada por *seção de choque microscópica*. Assim, podemos escrever que,

$$\sigma_t = \sigma_a + \sigma_s \tag{3.1}$$

Onde σ_t é a seção de choque microscópica total, σ_a é a seção de choque microscópica de absorção e σ_a é a seção de choque microscópica de espalhamento.

3.1.3 Seção de Choque Macroscópica

A seção de choque para um centímetro cúbico de material alvo é chamada seção de choque macroscópica Σ, e é o produto entre a seção de choque microscópica e o número total de núcleos em um centímetro cúbico do alvo, isto é,

$$\Sigma = N \cdot \sigma \tag{3.2}$$

Deste modo podemos definir,

$$\Sigma_c = N \cdot \sigma_c \quad \text{Seção de choque de captura.} \tag{3.3}$$

$$\Sigma_f = N \cdot \sigma_f \quad \text{Seção de choque de fissão.} \tag{3.4}$$

$$\Sigma_a = \Sigma_c + \Sigma_f \quad \text{Seção de choque de absorção.} \tag{3.5}$$

$$\Sigma_{se} = N \cdot \sigma_{se} \quad \text{Seção de choque de espalhamento elástico.} \tag{3.6}$$

$$\Sigma_{si} = N \cdot \sigma_{si} \quad \text{Seção de choque de espalhamento inelástico.} \tag{3.7}$$

$$\Sigma_s = \Sigma_{se} + \Sigma_{si} \quad \text{Seção de choque de espalhamento.} \tag{3.8}$$

Assim, temos que a seção de choque macroscópica total é dada por,

$$\Sigma_t = \Sigma_a + \Sigma_s \tag{3.9}$$

3.1.4 Livre Caminho Médio

A distância média percorrida por um nêutron sem sofrer uma reação é expresso por seu livre caminho médio. Esta distância γ é dada por,

$$\gamma = \frac{1}{\Sigma_t} \tag{3.10}$$

3.1.5 Método de Monte Carlo

A utilização de amostragens aleatórias como forma de resolver problemas matemáticos remonta ao Conde de Buffon e a Laplace, contudo seu grande desenvol-

vimento ocorreu durante a segunda grande guerra mundial, durante o projeto Manhatan e através dos trabalhos de Fermi, Ulam, Metropolis e Von Neumman que o utilizaram para solucionar problemas de transporte de partículas; podemos atribuir ao celebre trabalho publicado por Metropolis e Rosenbluth[44] ao nascimento desta técnica nos moldes atuais. O objetivo essencial do método de Monte Carlo é estimar médias de propriedades sobre um determinado conjunto denominado *ensemble*.

Ora, em mecânica estatística[45], um *ensemble* é um conjunto de vários sistemas que, apesar de suas condições iniciais diferentes, são idênticos a um sistema estatisticamente considerado. O método de Monte Carlo se distingue de outras técnicas de analise numérica por aproximar as propriedades médias do *ensemble* através de um passeio aleatório no espaço amostral de um sistema. O espaço amostral do sistema é representado por um conjunto, com um número finito de estados que representa as possíveis propriedades físicas sob analise, tais como temperatura, pressão e volume.

E portanto, o método de Monte Carlo, através de um modelo estocástico e com uma amostragem adequada de uma distribuição de probabilidades, pode estimar com grande precisão a solução numérica de problemas que de outra forma demandariam simplificações tais que dificultariam a validação dos resultados. No caso de problemas relacionados ao transporte de partículas [46] [47], em contraste aos métodos determinísticos ou outras soluções interativas da equação de Boltzmann; a complexidade geométrica das condições de contorno não é determinante para a aproximação da solução. Além disso podemos observar que acurácia do método de Monte Carlo depende basicamente do desvio estatístico das quantidades à serem estimadas, tornando fundamental um elevado número de simulações, bem como de um gerador de números aleatórios com distribuição não tendenciosa.

Ligado ao fato de à toda estatística estarem associados um valor médio e um erro em função da distribuição, a variância, e de modo a reduzir tal erro estatístico à valores aceitáveis, devemos efetuar um número suficientemente grande de simulações, o que pode representar eventualmente um longo tempo de processamento.

3.1.6 Processo de Poisson

O processo de Poisson é um processo estocástico que pode ser definido em termos de ocorrências de eventos. Ela expressa a probabilidade de um certo número de

eventos ocorrerem num dado período tempo, caso estes ocorram com uma taxa média conhecida e caso cada evento seja independente do tempo decorrido desde o último evento. Ora, se denotarmos este processo por $\{N_t\}_{t\geqslant 0}$ e fixarmos o tempo no instante t temos então que N_t é um número inteiro que representa o número de eventos até o instante t.

O processo de Poisson é portanto um processo estocástico em tempo contínuo, $t \in (0, \infty]$, e possui um espaço de estados $E = \mathbb{N}$. Por outro lado a interpretação física das seções de choque de absorção, espalhamento e do livre caminho médio nos diz que a probabilidade de absorção é dada por,

$$P_a = \frac{\Sigma_a}{\Sigma} \tag{3.11}$$

a probabilidade de espalhamento é dada por :

$$P_s = \frac{\Sigma_s}{\Sigma} \tag{3.12}$$

Ora, o livre caminho médio nos fornece a distancia entre duas colisões consecutivas e vamos considerar cada colisão um evento independente vemos que este descreve um processo de Poisson e a nossa variável aleatória é o livre caminho médio que denotamos por γ, pode assumir qualquer valor positivo e possui uma densidade de probabilidade dada por,

$$P(x) = \Sigma_t e^{-\Sigma_t x}, \quad 0 < x < \infty \tag{3.13}$$

ou seja,

$$P(x) = \frac{1}{\gamma} e^{-\frac{1}{\gamma}x}, \quad 0 < x < \infty \tag{3.14}$$

Ora, a função de densidade de probabilidade $P(x)$ define a probabilidade de ocorrência da variável aleatória γ enquanto que a função de distribuição de probabilidade $F(x)$ nos define a probabilidade de a variável aleatória γ estar em um certo intervalo. De forma que a função distribuição de probabilidade é dada por,

$$F(x) = \int_0^\infty P(x)dx = \int_0^\infty \frac{1}{\gamma} e^{-\frac{1}{\gamma}x} dx, \quad 0 < x < \infty \tag{3.15}$$

3.1.7 Seqüencias Randômicas

Na verdade não existe um número que possamos chamar de aleatório, podemos dizer sim, que existem seqüências de números aleatórios. Estas podem ser entendidas como seqüências de números independentes com uma específica distribuição. Assim, em uma seqüência de números aleatórios, cada número aparece de forma independentemente dos anteriores, sem nenhuma possibilidade de previsão. Diversos mecanismos podem ser utilizados para testar a qualidade da nossa seqüencia de números randômicos, podemos citar o teste de Kolmogorov-Smirnov [48], o teste de correlação e o teste do chi-quadrado.

De forma para passar no teste de aleatoriedade [49], uma dada seqüencia numérica precisa ter uma certa distribuição e não deve haver nenhuma correlação perceptível entre os números gerados. Se por exemplo, jogarmos uma moeda e os resultados forem C, K, C, K, C, K, C, K, C, K, C, K, C, K, C, K,... Essa seqüencia passa no teste da distribuição, pois resultados estão uniformemente distribuídos entre K e C, mas não passam no teste da correlação já que a seqüencia se repete sendo assim facilmente previsível.

As variáveis aleatórias de uma dada distribuição de probabilidade serão as entradas para o nosso modelo e são geradas através de uma seqüencia de números aleatórios com distribuição uniformemente. Na geração dessas variáveis aleatórias vamos utilizar o método da transformação inversa [50].

$$F(x) = \int_0^\infty P(x)dx = \int_0^\infty \Sigma_t e^{-\Sigma_t x} dx = \varepsilon, \quad 0 < x < \infty \tag{3.16}$$

Assim, podemos gerar variáveis aleatórias γ resolvendo esta integral por substituição,

$$u = -\Sigma_t x \quad \text{e} \quad du = -\Sigma_t dx \tag{3.17}$$

Substituindo vemos que,

$$\int_0^u -e^u du = \varepsilon \Longrightarrow -e^u|_0^u = \varepsilon \Longrightarrow -(e^{\Sigma_t x} - 1) = \varepsilon \tag{3.18}$$

$$\Longrightarrow -\Sigma_t x = \ln(1-\varepsilon) \Longrightarrow x = -\frac{1}{\Sigma_t}\ln(1-\varepsilon) \tag{3.19}$$

Observamos que o número $(1-\varepsilon)$ possui a mesma condição aleatória que ε no intervalo $(0,1]$, de forma que podemos escrever a equação que relaciona a variável aleatória γ em função do número aleatório ε como,

$$\gamma = -\frac{1}{\Sigma_t}\ln\varepsilon \tag{3.20}$$

O método de Monte Carlo caracteriza o resultado médio de uma análise estatística e portanto seus resultados são expressos em termos médios. Definimos o valor médio de uma variável aleatória contínua por,

$$\overline{x} = \int_{-\infty}^{\infty} x f(x) dx \tag{3.21}$$

Ora, a variável aleatória é estimada através de um número finito de números aleatórios ou histórias, e a equação anterior é computada com o somatório de cada história [51],

$$\overline{x} = \sum_{i=1}^{n} \gamma_i \tag{3.22}$$

onde, n é o numero de histórias simuladas.

A seguir apresentamos os algoritmos seqüencial e paralelo desenvolvidos para a aplicação.

3.1.8 Os Algoritmos

```
NH {Número de histórias}
WIDTH {Espessura do Material}
N {Número de Nêutrons que passam pela blindagem}
γ {Variável Randômica}
for i = 1 to NH do
  dx = 0 {Passo Incremental no Caminho do Nêutron}
  while new is false do
    U = Random()
    dx = −MeanFreePath(ln(U)) {Calcula o caminho do nêutron aleatoriamente}
    x ← x + dx {Olhe a Equação 3.22}
    if x > WIDTH then
      N ← N + 1 {Incrementa o Número de Nêutrons que saíram}
      new ← true
    end if
    U = PA(Random()) {Estima a Probabilidade de absorção}
    if PA ≥ 1 then
      new ← true {Se o nêutron foi absorvido}
    end if
  end while
end for
```

Figura 3.3: O Algoritmo Seqüencial

NHp {Número de histórias}

WIDTHp {Espessura do Material}

Np {Número de Nêutrons que passam pela blindagem}

$T[NT]$ {Reserva um vetor de *threads*}

for $j = 1$ to NT **do**

  $T_j \leftarrow$ start {Inicia o *thread* T_j}

  N_j {Vetor do Número de nêutrons que escapam da blindagem}

  γ_j {Vetor de Variáveis Randômicas}

  NHj=NH/NT {Número de histórias por *thread*}

  for $i = 1$ to NHj **do**

    $dx_j = 0$ {Variável do Caminho do Nêutron}

    while new is false **do**

      $U = Random()$

      $dx_j = -MeanFreePath(\ln(U))$ {Passo Incremental no Caminho do Nêutron}

      $x_j \leftarrow x_j + dx_j$ {Veja a equação 3.22}

      if $x_j > WIDTH$ **then**

        $N_j \leftarrow N_j + 1$ {Incrementa o Número de Nêutrons que Saíram}

        $new \leftarrow true$

      end if

      $U = PA(Random())$ {Calcula a probabilidade de Absorção}

      if $PA \geq 1$ **then**

        $new \leftarrow true$ {Se o Nêutron foi Absorvido}

      end if

    end while

  end for

end for

for $j = 1 \rightarrow NT$ **do**

  Tj $\leftarrow$ stop {Espera os *threads* T_j terminarem}

  $N \leftarrow N_j + N$ {Incrementa o Número de Nêutrons que saíram no Total}

end for

Figura 3.4: O Algoritmo Paralelo

3.1.9 Implementação na CPU

A implementação na CPU se fez com 1, 2, 4 e 8 *threads* e consiste em dividir o número de nêutrons em partes iguais e distribui-las entre os *threads* que executarão a função *history*. Ao final de cada processo retornamos o número de nêutrons que ultrapassou a blindagem e efetuamos a soma total. Como podemos ver nas figuras 3.4 e 3.5

```
struct TMeio{
#ifdef DOUBLET
    double Sigma_s; // Secao de choque macroscopica de espalhamento
    double Sigma_a; // Secao de choque macroscopica de absorcao
    double Width; // Espessura
    double ret; // retorna o que passou/NH
#else
    float Sigma_s; // Secao de choque macroscopica de espalhamento
    float Sigma_a; // Secao de choque macroscopica de absorcao
    float Width; // Espessura
    float ret; // retorna o que passou/NH
#endif
    unsigned long NH;
    long Seed; // Semente randomica
};

int main(void)
{
struct TMeio Meio;
#ifdef DOUBLET
      double x;
#else
      float x;
#endif
pthread_t t[NUM_THREADS];          // define NUM threads
pthread_attr_t attr;
unsigned long int Count;           // Neutrons que chegam
int eoh;                           // Fim de uma historia
unsigned long NH=1024*512,n;       // Numero de historias
int inc;
                 // Caracteristicas do material
Meio.Sigma_s=0.84;
Meio.Sigma_a=0.014;
Meio.Width=10.0;

pthread_mutex_init(&Meio.mutexsum, NULL); // Inicia o mutex
pthread_attr_init(&attr);
```

Código fonte Monte Carlo CPU

```
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
int thread[NUM_THREADS];

                  // loop de 2^20 ateh 2^28
   for(inc=0;inc<8;inc++){
     NH*=2; // multiplica o Num de historias
     Meio.ret = 0;
        // Cria um thread para cada uma das funcoes
     Meio.NH = NH/NUM_THREADS;
     struct TMeio M[NUM_THREADS];
     for(n=0;n<NUM_THREADS;n++){
       M[n]=Meio;
       M[n].Seed = SeedTh[n];
       thread[n] = pthread_create( &t[n], &attr,
                   (void *)&history, (void *) &M[n]);
       if(thread[n]){
         printf("erro na criacao do thread");
         pthread_exit(NULL);
         exit(1);
         }
      }
      pthread_attr_destroy(&attr);
      for(n=0;n<NUM_THREADS;n++){
      pthread_join( t[n], NULL);
   }
   x = 0;
   for(n=0;n<NUM_THREADS;n++)
     x += M[n].ret;
   printf("Percentual de neutrons :%f \n",x);
  }
  pthread_attr_destroy(&attr);
  pthread_exit(NULL);
  return 1;
}
```

Figura 3.5: Código fonte Monte Carlo CPU

Podemos observar na figura 3.6 a rotina *history* que efetivamente avalia a história de cada nêutron.

```
void* history(void *pt){
unsigned long n,Count=0; int eoh;
#ifdef DOUBLET
      double rnd;       // Numero randomico
      double x, dx;     // Posicao e variacao
#else
      float rnd;
      float x, dx;
#endif
struct TMeio *Meio;

Meio = (struct TMeio *)pt;
long Seed = Meio->Seed;
   for(n=0; n<Meio->NH; n++){
     x = 0; eoh = 0;
     do {
       rnd = Rand();
       dx = -MeanFreePath(Meio)*log(rnd);
       x += dx;
       if (x > Meio->Width) {
         Count++;
         eoh = 1; // fim de historia
       }
       else if (Rand() < Pa(Meio)) eoh = 1;
       } while(!eoh);
     }
Meio->ret=
#ifdef DOUBLET
(double)Count/(double)Meio->NH/NUM_THREADS;
#else
(float)Count/(float)Meio->NH/NUM_THREADS;
#endif
pthread_exit((void *) 0);
}
```

Figura 3.6: Código fonte da rotina *history* na CPU

3.1.10 Implementação na GPU

Na execução do programa uma parte do código é responsável pela interação entre a CPU e a GPU: definir o tamanho do grid, reservar memória, fazer o somatório do retorno do programa que executa no *thread*. Podemos ver seu código nas figuras 3.7 e 3.8

```
int main(int argc, const char **argv){
float *d_Random, *h_Random;
double gpuTime;
unsigned int hTimer;

CUT_DEVICE_INIT(argc,argv);              // Inicia a GPU
CUT_SAFE_CALL( cutCreateTimer(&hTimer) );// Cria o timer
h_Random = (float *)malloc(RAND_N *sizeof(float));// Aloca espaco no host
int loop = iDivUp(PATH_N,BLOCK);
initMonteCarloGPU(4096);                 // reserva espaco na GPU
long mul = PATH_N;
loop = 8;  printf("inicia loop");

for(int n =0;n<loop;n++){
  CUT_SAFE_CALL( cutResetTimer(hTimer) );      // Limpa o timer e o inicia
  CUT_SAFE_CALL( cutStartTimer(hTimer) );      // 32 blocos de 128 threads
  MonteCarloGPU(&val,Ss,Sa,ESPESSURA,BLOCK_N,d_Random,mul);
  CUDA_SAFE_CALL( cudaThreadSynchronize() );   // sincroniza os threads
  CUT_SAFE_CALL( cutStopTimer(hTimer) );       // para o timer
  gpuTime = cutGetTimerValue(hTimer);          // retorna o valor do timer
  printf("Tempo de GPU  : %f ms", gpuTime);
  mul*=2;
  }
CUDA_SAFE_CALL( cudaFree(d_Random) );
free(h_Random);
closeMonteCarloGPU();
CUT_SAFE_CALL( cutDeleteTimer( hTimer) );
CUT_EXIT(argc, argv);
}

void MonteCarloGPU(
    float *calculo,  // Retorna o percentual de neutrons
    float Sigma_s,   // Secao de choque de espalhamento
    float Sigma_a,   // Secao de Choque de absorcao
    float Width,     // Espessura da blindagem
    int N,           // Numero de Blocos de Grids
    float *d_Random, // N(0, 1) vetor de amostras randomicas
    long pathN       // Numero de historias
){
dim3 dimBlock(BLOCK);
dim3 dimGrid(BLOCK);
unsigned long soma=0;

float Pa =Sigma_a/(Sigma_a+Sigma_s);
float Lcm = -1.0f/(Sigma_s+Sigma_a);
float Ps = Sigma_s/(Sigma_a+Sigma_s);
```

Figura 3.7: Código fonte da rotina MonteCarlo, na GPU

```
MonteCarloKernel<BLOCK><<<dimGrid, N>>>(d_Sum, Pa, Ps,Lcm, Width,
                                        d_Random, pathN/(BLOCK*N));
CUT_CHECK_ERROR("Falha na execucao MonteCarloKernel()\n");
CUDA_SAFE_CALL( cudaMemcpy(h_Sum, d_Sum, 4096 * sizeof(unsigned long),
                           cudaMemcpyDeviceToHost) );
checkCUDAError("Erro de copia para a memoria");
for (int count = 0; count < BLOCK*N; count++){
  soma  += h_Sum[count];
}
*(calculo) = (double)soma/(double)pathN;
}
```

Figura 3.8: Código fonte da rotina Monte Carlo, continuação

Os programas que executam nos *threads* da GPU denominamos *kernels*, o kernel do programa de Monte Carlo calcula a história de cada nêutron avaliando a probabilidade de absorção e espalhamento. Seu código fonte pode ser visualizado na figura 3.9.

```
template <unsigned int blockSize>
__global__ void MonteCarloKernel(
    unsigned long *Sum, // endereco das Somas parciais
    float Pa,           // Probabilidade de absorcao
    float Ps,// Probabilidade de espalhamento
    float Lcm,          // Livre caminho medio
    float Width,// Espessura da blindagem
    float *d_Random,  // N(0, 1) vetor de amostras randomicas
    long   pathN        // Numero de historias do bloco
){
float rnd,x,dx,eoh;
unsigned long Count;
int i = blockIdx.x * blockDim.x + threadIdx.x;
Sum[i]=0; // Zera o numero de neutrons
Count =0;
```

Figura 3.9: Código fonte da rotina *kernel*, na GPU

```
for(int iPath=0; iPath < pathN;iPath++)
  {
   x = 0,eoh=0,dx=0;
   do {
// Desloca particula em x
     rnd = Rand();
     dx = Lcm*log(rnd);
     x += dx;
     if (x > Width){
       Count++;
       eoh = 1; // fim de historia
     }
// Testa a probabilidade de absorcao
     if(Rand()< Pa)
       eoh = 1;
   } while(!eoh); // se nao foi absorvido nem passou descarta
}
if(eoh){
// Salva o valor de Count de cada thread no vetor d_Sum
   Sum[i]+=Count;
  __syncthreads();
}
}
```

Figura 3.10: Código fonte da rotina *kernel*, continuação

3.1.11 Resultado Analítico

A estimativa do fluxo de neutrons através da blindagem é obtida de maneira determinística resolvendo a equação integro diferencial de Boltzmman. A equação de Boltzmann é originalmente relacionada aos estudos de Ludwig Boltzmann sobre teoria cinética dos gases e também conhecida como equação do transporte [43]. Na prática a estimativa do fluxo é obtida resolvendo uma série de $Liouville-Neumann$ obtida de modo à resolver a integral de Fredholm associada ao problema [52]. O resultado com boa aproximação à partir de nossas premissas é dado por 4.1.

$$I(x) = I_o e^{-\Sigma_a x} \tag{3.23}$$

3.2 Problema da Transferência de Calor

3.2.1 A equação do Calor

No nosso problema vamos abordar o problema de determinar o fluxo de calor em uma placa, fina e retangular, composta por algum material condutor de calor e sujeita a uma fonte externa de calor. Vamos considerar também que que o material é homogêneo e as condições iniciais e de contorno variam apenas com x e y, e podem ser modeladas em uma equação diferencial à duas variáveis. Seja $u(x, y, t)$ a função de distribuição de temperatura no ponto (x, y) e no instante t, onde $(x, y) \in \{(x, y) \in \mathbb{R}^2 | 0 < x < a \text{ e } 0 < y < b)\}$, neste caso a solução do problema tem a forma da equação do Calor,

$$u_t = (k(x, y)u_x)_x + (k(x, y)u_y)_y + \psi \tag{3.24}$$

Onde $k(x, y)$ é o coeficiente de difusão que pode variar com x e y e $\psi(x, y, t)$ é o termo de fonte. Vamos supor que as condições de contorno e o termo de fonte são independentes do tempo e vamos encontrar a solução para o problema estacionário. Vamos supor ainda que $k(x, y) = 1$ e portanto de 3.24 temos,

$$(k(x, y)u_x)_x + (k(x, y)u_y)_y = -\psi \tag{3.25}$$

Denominado *problema de Poisson* com condições de contorno de Dirichlet, por outro lado, se $\psi(x, y, t) = 0$, o chamamos *problema de Laplace*.

Podemos observar também que se houvesse um termo de dissipação de calor proporcional à posição (x, y) a equação do Calor seria dada por,

$$(k(x, y)u_x)_x + (k(x, y)u_y)_y + \nu u_y)_y = -\psi \tag{3.26}$$

e seria chamada de *equação de Helmholtz*.

$$(k(x,y)u_x)_x + (k(x,y)u_y)_y = 0 \tag{3.27}$$

Uma das maneiras de resolver numericamente este problema é fornecida pelo método das diferenças finitas.

3.2.2 Série de Taylor Real

Seja $u\colon I \to \mathbb{R}$ uma função definida no intervalo aberto I, possuindo derivadas de todas as ordens em $x_0 \in I$. Então sua série de potências é dada por,

$$T(x) = \sum_{k=0}^{\infty} \frac{u^{(k)}(x_0)}{k!}(x - x_0)^k \tag{3.28}$$

é chamada de *série de Taylor* de u centrada em x_0. Se efetuarmos um diferenciação termo a termo de $T(x)$ obteremos que $T^{(k)}(x_0) = f^{(k)}(x_0)$, e portanto vemos que T é uma aproximação de u baseada nas derivadas de u no ponto x_0. Assim podemos escrever,

$$u(x) = u(x_0) + \sum_{k=1}^{\infty} \frac{u^{(k)}(x_0)}{k!}(x - x_0)^k \tag{3.29}$$

$$u(x) - u(x_0) = (x - x_0)u'(x_0) + \frac{(x - x_0)}{2}u''(x_0) + O(h^2) \tag{3.30}$$

onde $O(h^2)$ representa o erro relativo ao truncamento da série. Fazendo $(x - x_0) = h$ temos que,

$$u'_+(x_0) = \lim_{h\to 0} \frac{u(x_0 + h) - u(x_0)}{h} \tag{3.31}$$

$$u'_-(x_0) = \lim_{h\to 0} \frac{u(x_0) - u(x_0 - h)}{h} \tag{3.32}$$

A aproximação centrada nos dá de 3.31 e 3.32

$$u'(x_0) = \frac{u(x_0 + h) - u(x_0 - h)}{2h} \tag{3.33}$$

A derivada de 2^a ordem de u,

$$u''_+(x_0) = \lim_{h\to 0} \frac{u'(x_0+h) - u'(x_0)}{h}$$
$$= \frac{1}{h}[\frac{u(x_0+2h) - u(x_0)}{2h} - \frac{u(x_0+h) + u(x_0-h)}{2h}] \tag{3.34}$$

$$u''_-(x_0) = \lim_{h\to 0} \frac{u'(x_0) - u'(x_0-h)}{h}$$
$$= \frac{1}{h}[\frac{u(x_0+h) - u(x_0-h)}{2h} - \frac{u(x_0) + u(x_0-2h)}{2h}] \tag{3.35}$$

Tomando as diferenças centradas em 3.34 e 3.35 temos,

$$u''_0(x_0) = \frac{[u(x_0-h) - 2u(x_0 + u(x_0+h)]}{h^2} - O(h^2) \tag{3.36}$$

E portanto a equação 3.36 define um esquema de diferenças centradas em uma dimensão. O próximo passo consiste em construir através de uma expansão em série de Taylor em duas dimensões o mesmo esquema de diferenças centradas de 2^a ordem. Em termos bidimensionais temos que 3.25 é dada por [53],

$$\frac{1}{\delta x^2}[u_{i-1,j} - 2u_{i,j} + u_{i+1,j}] + \frac{1}{\delta y^2}[u_{i,j-1} - 2u_{i,j} + u_{i,j+1}] = \psi_{i,j} \tag{3.37}$$

Vamos supor que $\delta x = \delta y = h$, e que para um h suficiente pequeno o erro de truncamento $O(h^2)$ tende à ser um número muito pequeno; logo de 3.37,

$$\frac{1}{h^2}[u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{i,j}] = \psi_{i,j} \tag{3.38}$$

Tomemos um *grid* cartesiano uniforme, consistindo de pontos (x_i, y_j) onde $x_i = ih = y_j$,

$$h = \frac{1}{m+1} \begin{cases} i = 1 \ldots m \\ \\ j = 1 \ldots m \end{cases} \tag{3.39}$$

$$\therefore u_{i,j} = \frac{[u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - \psi_{i,j}h^2]}{4} \tag{3.40}$$

Com as seguintes condições de contorno,

$$u_{i,j} = \begin{cases} 0 & \text{Se} \quad i = 0, \quad \forall j \\ 100 & \text{Se} \quad j = 0 \quad \text{ou} \quad j = m \quad \text{ou} \quad i = m \end{cases} \tag{3.41}$$

Este esquema de diferenças finitas pode ser representado por um *stencil* de cinco pontos como mostrado na figura 3.11

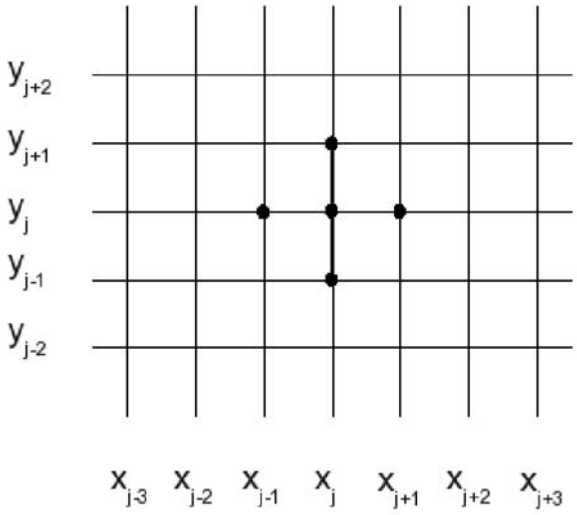


Figura 3.11: O *stencil* de 5 pontos para o Laplaciano sobre o ponto (x_i, y_j)

Por outro lado como supomos $f \equiv 0$ temos que,

$$u_{i,j} = \frac{(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1})}{4}, \quad i = 0 \ldots m \quad \text{e} \quad j = 0 \ldots m \tag{3.42}$$

Definição O raio espectral de uma matriz M é dado por $\rho(M) = \max_{i=1,\ldots,n} |\lambda_i|$ onde $\lambda_1, \ldots, \lambda_n$ representam os autovalores de M.

Definição Dizemos que uma matriz M é positiva definida se a forma quadrática associada a M, $Q_M(x) > 0, \forall x \neq 0$. Isto é equivalente a dizer que, se M é uma matriz tal que $M = M^T$ então M é positiva definida se e somente se $x^T M x > 0 \forall x \neq 0$, ou ainda que todos os seus autovalores sejam positivos.

Definição Uma matriz M é dita *estritamente diagonal dominante* se, para todas as linhas ou colunas da matriz, o módulo do valor da matriz na diagonal é maior que a soma dos módulos de todos os demais valores daquela linha ou coluna. Mais precisamente, a matriz M é de diagonal dominante se,

$$|a_{i,i}| > \sum_{i \neq j} |a_{i,j}| \quad \forall a_{i,j} \tag{3.43}$$

A equação 3.42 nos fornece um sistema de equações lineares com m^2 incógnitas e uma matriz A quadrada associada ao problema de tamanho $m^2 \times m^2$, esta possui uma forma em blocos de acordo com a representação na matriz 3.1.

Matriz 3.1: Matriz de Blocos

$$A = \frac{1}{h^2} \begin{pmatrix} T & I & & & & & \\ I & T & I & & & & \\ & I & T & I & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & & & I & T \end{pmatrix}$$

A matriz de blocos 3.1 é tri-diagonal, esparsa e diagonal dominante, e além disso cada bloco I é uma matriz identidade de tamanho $m \times m$. Ora, cada bloco T também é uma matriz tri-diagonal, esparsa e diagonal dominante como podemos ver na matriz 3.2,

Matriz 3.2: Sub-Matriz de Blocos

$$T = \begin{pmatrix} -4 & 1 & & & & & \\ 1 & -4 & 1 & & & & \\ & 1 & -4 & 1 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & & & 1 & -4 \end{pmatrix}$$

A matriz 3.1 é positiva definida, por 3.2.2, e diagonal dominante, por 3.2.2, de forma que Meis et al [54] (teoremas 13.3 à 13.13) nos mostra que o método de diferenças finitas é consistente e estável e portanto converge [55].

Existem diversas maneiras de resolver o problema $Au = b$, tanto por métodos determinísticos como as fatorações LU ou QR ou ainda o método de Gauss; quanto por métodos iterativos, dentre os quais podemos destacar o procedimento iterativo de Jacobi [56], o método do gradiente conjugado [57] ou ainda o método de Gauss-Seidel [58]; e nosso trabalho está focado em implementar este último.

3.2.3 O Método de Gauss-Seidel

O método de Gauss-Seidel é um algoritmo iterativo que nos aproxima da solução do problema $Ax = b$ e consiste inicialmente, sob a forma matricial, em escolher uma matriz D como a diagonal de A. Fazendo $D - A = -(L + U)$ onde L e U designam matrizes estritamente triangulares inferior e superior respectivamente. Nesta decomposição supomos que D é uma matriz invertível, o que implica que os seus elementos diagonais sejam todos diferentes de zero. Caso algum elemento da diagonal seja zero devemos fazer uma troca apropriada de linhas de modo que todos os elementos da diagonal de A sejam diferentes de zero. Assim, como D é invertível,

$$\begin{aligned} D^{-1}Au &= D^{-1}f \\ &= D^{-1}(L+U)u + D^{-1}Du = D^{-1}f \\ &= D^{-1}(L+U)u + Iu = D^{-1}f \end{aligned} \tag{3.44}$$

$$Iu = D^{-1}f - D^{-1}(L+U)u \tag{3.45}$$

$$u = D^{-1}f - D^{-1}Lu - D^{-1}Uu \tag{3.46}$$

O procedimento iterativo de Gauss-Seidel consiste em, após imprimir uma condição inicial $u^{(0)}$ ao sistema, supor que a seqüencia u^k,onde $k \in \mathbf{N}$, seja convergente e convirja para um ponto fixo; deste modo a solução $u^{(k+1)}$ é tal que,

$$u^{k+1} = D^{-1}f - D^{-1}Lu^{k+1} - Uu^k \tag{3.47}$$

Mas em nosso problema $f \equiv 0$ então,

$$u^{k+1} = -D^{-1}Lu^{k+1} - Uu^k \tag{3.48}$$

$$u_{i,j}^{(k+1)} = -\frac{\sum_{j=1}^{i-1} a_{i,j}u_j^{(k+1)} + \sum_{j=i+1}^{n} a_{i,j}u_j^{(k)}}{a_{i,i}} \tag{3.49}$$

3.2.4 Análise do Erro

Uma análise aprofundada da convergência e do erro associado a aproximação da solução pelo método de Gauss-Seidel pode ser encontrada em [53] e [56]. Assim, se tomamos $e \in \mathbb{R}^m$, onde $m \in \mathbb{N}$ e $e^{(k)}$ é o vetor associado ao erro em sua k-ésima iteração então,

$$e^{(k+1)} = u^{(k+1)} - u^{(k)} = D^{-1}(L+U)(u^{(k)} - u^{(k-1)}) = D^{-1}(L+U)e^{(k)} \tag{3.50}$$

Vamos chamar $B = D^{-1}(L+U)$ e então,

$$e^{(k+1)} = Be^{(k)} = B^{(k+1)}e^{(0)}. \tag{3.51}$$

A equação 3.51 nos dá a forma matricial para o cálculo do erro. Ao efetua o cálculo do *stencil* tomamos o erro absoluto como o módulo da diferença para cada $u_{i,j}$ entre seu valor na iteração k e a seguinte, ou seja,

$$E = max\{|u_{i,j}^{(k+1)} - u_{i,j}^{(k)}|; \forall i, \forall j\}. \tag{3.52}$$

3.2.5 Critério de Convergência

Teorema 3.2.1 *O* $\lim_{k\to\infty} \|B^k\| = 0$ *se e somente se o raio espectral da matriz B, que denotamos por* $\rho(B)$*, é menor do que 1.*

Teorema 3.2.2 *Se uma matriz M é estritamente diagonal dominante então:* $|a_{i,i}| > \sum_{i\neq j} |a_{i,j}|, \forall i = 1, \ldots, n$*, então* $\forall x_0$ *o algorítimo de Gauss-Seidel converge para a solução do sistema* $Au = b$*.*

O algorítimo de Gauss-Seidel converge se $\lim_{k\to\infty} \|e^{(k)}\| = 0 \Leftrightarrow \lim_{k\to\infty} \|B^k\| = 0$

A seguir apresentaremos os algoritmos seqüencial e paralelo do método.

3.2.6 O Algoritmos do Método de Gauss-Seidel

O Algoritmo Seqüencial de Gauss-Seidel

Neste caso o processo iterativo de atualização é seqüencial, componente por componente, e no momento de realizar-se a atualização das componentes do vetor $u_{i,j}^{k+1}$ numa determinada iteração, a formulação utiliza algumas componentes já atualizadas na iteração atual $u_{i+1,j}$ e $u_{i,j+1}$, e não atualizadas da iteração anterior $u_{i,j-1}$ e $u_{i-1,j}$.

Tomemos $u^0 \in \mathbb{R}^{NX \times NY}$

N {Número de interações}

NX {Número de pontos do grid na ordenada}

NY {Número de pontos do grid na abcissa}

for $k = 1$ to N **do**

  for $i = 1$ to NX **do**

    for $j = 1$ to NY **do**

      $u_{i,j}^{(k+1)} = \frac{1}{4}[u_{i-1,j}^{(k)} + u_{i+1,j}^{(k+1)} + u_{i,j-1}^{(k)} + u_{i,j+1}^{(k+1)}]$

      $j \leftarrow j + 1$

      $e \leftarrow |u - u_{old}|$ {Avalia o erro}

    end for

    $i \leftarrow i + 1$

  end for

  $k \leftarrow k + 1$

end for

Figura 3.12: O Algoritmo Seqüencial

O Algorítimo Paralelo

Duas características importantes do método de Gauss-Seidel devem ser mencionadas, em primeiro lugar os cálculos se apresentam de maneira serial, visto que cada componente em uma iteração depende do componentes previamente calculados; na outra as novas interações u^k dependem da ordem com que as equações são examinadas [59] [60] [61] [62] [63] nos apresentam métodos eficazes de efetuar a paralelização o método de Gauss-Seidel, aplicado à *stencils* de diferenças finitas, denominado *RED/BLACK Gauss-Seidel* [64] e suas variantes multi-cores.

Nestas metodologias efetuamos uma decomposição do domínio e em cada subdomínio aplicamos a decomposição de cores. No caso do processamento em CPU adotamos o RED-BLACK Gauss-Seidel, que basicamente separa o produto de índices pares e ímpares, e pode ser observada na figura 3.13.

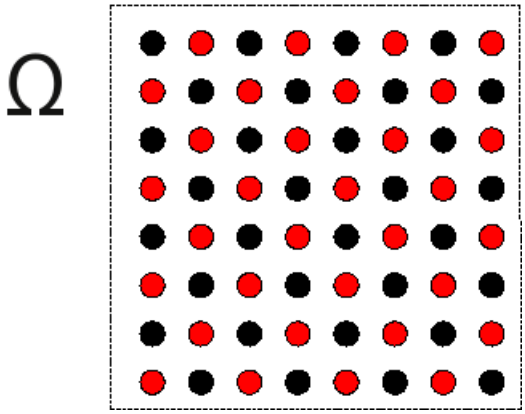


Figura 3.13: Partição do Domínio RED/BLACK

Estes procedimentos decompõe, ou particionam, o domínio em *cores* distintas, de forma à desacoplar as incógnitas. Deste modo dividimos a matriz em partes, cada uma ligada à um processo ou núcleo de processamento.

Tomando como P o número de processos concorrentes que desejamos implementar, efetuamos o balanço de carga entre cada processo dividindo o domínio Ω em P regiões, onde cada processo é responsável apenas pelo interior de cada região como vemos na figura 3.14.

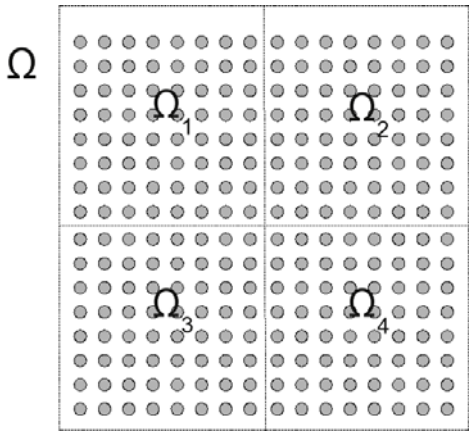


Figura 3.14: Partição do Domínio

Tomando como exemplo a figura 3.14 temos um total de quatro processos concorrentes que particionam o domínio Ω em quatro partes iguais Ω_1, Ω_2, Ω_3, Ω_4.

Cada processo independentemente, que de agora em diante denominamos por *thread*, manipula o interior de cada sub-domínio em paralelo e após o término de

todos os *threads* um função puramente seqüencial realiza a troca de informações entre as bordas dos sub-domínios que denominaremos *halo* ou *ghost*, segundo [62], a região do *halo* pode ser observada na figura 3.15.

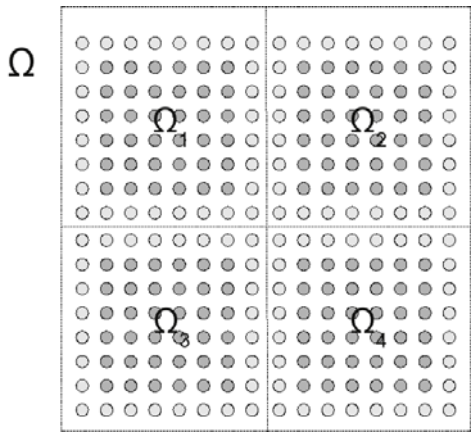


Figura 3.15: Partição do Domínio e Visualização do Halo

Apresentamos a seguir o algorítimo de Gauss-Seidel RED-BLACK com implementação paralela. Note-se que o truque ao paralelizar o algoritmo de Gauss-Seidel consiste em usar um esquema de enumeração que desacopla as variáveis superiores da diagonal das inferiores.

Tomemos $u^0 \in \mathbb{R}^{NX \times NY}$

P {Número de *threads*}

N {Número de interações}

NX_p {Número de pontos do grid na ordenada do subdomínio Ωp}

NY_p {Número de pontos do grid na abcissa do subdomínio Ωp}

for $p = 1$ to P **do**

  $thread_p \leftarrow newthread()$ {Cria um novo *thread*}

  for $k = 1$ to N **do**

    cor = RED {Pontos Vermelhos}

    for $i = 1$ to NX_p **do**

      for $j = 1$ to NY_p **do**

        $u_{i,j}^{(k+1)} = \frac{1}{4}[u_{i-1,j}^{(k)} + u_{i+1,j}^{(k)} + u_{i,j-1}^{(k)} + u_{i,j+1}^{(k)}]$

        $j \leftarrow j + 1$

        $e \leftarrow |u - u_{old}|$ {Avalia o erro}

      end for

      $i \leftarrow i + 1$

    end for

    cor = BLACK {Pontos Pretos}

    for $i = 1$ to NX_p **do**

      for $j = 1$ to NY_p **do**

        $u_{i,j}^{(k+1)} = \frac{1}{4}[u_{i-1,j}^{(k)} + u_{i+1,j}^{(k)} + u_{i,j-1}^{(k)} + u_{i,j+1}^{(k)}]$

        $j \leftarrow j + 1$

        $e \leftarrow |u - u_{old}|$ {Avalia o erro}

      end for

      $i \leftarrow i + 1$

    end for

    jointhreads() {Espera todos os *threads* terminarem}

  end for

  $k \leftarrow k + 1$

end for

Figura 3.16: O Algoritmo Paralelo

3.2.7 Implementação na CPU

A implementação na CPU se fez com 1, 2, 4, 8 e 16 *threads* com partições uni e bi-dimensionais uniformes, de acordo com a metodologia desenvolvida no capítulo 2 e nas figuras 2.3 e 2.4 podemos observar muito bem o problema que se apresenta;

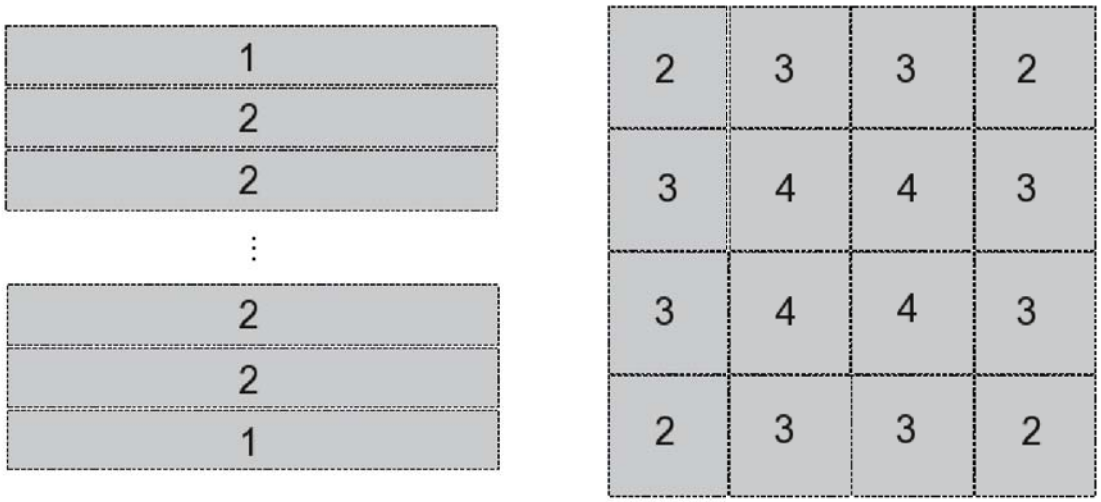


Figura 3.17: Comunicação Uni e Bi-Dimensional

Nesta implementação cada sub-domínio é manipulado por um *thread*, o código fonte foi implementado em C ANSI [65] e o objeto gerado no compilador GCC [66]. Criamos programas com 1, 2, 4, 8 e 16 *threads* e cada um avalia matrizes com 256×256, 512×512, 1024×1024, 2048×2048, 4096×4096 e 8192×8192 pontos Seu código pode ser observado nas figuras 3.18 e 3.19

```
void* interacao_th_gs(void *M)
{
    int i,j;
    double aux,temp = 0.0;
    struct Matrix_par *Mth;
    Mth = (struct Matrix_par *)M;
    aux = Mth->diff;
```

Figura 3.18: Código fonte da rotina iteração

```
    for (i = Mth->Mi; i < Mth->Mf; i++ ){
      for (j = Mth->Ni; j < Mth->Nf; j++){
        if(i>0 && i<NX-1 && j>0 && j<NY-1){
          if((i*j)%2 ==1){
// BLACK
          temp = Mth->u[i][j];
          Mth->u[i][j] = ( Mth->u[i-1][j]
                         + Mth->u[i+1][j]
                         + Mth->u[i][j-1]
                         + Mth->u[i][j+1] ) *.25;
// Calcula o erro
          temp -= Mth->u[i][j];
          if(temp<0.0)
            temp*=-1.0;
          if(temp<aux)
            aux = temp;
        }}}
//RED
    for (i = Mth->Mi; i < Mth->Mf; i++ ){
      for (j = Mth->Ni; j < Mth->Nf; j++ ){
        if(i>0 && i<NX-1 && j>0 && j<NY-1){
          if((i*j)%2 ==0){ // RED
            temp = Mth->u[i][j];
            Mth->u[i][j] = ( Mth->u[i-1][j]
                           + Mth->u[i+1][j]
                           + Mth->u[i][j-1]
                           + Mth->u[i][j+1] ) *.25;
// Calcula o erro

            temp -= Mth->u[i][j];
            if(temp<0.0)
              temp*=-1.0;
            if(temp<aux)
              aux = temp;
        }}}
    barrier_th(); // espera os threads terminarem
    Mth->diff=aux; // retorna o erro
    pthread_exit((void *) 0);
}
```

Figura 3.19: Código fonte da rotina de iteração, continuação

3.2.8 Implementação na GPU

Neste caso, dadas as características particulares da GPU em relação ao acesso à memória foi implementado o algorítimo de Gauss-Seidel com 5 cores [67] [59]. Na figura 3.20 podemos observar o domínio e o desacoplamento das incógnitas

superiores e inferiores da matriz, que referenciam as interações do presente $(k+1)$ e do passado (k), respectivamente; com o *stencil* de 5 pontos no detalhe.

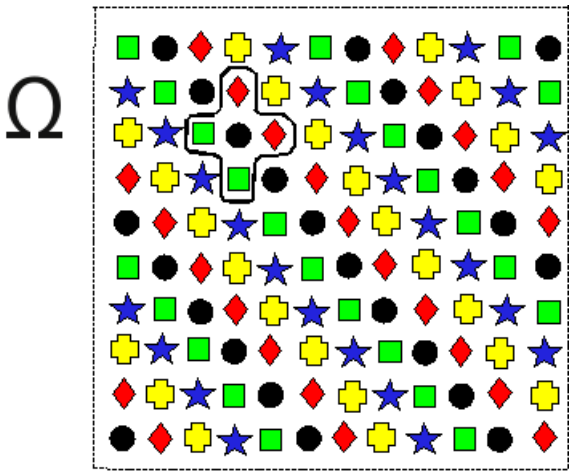


Figura 3.20: Partição do Domínio em 5 cores

A fim de avaliar como o tamanho do bloco de grids influencia a performance, criamos programas com tamanho de blocos de 8×8, 16×8, 16×16, 32×16, 32×32, 64×16 e 128×8 *threads* por bloco. Cada programa avalia matrizes com 256×256, 512×512, 1024×1024, 2048×2048, 4096×4096 e 8192×8192 pontos.

Apresentamos nas figuras 3.21 e 3.22 um trecho de código com a implementação da rotina de Gauss-Seidel que executará em cada *thread* da GPU, tais funções denominamos *kernels*.

```
__global__ void GPU_laplace2d(int N, int M, float *u)
{    __shared__  float us[BLOCK_X+2][BLOCK_Y+2];
     __shared__  float usx0[BLOCK_X+2][BLOCK_Y+2];
     __shared__  float usx1[BLOCK_X+2][BLOCK_Y+2];
     __shared__  float usy0[BLOCK_X+2][BLOCK_Y+2];
     __shared__  float usy1[BLOCK_X+2][BLOCK_Y+2];
     int tx = threadIdx.x;
     int ty = threadIdx.y;
     int bx = blockIdx.x*(BLOCK_X);
     int by = blockIdx.y*(BLOCK_Y);
     int x = tx + bx;
     int y = ty + by;
```

Figura 3.21: Código fonte da rotina *kernel*

```
    int ind = INDEX(x,y,N);
    if(x>0 && y>0 && x<N-1 && y<M-1){
       us[tx][ty] = u[ind];
       usx0[tx][ty] = u[ind-1];
       usx1[tx][ty] = u[ind+1];
       usy0[tx][ty] = u[INDEX(x,y-1,N)];
       usy1[tx][ty] = u[INDEX(x,y+1,N)];
       }
    __syncthreads();
    if(x>0 && y>0 && x<N-1 && y<M-1)
       u[ind] = .25f*(usx0[tx][ty]
                     +usx1[tx][ty]
                     +usy0[tx][ty]
                     +usy1[tx][ty]);
    __syncthreads();
}
```

Figura 3.22: Código fonte da rotina *kernel*, continuação

Note que cada região de memoria compartilhada, as variáveis definidas na região de memória compartilhada, o são com valores de bloco que tornam o acesso à memória coalescente com a memoria global. O acesso, de leitura ou escrita, à memória global tem melhor desempenho somente quando esta é acessada de forma coalescente dentro de um *half-warp*; neste caso o *hardware* pode então, buscar ou armazenar, na memória global com o menor número de operações.

Observe que a rotina em GPU não possui os *loops* que encontramos nas rotinas para a CPU, isto se deve ao fato de que estes loops são implementados no momento em que definimos os *grids* de blocos de *threads*, tal como no código apresentado na figura 3.23.

```
#define NX      512                 // Tamanho da matriz em linhas
#define NY      512                 // Tamanho da matriz em colunas
#define BLOCK_X 16                  // Tamanho do Bloco em linhas
#define BLOCK_Y 16                  // Tamanho do Bloco em colunas
#define INTERACTIONS 100            // Numero de Interacoes
int blockx = 1 + (NX-1)/BLOCK_X;    // Calcula Blocos
int blocky = 1 + (NY-1)/BLOCK_Y;
dim3 dimBlock(BLOCK_X,BLOCK_Y);     // Ajusta o numero de blocos
dim3 dimGrid(blockx,blocky);        // Ajusta a dimensao do Grid
     for (int i = 1; i <= INTERACTIONS; ++i) {
         GPU_laplace2d<<<dimGrid, dimBlock>>>(NX, NY,u);
         CUDA_SAFE_CALL( cudaThreadSynchronize() );
         CUT_CHECK_ERROR("GPU_laplace2d falhou\n");
     }
```

Figura 3.23: Código fonte com a partição do problema na GPU

Capítulo 4

Experimentos e Resultados

A plataforma utilizada para o teste é composta por uma *workstation* HP vx8600 com dois processadores INTEL XEON 5440, *clock* de 2.5Ghz, e 8 Gbytes de memória RAM. Cada processador possui quatro núcleos de processamento e 12 Mbytes de memória cache; e uma placa gráfica GTX-280 da NVIDIA com 1Gbyte de memória.

Observamos que na área nuclear, com algumas excessões, os pesquisadores utilizam códigos sequenciais; contruídos em um momento em que não havia a preocupação com modelos de programação paralela pois esta não estava ainda disponível. De forma que devemos focar a avaliação dos resultados obtidos pela CPU em relação a GPU, no caso em que o código da CPU é sequencial.

4.1 O Transporte de Nêutrons

Nosso experimento consiste em medir o tempo de execução de programas criados para o cálculo simplificado de uma blindagem de nêutrons conforme a explanação do capítulo 3 seção, 3.1. Criamos estes programas tanto para execução na CPU quanto na GPU e cada um efetua simulações variando o número de nêutrons nas seguintes faixas: 2^{20}, 2^{21}, 2^{22}, 2^{23}, 2^{24}, 2^{25}, 2^{26}, 2^{27} e 2^{28} partículas.

Os programas sob análise diferem entre si no paralelismo que implementam, na CPU foram criados 4 programas nos quais estudamos os casos seqüencial, com 2, 4 e 8 *threads*, este último emprega ao máximo o potencial de paralelismo da estação de trabalho que possui 8 núcleos de processamento. No caso na GPU foram criados 6 programas nos quais analisamos a variação do tamanho do bloco de *threads*,

utilizamos blocos com 16×1, 32×1, 64×1, 128×1, 256×32 e 256×64 *threads*.

Vamos considerar no problema sob análise um SLAB homogêneo de alumínio, com seção de choque de absorção $\Sigma_a = 0.015$, seção de choque de espalhamento $\Sigma_s = 0.84$ e espessura de 10cm, como a possível blindagem. Cada um destes programas foi executado 10 vezes com variação da semente do gerador randômico e das medidas extraímos o tempo médio de execução.

O resultado analítico é dado por,

$$I(x) = I_o e^{-\Sigma_a x} \quad = 0.8607 I_o \tag{4.1}$$

Resultados na CPU

Os conjuntos de dados formados pela média das medidas dos tempos de simulação em relação ao número de *threads* e ao número de nêutrons, são apresentados na tabela 4.1 em segundos.

Tabela 4.1: Velocidade da CPU em segundos

	Número de Histórias de Nêutrons								
Threads	2^{20}	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}	2^{26}	2^{27}	2^{28}
1	0,88	1,74	3,51	7	14,29	28,72	57,54	114,77	228,46
2	0,42	0,86	1,67	3,38	6,69	13,4	26,85	53,89	107,57
4	0,21	0,44	0,86	1,74	3,43	6,89	13,67	27,76	54,59
8	0,11	0,21	0,41	0,82	1,64	3,28	6,55	13,1	26,2

Na figura 4.1 observamos um gráfico do desempenho da CPU gerado em função dos dados da tabela 4.1. O problema se comporta de maneira linear em relação ao número de processos concorrentes pois é do tipo *absurdamente paralelizavel*, ou seja, as componentes paralelizáveis do problema não sofrem influências de qualquer parte sequencial.

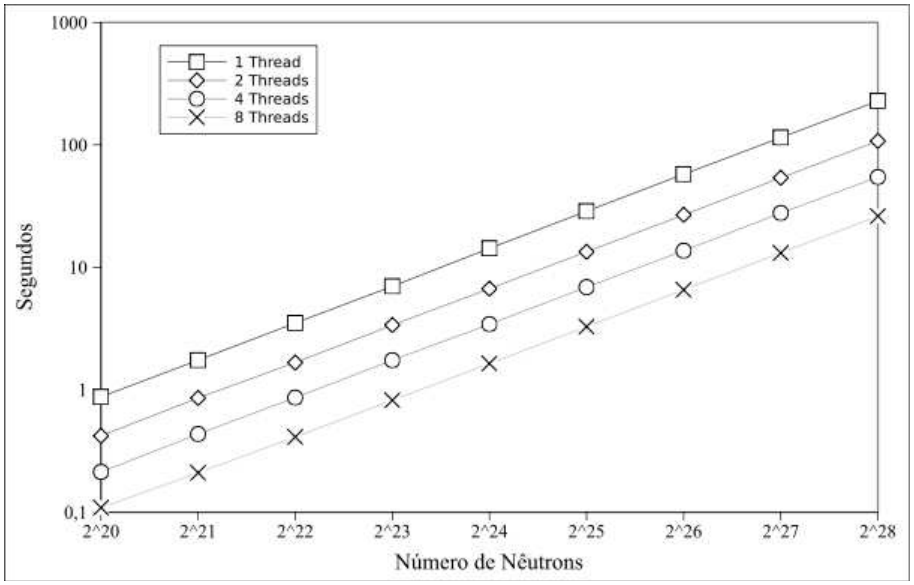


Figura 4.1: Tempo de execução na CPU

Resultados na GPU

A tabela 4.2 nos apresenta o tempo de resolução do problema em milissegundos da GPU em relação ao tamanho dos blocos de *threads* e o número de histórias de nêutrons. A escala de medição foi alterada devido a uma diferença de quase 2 ordens de grandeza no desempenho em pró da GPU. Podemos observar na 4.2 e na figura 4.1 que a performance do problema apresenta uma linearidade em relação ao número de *threads*, aumentando conforme este cresce;

Tabela 4.2: Tempo de execução na GPU em milissegundos

	Número de Histórias de Nêutrons								
Bloco	2^{20}	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}	2^{26}	2^{27}	2^{28}
16x1	34,98	69,51	137,93	277,12	550,17	1097,74	2205,08	4401,33	8800,52
32x1	18,9	38,66	76,22	150,26	306,16	601,71	1199,67	2410,56	4811,26
64x1	14,73	30,37	59,68	121,06	234,77	473,95	943,66	1880,27	3779,67
128x1	13,98	29,78	59,44	117,79	231,74	468,6	931,74	1856,22	3730,81
256x32	15,69	32,43	63,21	125,42	250,88	499,64	996,14	1996,69	3990,04
256x64	13,14	27,97	55,81	108,91	215,79	432,91	863,26	1721,86	3448,96

Análise dos Resultados

A tabela 4.3 apresenta o resultado médio da simulação do problema e seu desvio padrão na CPU e na GPU.

Tabela 4.3: Resultado médio e desvio padrão

Dispositivo	CPU	GPU					
Threads	1 à 8	16x1	32x1	64x1	128x1	256x32	256x64
Valor Médio	0,8660	0,8638	0,8592	0,8589	0,8616	0,8577	0,8661
Desvio Padrão	0,0072	0,0111	0,0251	0,0264	0,0185	0,0242	0,0036
Módulo Erro	0,0053	0,0031	0,0015	0,0018	0,009	0,003	0,0054

A figura 4.2 dispõe o gráfico da relação entre a velocidade de simulação média da GPU em relação à CPU, com apenas 1 *thread*, em escala logarítmica.

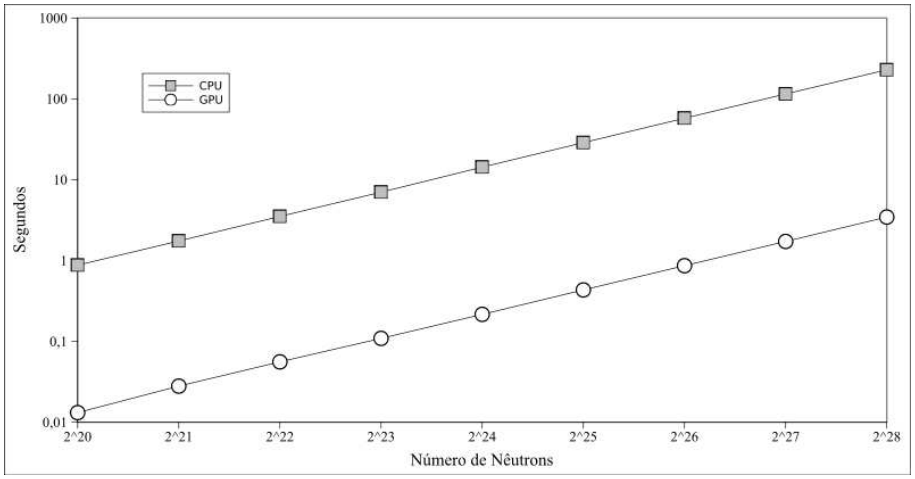


Figura 4.2: Tempo Absoluto GPU e CPU

Analizando as tabelas 4.1 e 4.2 observamos que a GPU tem um ganho de desempenho maior que 60 vezes no caso do código ser puramente seqüencial. Devemos salientar que o código executado na GPU pode ser alterado para utilização de múltiplas placas e mesmo *clusters* de placas gráficas comunicando-se entre os nós por MPI.

Atualmente alguns projetos de computação de alto desempenho [68] e a própria NVIDIA [69], dispõe de implementações destes *clusters* heterogêneos. Considerando

o preço relativamente baixo da GPU empregada, em torno de US$400,00, em relação à estação de trabalho, cujo preço gira em torno de US$4000,00, o ganho de performance conseguido à transforma e uma excelente opção para este tipo de aplicação.

4.2 Transferência de Calor

Nosso experimento consiste em medir o tempo de execução de programas criados para a resolução da equação do calor bidimensional e estacionária, pelo método interativo de Gauss-Seidel conforme a explanação do capítulo 3, seção 3.2. Criamos estes programas tanto para execução na CPU quanto na GPU e cada um efetua simulações variando o tamanho da matriz nos seguintes valores : 256×256, 512×512, 1024×1024, 2048×2048, 4096×4096, 8192×8192 e 16384×16384 pontos. O teste na GPU foi limitado superiormente por 8192×8192 em função da quantidade de memória utilizada e do algoritmo empregado no programa. Cada um destes programas foi executado 5 vezes e guardamos o valor médio.

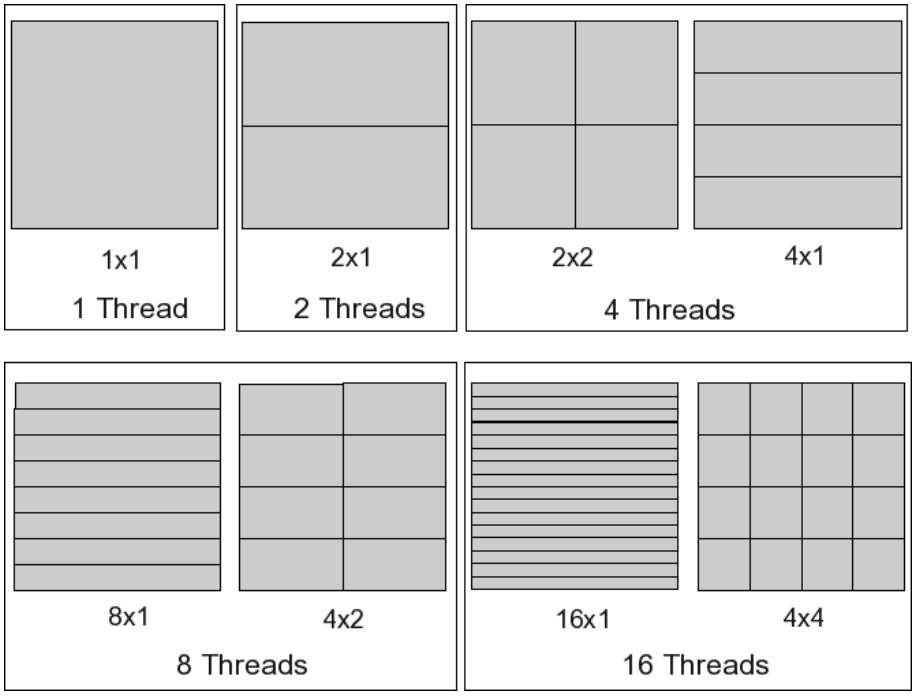


Figura 4.3: Distribuição do Problema de Poisson em *Threads*

Os programas sob análise diferem entre si no paralelismo que implementam, como podemos ver na figura 4.3. Na CPU foram criados 8 programas nos quais

estudamos os casos seqüencial, com $2 \times 1 = 2$, $2 \times 2 = 4$, $4 \times 1 = 4$, $8 \times 1 = 8$, $4 \times 2 = 8$, $16 \times 1 = 16$ e $4 \times 4 = 16$ *threads*.

Os programas de 8×1, 4×2 empregam ao máximo o potencial de paralelismo da estação de trabalho que possui 8 núcleos de processamento. Os de 16×1 e 4×4 foram incluídos para avaliar o desempenho da comunicação entre os *threads*. Na CPU foram realizadas as medidas do tempo de simulação para 1000 interações.

Na GPU foram criados 6 programas que diferem entre si na definição do tamanho do bloco de *threads*, com blocos de 8×8, 16×8, 16×16, 32×16, 64×8 e 128×4 *threads* por bloco. Na GPU foi medido o tempo decorrido para a solução do problema para 10000 interações nos casos de 256×256 até 2048×2048 e 1000 interações nos outros casos, a escala de medição foi alterada nas matrizes menores devido a uma diferença de quase 2 ordens de grandeza no desempenho em pró da GPU.

Resultados na CPU

Na tabela 4.4, a primeira coluna nos informa o tamanho da matriz quadrada utilizada, nas seguintes temos o tempo de simulação em função do número de *threads* utilizados para particionar o problema. No total efetuamos 1000 interações do método.

Tabela 4.4: Tempo da CPU em Segundos

Matriz	1x1	2x1	4x1	8x1	16x1	2x2	4x2	4x4
16384	6673,13	3348,72	1670,25	822,99	884,75	1677,73	865,3	913,23
8192	1661,08	831,94	419,65	220,07	253,35	419,31	220,65	256,52
4096	420,1	212,71	109,88	62,97	91,13	106,63	62,05	97,31
2048	109,47	56,57	33,61	23,75	37,39	32,21	22,96	37,21
1024	32,4	17,89	14,59	11,93	11,93	13,32	11,79	11,79
512	9,48	4,723	3,294	3,232	3,583	3,339	3,355	3,822
256	2,418	1,255	1,005	1,138	1,469	1,025	1,202	1,576

Na figura 4.4 observamos o ganho de desempenho entre o número de *threads*, como eles são distribuídos e o tamanho da matriz em relação ao modo sequencial..

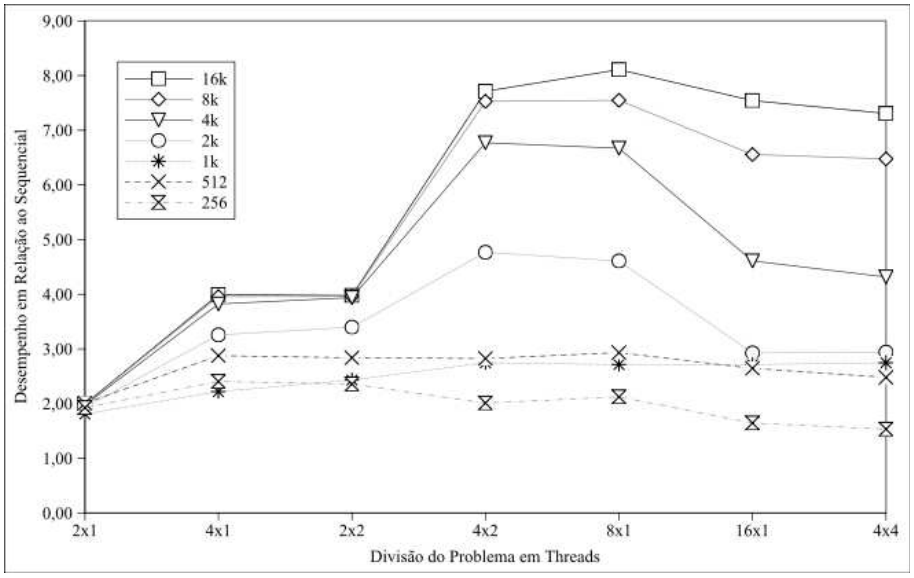


Figura 4.4: Velocidade Relativa da CPU

Analisando a tabela 4.4 e a figura 4.4 vemos que as matrizes de maior dimensão, principalmente a de 8192×8192 e 16384×16384, tem desempenho relativo muito maior do que as matrizes de dimensões menores até o caso em que utilizamos 8 *threads*, ou seja, 1 processo por núcleo. Nos casos de 16 *threads* temos um decréscimo de desempenho devido à comunicação, exatamente como prevê a lei de Amdahl [28], pois está é estritamente sequencial. Observamos também que nas matrizes de dimensão menor, por exemplo 256×256, o procedimento puramente seqüencial da comunicação entre os *halos* tem grande influência .

Resultados na GPU

A tabela 4.5 nos mostra os resultados das simulações na GPU em segundos. A primeira coluna nos informa o tamanho da matriz quadrada utilizada e as seguintes o tempo de simulação em função do número de *threads* para cada bloco de grids utilizados para particionar o problema.

Tabela 4.5: Tempo de simulação da GPU em segundos

Tamanho do Bloco	**Tamanho da Matriz**					
	10000 Interações				1000 Interações	
	256	512	1024	2048	4096	8192
8 x 8	0,372	1,097	4,039	15,090	6,459	32,270
16 x 8	0,348	0,917	3,180	12,160	4,810	20,820
16 x 16	0,373	1,030	3,532	13,820	5,590	24,570
32 x 16	0,431	1,158	4,172	15,820	6,185	25,68
64 x 8	0,420	1,156	4,136	16,180	6,210	23,73
128 x 4	0,417	1,206	4,096	15,330	6,080	24,112

Análise dos Resultados

A medição do erro absoluto em relação ao número de iterações na CPU e na GPU pode ser observada na tabela 4.6.

Tabela 4.6: Erro Absoluto na CPU e GPU

Interações	Erro Absoluto CPU	Erro Absoluto GPU
10	1,30E-001	1,30E-001
100	1,40E-003	1,47E-003
1000	1,90E-006	1,50E-005
10000	1,00E-007	2,00E-006

Como podemos observar na figura 4.5 a GPU tem um desempenho maior do que 10 vezes que a CPU sob teste com 8 *threads* e cerca de 100 vezes mais rápida no caso do código ser puramente seqüencial. De maneira análoga ao caso do problema de transporte de nêutrons salientamos que o código executado na GPU pode ser alterado para utilização de múltiplas placas, e múltiplos computadores de um *cluster* heterogêneo.

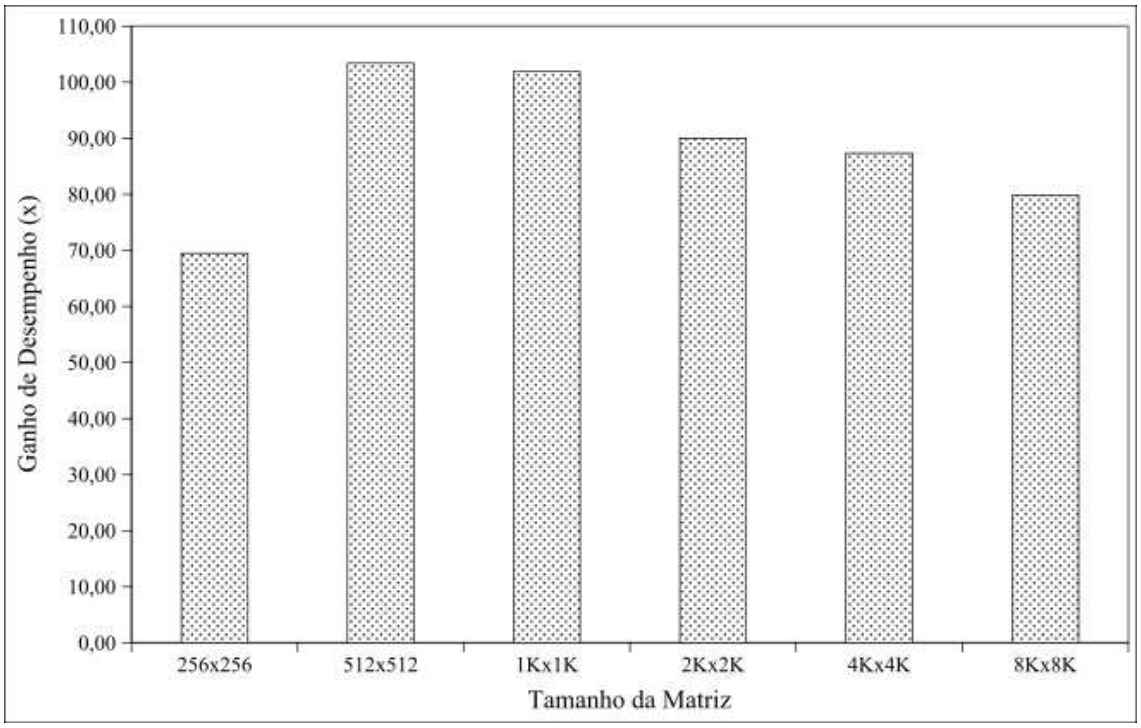
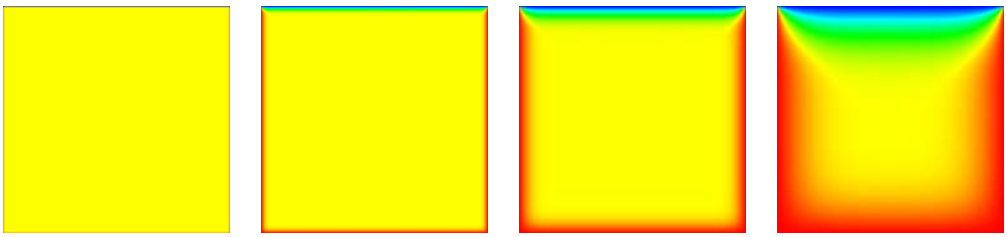


Figura 4.5: Ganho de Desempenho da GPU sobre a CPU

Podemos ver na figuras 4.6 o processo de convergência em função do número de interações. Note na imagem correspondente ao *Início*, o ajuste das condições iniciais do problema; nas bordas laterais e inferior temos uma temperatura de 100 ℃, representada pela cor vermelha, e na borda superior uma temperatura de 0 ℃, representada pela cor azul. No interior aplicamos uma temperatura de 75 ℃, representada em amarelo, como um bom primeiro valor para o processo interativo.



(a) Início (b) 100 (c) 1000 (d) 10000

Figura 4.6: Convergência em Função do Número de Interações

Capítulo 5

Conclusão e Perspectivas Futuras

Os resultados apresentados no capítulo 4 são encorajadores e indicam um rumo interessante à trilhar na seara da computação científica, e ainda assim poderiam ser otimizados por novos *layouts* de manipulação da memória, principalmente se utilizarmos a memória de texturas [70] e manipulações fora de ordem [71].

Uma análise mais acurada do resíduo, incutido pelo particionamento em blocos, do problema de Poisson pode ser feita no futuro.

É claro que sem esquecer o fato de os métodos númericos que foram analisados serem modelos simplificados de problemas que a comunidade científica, e em particular a área nuclear, necessitam resolver. Ao examinar a interminável lista de aplicações prováveis na análise numérica, na física de reatores, em física estatística e CFD, vemos que estas novas ferramentas computacionais sob a forma de computação heterogênea em *clusters* de GPUs representam um grande passo para o incremento da densidade e profundidade nos modelos atuais pelo salto de desempenho que proporcionam.

No passado, uma abordagem simplificadora do modelo matemático de um determinado problema em engenharia de reatores, e em muitas outras áreas da ciência, sempre se fez premente em virtude da necessidade de diminuir a escala e a dimensão do problema. A ampliação da capacidade computacional e redução de custos associados a esta tecnologia são fatores importantes que podem impulsionar os grupos de pesquisa da área nuclear em um salto qualitativo e quantitavo em seus trabalhos.

A implementação dos problemas abordados neste trabalho mostra que os *clusters* heterogêneos constituidos por CPUs e GPUs em uma arquitetura similar a

TESLA [69] constituem um ferramenta valiosa para a comunidade científica. Podemos salientar que o tempo de aprendizado da linguagem CUDA não é maior do que qualquer outra linguagem estruturada, a menos do entendimento e compreensão da implementação dos algoritmos paralelos.

O principal gargalo desta tecnologia consiste na dificuldade de integração com os programa antigos, muitos dos quais escritos para trabalhar em mono-tarefa. A aplicação de tecnologias de paralelismo em códigos como HAMMER [72] otimizando os resultados por enxame de partículas (PSO) [73] já seria um grande avanço.

Uma opção interessante seria fazer uma analise do perfil de execução do código [74] [75] e assim substituir as operações de maior custo computacional por rotinas implementadas preferencialmente em OpenCL [39] que executariam na GPU e em CPU *multi-cores*. Efetuando o controle da quantidade de processos por mecanismos de *spawning* e a troca de informações entre os processos através da *shared memory* do sistema operacional.

Perspectivas Futuras

Vamos dividir nossas perspectivas em 3 linhas de pesquisa:

- Infraestrutura Básica para GPU
  - Programação em OpenCL.
  - Implementação do método de acurácia mista [76], [77].
  - Implementacao de métodos em GPU para tratamento de matrizes esparsas baseado em [78].
  - Implementação do método de Gauss-Seidel em 2D e 3D transiente, com opção de reolver o sistema linear por blocos, linhas e colunas.
  - Implementação do método do gradiente conjugado e suas variantes para resolver o caso Ax=b, onde a matriz é positiva definida [79].
  - Implementação do método *Generalized Minimal Residual* GMRES [80].
  - Implementação do método *Meshless Local Petrov-Galerkin* MLPG [81].
- Infraestrutura de Comunicação

  - Implementação de duas ou mais GPUs em um computador.
  - Implementação de um cluster heterogêneo com troca de menssagens por MPI [6].

- Demandas Específicas
  - Implementação de redes neurais na GPU (BackProp [82], GRNN [83] , Kohonen [84]).
  - Implementação de uma biblioteca de algoritmos genéticos na GPU [85] [86].
  - Implementação de otimização por enxame de partículas (PSO) [87] na GPU.

Bibliografia

[1] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, et al. Larrabee: a many-core x86 architecture for visual computing. 2008.

[2] A.M. Devices. AMD Fusion Whitepaper.

[3] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the cell processor for scientific computing. In *Proceedings of the 3rd conference on Computing frontiers*, pages 9–20. ACM New York, NY, USA, 2006.

[4] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. 2008.

[5] John von Neumann. First draft of a report on the edvac. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

[6] Jeffrey M. Squyres. Definitions and fundamentals – the message passing interface (MPI). *ClusterWorld Magazine, MPI Mechanic Column*, 1(1):26–29, December 2003.

[7] V. S. Sunderam. Pvm: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2:315–339, 1990.

[8] Cristiana Amza, Alan L. Cox, Hya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29:18–28, 1996.

[9] Axel Keller, Matthias Brune, and Er Reinefeld. Resource management for high-performance pc clusters. In *Lecture Notes in Computer Science*, pages 25–34, 1593.

[10] T. Sterling, D.J. Becker, D. Savarese, J.E. Dorband, U.A. Ranawake, and C.V. Packer. BEOWULF: A parallel workstation for scientific computation. In *In Proceedings of the 24th International Conference on Parallel Processing*, 1995.

[11] Honghui Lu, Y. Charlie Hu, and Willy Zwaenepoel. Openmp on networks of workstations, 1998.

[12] Frank Mueller. Implementing posix threads under unix: Description of work in progress. In *In Proceedings of the Second Software Engineering Research Forum*, pages 253–261, 1992.

[13] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. The gpu as numerical simulation engine, 2003.

[14] Rob H. Bisseling. Basic techniques for numerical linear algebra on bulk synchronous parallel computers. In *Workshop Numerical Analysis and its Applications 1996, volume 1196 of Lecture Notes in Computer Science*, pages 46–57. Springer-Verlag, 1997.

[15] Uzi Vishkin. A case for the pram as a standard programmer's model, 1992.

[16] Uzi Vishkin, Shlomit Dascal, Efraim Berkovich, and Joseph Nuzman. Explicit multi-threading (xmt) bridging models for instruction parallelism (extended abstract). In *Proc. 10th ACM Symposium on Parallel Algorithms and Architectures (SPAA*, pages 140–151. ACM Press, 1998.

[17] M.J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9):948–960, 1972.

[18] Andrew Corrigan, Fernando Camelli, Rainald Löhner, and John Wallin. Running unstructured grid cfd solvers on modern graphics hardware. In *19th AIAA Computational Fluid Dynamics Conference*, number AIAA 2009-4001, June 2009.

[19] M. Januszewski and M. Kostur. Accelerating numerical solution of Stochastic Differential Equations with CUDA. *Arxiv preprint arXiv:0903.3852*, 2009.

[20] C. Share. Evaluating the Use of GPGPUs in Life Science and Supercomputing Applications.

[21] P.B. Noel, A.M. Walczak, K.R. Hoffmann, J. Xu, J.J. Corso, and S. Schafer. Clinical Evaluation of GPU-Based Cone Beam Computed Tomography. In *MICCAI Workshop: High–Performance Medical Image Computing and Computer Aided Intervention*, 2008.

[22] V. Kumar. *Introduction to parallel computing*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2002.

[23] D.R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1997.

[24] T.L. Sterling. *Beowulf cluster computing with Linux*. MIT Press, 2002.

[25] J.M. Bull and C. Johnson. Data distribution, migration and replication on a cc-NUMA architecture. In *Proceedings of the Fourth European Workshop on OpenMP*, 2002.

[26] T. Shanley. *InfiniBand network architecture*. Addison-Wesley Professional, 2003.

[27] R. Shrout. Intel 80 core Terascale Chip Explored-4GHz clocks and more. *PC Perspective*, 2007.

[28] G.M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM New York, NY, USA, 1967.

[29] D.R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1997.

[30] M. Barabanov and V. Yodaiken. Real-time linux. *Linux journal*, 23, 1996.

[31] C. Lejdfors and L. Ohlsson. Implementing an embedded GPU language by combining translation and generation. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 1610–1614. ACM New York, NY, USA, 2006.

[32] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *International Conference on Computer Graphics and Interactive Techniques*, pages 777–786. ACM New York, NY, USA, 2004.

[33] G. Cummins, R. Adams, and T. Newell. Scientific computation through a GPU. *IEEE Southeastcon, 2008*, pages 244–246, 2008.

[34] A. Frezzotti, G.P. Ghiroldi, and L. Gibelli. Solving Kinetic Equations on GPUs I: Model Kinetic Equations. *Arxiv preprint arXiv:0903.4044*, 2009.

[35] O. Developers. The Open64 web site.

[36] J.M. Danskin, J.S. Montrym, J.E. Lindholm, S.E. Molnar, and M. French. Parallel Array Architecture for a Graphics Processor, December 15 2006. US Patent App. 11/611,745.

[37] C.S. Guiang, K.F. Milfeld, A. Purkayastha, and J. Boisseau. Memory performance of dual-processor nodes: comparison of Intel Xeon and AMD Opteron memory subsystem architectures. In *4th LCI International Conference on Linux Clusters: The HPC Revolution*, 2003.

[38] C. NVIDIA. Programming Guide 2.0, 2008.

[39] A. Munshi. Opencl: Parallel computing on the gpu and cpu. *SIGGRAPH 08: ACM SIGGRAPH 2008 classes*, 2008.

[40] G. Hiebert. Openal 1.1 specification and reference.

[41] ARB OpenGL. OpenGL Specification. *Draft Version*, 1, 1997.

[42] D. Hough. Applications of the Proposed IEEE 754 Standard for Floating-Point Arithetic. *Computer*, 14(3):70–74, 1981.

[43] C. Cercignani. *The Boltzmann equation and its applications*. Springer, 1988.

[44] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953.

[45] K. Binder. Applications of monte carlo methods to statistical physics. *Reports on Progress in Physics*, 60(5):487–559, 1997.

[46] L. L. Carter and E. D. Cashwell. *Particle Transport Simulation with the Monte Carlo Method; Prepared for the Division of Military Application, U.S. Energy Research and Development Administration*. U. S. Department of Energy, 1975.

[47] S.A. Dupree and SK Fraley. *A Monte Carlo primer: A Practical approach to radiation transport*. Kluwer Academic/Plenum Publishers, 2004.

[48] H.W. Lilliefors. On the Kolmogorov-Smirnov test for normality with mean and variance unknown. *Journal of the American Statistical Association*, pages 399–402, 1967.

[49] C.P. Williams and S.H. Clearwater. *Explorations in quantum computing*. Springer-Verlag TELOS Santa Clara, CA, USA, 1997.

[50] C.A.B. Dantas. *Probabilidade: um curso introdutório*. Edusp, 1997.

[51] M.H. Kalos and P.A. Whitlock. *Monte carlo methods*. Wiley-VCH, 2008.

[52] C. Wagner. On the numerical evaluation of Fredholm integral equations with the aid of the Liouville-Neumann series. *J. Math. Phys*, 30:232–234, 1952.

[53] S.F. McCormick. *Multigrid methods*. Society for Industrial Mathematics, 1987.

[54] T. Meis, U. Marcowitz, and P.R. Wadsack. Numerical solution of partial differential equations. 1981.

[55] R. Bagnara. A unified proof for the convergence of Jacobi and Gauss-Seidel methods. *SIAM Review*, 37(1):93–97, 1995.

[56] A. Greenbaum. *Iterative methods for solving linear systems*. Society for Industrial Mathematics, 1997.

[57] M.R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J. Res. Nat. Bur. Stand*, 49(6):409–436, 1952.

[58] WM Kahan. *Gauss-Seidel methods of solving large systems of linear equations*. Toronto, 1958.

[59] L. Adams and J. Ortega. A multi-color SOR method for parallel computation. In *Proceedings of the International Conference on Parallel Processing*, pages 53–58. IEEE Computer Society, 1982.

[60] L.M. Adams and H.F. Jordan. Is SOR color-blind? *SIAM Journal on Scientific and Statistical Computing*, 7:490, 1986.

[61] J.M. Ortega and R.G. Voigt. *Solution of partial differential equations on vector and parallel computers*. Society for Industrial Mathematics, 1985.

[62] J.E. Jones and S.F. McCormick. Parallel multigrid methods. *ICASE LARC INTERDISCIPLINARY SERIES IN SCIENCE AND ENGINEERING*, 4:203–224, 1997.

[63] RD Falgout and JE Jones. Multigrid on massively parallel architectures. *1999.*, 1999.

[64] I. Yavneh. On red-black SOR smoothing in multigrid. *SIAM Journal on Scientific Computing*, 17(1):180–192, 1996.

[65] B.W. Kernighan and D.M. Ritchie. *C: a linguagem de programação padrão ANSI*. Campus, 1989.

[66] R. Stallman et al. *Using and porting the GNU compiler collection*. Free Software Foundation, 1999.

[67] L.M. Adams and H.F. Jordan. Is SOR color-blind? *SIAM Journal on Scientific and Statistical Computing*, 7:490, 1986.

[68] M. Showerman, J. Enos, A. Pant, V. Kindratenko, C. Steffen, R. Pennington, and W. Hwu. QP: A Heterogeneous Multi-Accelerator Cluster. In *Proceedings of the 10th LCI International Conference on High-Performance Clustered Computing (Boulder, Colorado, March 10–12, 2009)*, 2009.

[69] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, pages 39–55, 2008.

[70] S. Ryoo, C.I. Rodrigues, S.S. Baghsorkhi, S.S. Stone, D.B. Kirk, and W.H. Wen-mei. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82. ACM New York, NY, USA, 2008.

[71] J. Mellor-Crummey, D. Whalley, and K. Kennedy. Improving memory hierarchy performance for irregular applications using data and computation reorderings. *International Journal of Parallel Programming*, 29(3):217–247, 2001.

[72] JE Suich and HC Honeck. The HAMMER System. *Heterogeneous Analysis by Multigroup Methods of Exponentials and Reactors, Rep. DP-1064, duPont Savannah River National Laboratory*, 1967.

[73] M. Waintraub, R. Schirru, and C.M.N.A. Pereira. Multiprocessor modeling of parallel Particle Swarm Optimization applied to nuclear engineering problems. *Progress in Nuclear Energy*, 2009.

[74] Doug Burger and Todd M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, 1997.

[75] Monica S. Lam and Robert P. Wilson. Limits of control flow on parallelism. *SIGARCH Comput. Archit. News*, 20(2):46–57, 1992.

[76] Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Piotr Luszczek, and Stanimir Tomov. Using mixed precision for sparse matrix computations to

enhance the performance while achieving 64-bit accuracy. *ACM Trans. Math. Softw.*, 34(4):1–22, 2008.

[77] J. Kurzak and J. Dongarra. Implementation of the mixed-precision high performance LINPACK benchmark on the CELL processor. *LAPACK Working Note*, 177:06–580, 2006.

[78] CC Paige and MA Saunders. Solution of sparse indefinite systems of linear equations. *SIAM Journal on Numerical Analysis*, pages 617–629, 1975.

[79] P. Wesseling, INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE, and ENGINEERING HAMPTON VA. *An introduction to multigrid methods*. Wiley Chichester, 1992.

[80] R. Barrett, M. Berry, TF Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods SIAM. *Philadelphia, PA*, 1994.

[81] SN Atluri and T. Zhu. A new meshless local Petrov-Galerkin (MLPG) approach in computational mechanics. *Computational Mechanics*, 22(2):117–127, 1998.

[82] F.J. Pineda. Generalization of back-propagation to recurrent neural networks. *Physical Review Letters*, 59(19):2229–2232, 1987.

[83] DF Specht. A general regression neural network. *IEEE Transactions on Neural Networks*, 2(6):568–576, 1991.

[84] T. Kohonen. Self-organization and associative memory. 1989.

[85] C.M. Fonseca, P.J. Fleming, et al. Genetic algorithms for multiobjective optimization: Formulation, discussion and generalization. In *Proceedings of the fifth international conference on genetic algorithms*, volume 423. Citeseer, 1993.

[86] M. Mitchell. *An introduction to genetic algorithms*. The MIT press, 1998.

[87] J. Kennedy and R. Eberhart. Particle swarm optimization. In *IEEE International Conference on Neural Networks, 1995. Proceedings.*, volume 4, 1995.