Alessandra Matos Campos

Implementações sequencial e paralela de um novo algoritmo para a simulação de elementos e compostos magnéticos

Dissertação apresentada ao Programa de Pós-graduação em Modelagem Computacional, da Universidade Federal de Juiz de Fora como requisito parcial à obtenção do grau de Mestre em Modelagem Computacional.

Orientador: Prof. D.Sc. Marcelo Lobosco

Coorientador: Marcelo Bernardes Vieira

Coorientador: Sócrates de Oliveira Dantas

Juiz de Fora

2011

Campos, Alessandra Matos

Implementações sequencial e paralela de um novo algoritmo para a simulação de elementos e compostos magnéticos/Alessandra Matos Campos. – Juiz de Fora: UFJF/MMC, 2011.

XI, 72 p.: il.; 29, 7cm.

Orientador: Marcelo Lobosco

Coorientador: Marcelo Bernardes Vieira

Coorientador: Sócrates de Oliveira Dantas

Dissertação (mestrado) – UFJF/MMC/Programa de Modelagem Computacional, 2011.

Referências Bibliográficas: p. 69 – 72.

1. Física Computacional. 2. Modelo de Spins de Heisenberg. 3. Avaliação de Desempenho. 4. Computação de Alto Desempenho. I. Lobosco, Marcelo *et al.*. II. Universidade Federal de Juiz de Fora, MMC, Programa de Modelagem Computacional.

Alessandra Matos Campos

Implementações sequencial e paralela de um novo algoritmo para a simulação de elementos e compostos magnéticos

Dissertação apresentada ao Programa de Pós-graduação em Modelagem Computacional, da Universidade Federal de Juiz de Fora como requisito parcial à obtenção do grau de Mestre em Modelagem Computacional.

Aprovada em 25 de Fevereiro de 2011.

BANCA EXAMINADORA

Prof. D.Sc. Marcelo Lobosco - Orientador
Universidade Federal de Juiz de Fora

Prof. D.Sc. Marcelo Bernardes Vieira - Coorientador
Universidade Federal de Juiz de Fora

Prof. D.Sc. Sócrates de Oliveira Dantas - Coorientador
Universidade Federal de Juiz de Fora

Prof. D.Sc. Vitor Rafael Coluci
Universidade Estadual de Campinas

AGRADECIMENTOS

Aos meus pais por terem me apoiado no curso que escolhi.

A Celina e ao Geraldo pela amizade e pelo apoio em Juiz de Fora.

Aos amigos, em especial a Patrícia que esteve junto comigo em todos os momentos.

Aos orientadores, pela oportunidade de trabalhar neste projeto.

Ao João Paulo e ao Rafael por também terem aceitado encarar este desafio.

Ao Grupo de Computação Gráfica e ao Grupo de Física da Matéria Condensada.

A CAPES.

"Qualquer um que não se choque
com a Mecânica Quântica é
porque não a entendeu."
(Niels Bohr)

RESUMO

O fenômeno magnético é amplamente utilizado nos mais diversos dispositivos eletrônicos, de armazenamento de dados e de telecomunicações, dentre outros. O entendimento deste fenômeno é portanto de grande importância para dar suporte ao aperfeiçoamento e desenvolvimento de novas tecnologias. Uma das formas de melhorar a compreensão do fenômeno magnético é estudá-lo em escala atômica. Quando os átomos magnéticos se aproximam, interagem magneticamente, mesmo que submetidos a um campo magnético externo, e podem formar estruturas em escala nanométrica. Programas computacionais podem ser desenvolvidos com o objetivo de simular o comportamento de tais estruturas. Tais simuladores podem facilitar o estudo do magnetismo em escala nanométrica porque podem prover informações detalhadas sobre este fenômeno. Cientistas podem usar um simulador para criar e/ou modificar diferentes propriedades físicas de um sistema magnético; dados numéricos e visuais gerados pelo simulador podem ajudar na compreensão dos processos físicos associados com os fenômenos magnéticos. Entretanto, a execução de tais simulações é computacionalmente cara. A interação entre átomos ocorre de forma similar ao problema dos N corpos. Sua complexidade nos algoritmos tradicionais é O(N^2), onde N é o número de *spins*, ou átomos, sendo simulados no sistema. Neste trabalho propomos um novo algoritmo capaz de reduzir substancialmente este custo computacional, o que permite que uma grande quantidade de *spins* possa ser simulada. Adicionalmente ferramentas e ambientes de computação paralela são empregados para que os custos em termos de tempo de computação possam ser ainda mais reduzidos.

Palavras-chave: Física Computacional. Modelo de Spins de Heisenberg. Avaliação de Desempenho. Computação de Alto Desempenho.

ABSTRACT

The magnetic phenomena are widely used in many devices, such as electronic, data storage and telecommunications devices. The understanding of this phenomenon is therefore of great interest to support the improvement and development of new technologies. To better understand the magnetic phenomena, it is essential to study interactions at nano scale. When magnetic atoms are brought together they interact magnetically, even with an external magnetic field, and can form structures at nanoscale. Special design computer programs can be developed to simulate this interaction. Such simulators can facilitate the study of magnetism in nanometer scale because they can provide detailed information about this phenomenon. Scientists may use a simulator to create and/or modify different physical properties of a magnetic system; visual and numerical data generated by the simulator can help to understand the physical processes associated with the magnetic phenomenon. However, there is a natural high complexity in the numerical solution of physical models. The interaction between spins occurs in a similar way to the classical n-body problem. The complexity of this problem is $O(N^2)$, where N is the number of spins or atoms in the system. In this work we propose a new algorithm that can substantially reduce the computational cost, and allows the simulation of a large number of spins. Besides, tools and environments for high-performance computing are used so that the costs of computation time may be further reduced.

Keywords: Computational Physics. Heisenberg Spins Model. Performance Evaluation. High Performance Computing.

SUMÁRIO

1 INTRODUÇÃO 12
1.1 Motivação 12
1.2 Objetivo 13
1.3 Organização 13

2 MODELO FÍSICO-MATEMÁTICO 15
2.1 Introdução 15
2.2 Magnetismo nos Materiais 16
2.3 Momento de Spin Atômico 16
2.4 Momento Magnético Orbital do Elétron 17
2.5 Magnetização e Susceptibilidade Magnética 19
2.5.1 Paramagnetismo e Ferromagnetismo 20
2.6 Energia Potencial de Interação 21
2.7 Métodos Numéricos Aplicados à Simulação Computacional 23
2.7.1 Método de Monte Carlo 23
2.7.2 Distribuição de Boltzmann 24
2.7.3 Algoritmo de Metropolis 24

3 NOVO ALGORITMO 27
3.1 Demonstração Matemática da Fórmula 2xN 27
3.2 Algoritmo 2xN 30

4 COMPUTAÇÃO PARALELA 33
4.1 Unidade de Processamento Gráfico de Propósito Geral - GPGPU 33
4.2 CUDA (*Compute Unified Device Architecture*) 35
4.2.1 Arquitetura de uma placa de vídeo NVIDIA 36
4.2.2 Modelo de Programação 37
4.2.3 Utilizando Múltiplas GPUs Simultaneamente 40

5 VERSÕES PARALELAS IMPLEMENTADAS 42
5.1 Algoritmo NxN Paralelo 43
5.1.1 Geração Automática da Configuração de Execução 45
5.1.2 Execução do Kernel 47
5.2 Algoritmo 2xN Paralelo para uma GPU 47
5.3 Algoritmo 2xN Paralelo para Múltiplas GPUs 49

6 RESULTADOS EXPERIMENTAIS 51
6.1 Ambiente Experimental 51
6.2 *Métricas de Medida de Desempenho* 52
6.2.1 Considerações sobre Geometria 53
6.3 Comparação entre as Implementações Sequenciais dos Algoritmos . 54
6.4 Comparação entre as Implementações Paralelas e Sequenciais dos Algoritmos 55
6.4.1 Implementação NxN 55
6.4.2 Implementação 2xN 56
6.5 Comparação entre a Versão com uma GPU e a Versão com Múltiplas GPUs 57
6.6 Comparação entre os valores de energia obtidos 59

7 TRABALHOS CORRELATOS 60

8 CONCLUSÕES 62

APÊNDICES 63

REFERÊNCIAS 69

LISTA DE ILUSTRAÇÕES

2.1 Partícula com carga q e massa m em movimento sobre uma circunferência de raio r. 18

2.2 Exemplo de formação de domínios magnéticos em materiais ferromagnéticos. . 21

2.3 Fluxograma exemplo do algoritmo de Metropolis, onde ΔE é a variação de energia obtida (antes e após a mudança do *spin*). 25

3.1 Configuração de um sistema bidimensional composto por 4 *spins*. O *spin* com orientação trocada está destacado na parte inferior esquerda da figura. . . 28

4.1 Processamento sequencial(à esquerda) e processamento paralelo(à direita) de uma mesma fila de tarefas de processamento. 34

4.2 Componentes da CPU e da GPU. Retirado de [16]. 35

4.3 Interior do *chip* gráfico da GeForce 8800 GTX. Retirado de [20]. 37

5.1 Exemplo mostrando as etapas de *tilling* na GPU. 44

6.1 Exemplo de objetos implícitos gerados pelo simulador. 54

6.2 Linha do tempo para a execução da configuração 100x100x100 com 6 GPUs. . 58

6.3 Energia total do sistema ao longo da simulação. 59

6.4 Média de energia do sistema ao longo da simulação. 59

LISTA DE TABELAS

6.1 Número de total de *spins* para cada tipo de objeto. 54
6.2 Tempo médio de execução dos algoritmos NxN e 2xN sequencial, em segundos. 55
6.3 Tempo médio de execução dos algoritmos NxN sequencial e NxN paralelo, em segundos. 56
6.4 Tempo médio de execução dos algoritmos 2xN sequencial e 2xN paralelo, em segundos. 57
6.5 Tempo médio de execução do algoritmo 2xN paralelo, em segundos, quando executado na Tesla C1060. 58

B.1 Tempo médio de execução dos algoritmos NxN e 2xN sequencial, em segundos. 68
B.2 Tempo médio de execução dos algoritmos NxN sequencial e NxN paralelo, em segundos. 68
B.3 Tempo médio de execução dos algoritmos 2xN sequencial e 2xN paralelo, em segundos. 68

1 INTRODUÇÃO

1.1 Motivação

Os fenômenos magnéticos são amplamente utilizados no desenvolvimento de novas tecnologias, como sistemas de geração e distribuição de energia, dispositivos eletrônicos e de telecomunicações, e sistemas de conversão eletromagnética, dentre muitas outras. Dispositivos que utilizam esses fenômenos estão muitas vezes presentes em nossas atividades cotidianas, muito embora sem que muitos se dêem conta disso, como é o caso dos discos rígidos dos computadores, cartões de crédito, televisores, aparelhos de videocassete e dos exames de ressonância magnética.

Para uma melhor análise e compreensão do comportamento dos fenômenos magnéticos, é imprescindível o estudo destes em escala nanométrica. Foi assim que os físicos Albert Fert e Peter Grünberg descobriram, em trabalhos simultâneos e independentes, o efeito da Magnetorresistência Gigante, trabalho que lhes valeu o prêmio Nobel de Física em 2007. A descoberta deste fenômeno proporcionou um aumento da ordem de 100 vezes na capacidade de armazenamento dos discos rígidos. Além do mais, a própria origem do magnetismo está associada a duas propriedades dos elétrons em escala atômica: a) o momento angular, associada ao movimento destes ao redor do núcleo atômico; e b) *spins*, associada a forma como os elétrons ocupam os níveis de energia no átomo. Na mecânica quântica, o *spin* de um átomo refere-se às possíveis orientações que partículas subatômicas (prótons, elétrons, e alguns núcleos atômicos) têm quando estão sob ação, ou não, de um campo magnético externo. O *spin* não possui uma interpretação clássica, ou seja, é um fenômeno estritamente quântico.

O estudo de fenômenos magnéticos em escala nanométrica pode ser facilitada com o emprego de simuladores. Estes são capazes de fornecer um conjunto detalhado de informações do comportamento dos fenômenos magnéticos quando sujeitos às mais diversas situações, que podem ser criadas e alteradas livremente pelo cientista. Detalhes visuais e numéricos gerados a partir da modelagem computacional proporcionam ao cientista uma importante ferramenta que auxilia no melhor entendimento de fenômenos físicos. Entretanto, o custo computacional para o cálculo das interações entre *spins*, cuja

complexidade computacional é da ordem de $O(N^2)$, onde N é o número de spins que formam o sistema sendo simulado, é hoje um fator limitante das simulações. Em última instância, o número de spins que será simulado é limitado pelo custo do cálculo destas interações, impedindo assim a simulação de grandes sistemas. Por exemplo, a simulação de um sistema constituído de $N = 10.648$ *spins* é executada em um processador Intel Core 2 Quad de 2.83 Ghz em 10.800 segundos, aproximadamente.

1.2 Objetivo

O principal objetivo deste trabalho é apresentar alternativas para que os custos computacionais associados ao cálculo de energia resultante das interações entre *spins* possa ser significativamente reduzido, de modo que sistemas maiores, e portanto mais próximos às situações reais, possam ser simulados.

Neste sentido, um novo algoritmo para o cálculo da interação entre *spins* é proposto. Quando comparado ao algoritmo tradicionalmente utilizado para calcular a energia de um sistema composto por materiais ferromagnéticos no estado condensado, o novo algoritmo permite ganhos de desempenho de até 1.160 vezes. Ganhos adicionais de desempenho podem ser obtidos com o auxílio de ferramentas e arquiteturas para computação de alto desempenho, como GPGPUs (*General-purpose Graphics Processing Units*). Neste caso, os ganhos de desempenho são superiores a 11.000 vezes. Adicionalmente, o uso conjunto do novo algoritmo e da plataforma para computação de alto desempenho permitem que sistemas com 16 milhões de *spins* sejam simulados.

1.3 Organização

Esta dissertação foi organizada da seguinte forma. Os dois próximos capítulos apresentam uma revisão dos temas necessários para a compreensão deste trabalho. Enquanto o capítulo 2 apresenta uma visão breve do comportamento físico dos sistemas compostos por materiais ferromagnéticos no estado condensado, o capítulo 3 apresenta uma visão geral sobre a plataforma e o ambiente de computação paralela utilizada neste trabalho: GPGPUs e CUDA, respectivamente. O capítulo 4 apresenta o algoritmo proposto, bem como sua implemetação computacional. No capítulo 5 são apresentados os resultados das simulações computacionais realizadas. Por fim, o capítulo 6 traz as considerações finais e

perspectivas de trabalhos futuros.

2 MODELO FÍSICO-MATEMÁTICO

Este capítulo apresenta a base teórica fundamental que descreve o comportamento físico dos sistemas compostos por materiais ferromagnéticos no estado condensado.

2.1 Introdução

A Física da Matéria Condensada é o ramo da física que estuda as propriedades macroscópicas da matéria em sua fase condensada. Esta fase se manifesta em sistemas constituídos por um número extremamente grande de partículas atômicas, onde a interação entre elas é forte. A fase ferromagnética dos *spins* em um sistema cristalino é um exemplo de fase condensada.

O comportamento de um sistema, do ponto de vista macroscópico, pode ser compreendido partindo-se do estudo de sua natureza atômica. Energia, temperatura e volume são propriedades macroscópicas da matéria, enquanto posição, velocidade e momento angular podem ser tratadas como propriedades microscópicas de seus constituintes. As leis que governam o comportamento mecânico das partículas atômicas de um sistema são fornecidas pela Mecânica Estatística (ou Física Estatística).

Alguns problemas em Mecânica Estatística são exatamente solúveis. São os problemas triviais, em que a partir das propriedades microscópicas de um sistema molecular é possível calcular analiticamente e sem aproximações o seu comportamento. No entanto, os problemas não triviais necessitam de um modelo alternativo que permita a sua resolução. O cálculo do potencial de interação entre partículas é um exemplo do conjunto de problemas da Mecânica Estatística considerado não trivial devido à complexidade de se encontrar soluções analíticas para as equações que modelam este problema. Neste caso, a simulação computacional torna-se uma ferramenta poderosa para a Mecânica Estatística dos estados da matéria condensada, onde obter resultados experimentais em situações extremas com as tecnologias atuais é muito difícil ou mesmo quase impossível, visto que os sistemas dessa classe são altamente complexos de serem analisados.

O objetivo da simulação computacional é permitir que seja feita a evolução temporal (ou mapeamento configuracional) de um sistema, até que este atinja um estado de equilíbrio. Uma técnica comumente empregada em tais simulações computacionais é o método de Monte Carlo. Para que tal simulação seja feita, a função matemática (ou as energias de interação) do problema deve ser conhecida. Neste caso, a função de interesse corresponde à do potencial de interação intermolecular. Os conceitos da Mecânica Estatística são assim aplicados para a obtenção das propriedades termodinâmicas macroscópicas do sistema.

O intuito deste trabalho é o de investigar o processo de magnetização de sistemas compostos por materiais ferromagnéticos via simulação computacional. Para isso, os potenciais de interação de Heisenberg serão adotados para descrever a energia da estrutura atômica destes sistemas e o método de Monte Carlo será empregado para validar sua evolução ao longo da simulação.

2.2 Magnetismo nos Materiais

A causa física do magnetismo nos materiais se deve aos dipolos atômicos magnéticos. Os dipolos magnéticos (ou momentos magnéticos) resultam de dois tipos diferentes do movimento dos elétrons. Um deles é o movimento orbital sobre seu núcleo atômico. O outro é devido ao momento angular intrínseco dos elétrons chamado *spin*, que por sua vez possui um momento de dipolo magnético associado a ele.

2.3 Momento de Spin Atômico

A mecânica clássica define o momento de *spin* como a rotação do elétron em torno de seu próprio centro de massa. O *spin* está associado a um momento angular e é representado através da sua quantização. O momento angular possui magnitude, isto é, quão rápido o elétron está girando, e uma direção, que é o eixo de rotação da partícula. De acordo com a mecânica quântica, o momento angular de *spin* de um sistema é dado por:

$$S = \sqrt{s \cdot (s+1)} \cdot \hbar \tag{2.1}$$

onde $\hbar$ é a constante de Planck e s é um número fracionário na forma $n/2$, em que n é um inteiro ≥ 0.

Além disso, a componente do momento angular medida ao longo de um eixo cartesiano (o eixo-z, por exemplo) pode assumir os seguintes valores:

$$\hbar, s_z = -s; -s+1; ...; s-1; s.$$

Há $2s+1$ valores possíveis para s_z. Pelo princípio de exclusão de Pauli, o elétron possui $s = 1/2$. Portanto, os valores de s_z para o elétron são $s_z = 1/2$ e s_z= -1/2. Estes valores se referem a direção para onde o *spin* está apontando no eixo-z e correspondem a para cima e para baixo, respectivamente.

A aplicação destes conceitos de mecânica quântica permite a compreensão de um sistema atômico em seu nível fundamental, o que por sua vez implica na necessidade da análise de cada partícula individual do sistema para que a solução de um problema possa ser encontrada. Porém, um sistema atômico contém um número elevado de partículas. Para simplificar o modelo quântico dos *spins*, Werner Karl Heisenberg propôs um modelo [1] onde os *spins* são sítios de uma rede cristalina. A direção de cada *spin* da rede é tratada como um vetor tridimensional $S = (s_x, s_y, s_z)$, onde s_x, s_y e s_z são valores estimados para a direção do *spin* em cada eixo.

2.4 Momento Magnético Orbital do Elétron

Considere uma partícula carregada de massa m e carga q, movendo-se numa órbita fechada de raio r, com velocidade v, como é o caso do elétron no átomo (Figura 2.1). Este movimento equivale a uma corrente elétrica circular. Pela teoria eletromagnética, uma corrente elétrica circular gera um campo de dipolo magnético, o que é verificado para o elétron em órbita em torno do núcleo. O módulo do momento angular do elétron é dado por:

$$L = mvr \tag{2.2}$$

O momento de dipolo magnético orbital de um condutor circular é definido como o

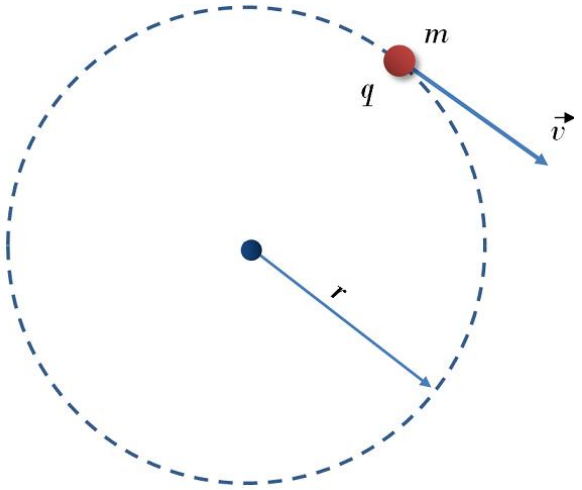


Figura 2.1: Partícula com carga q e massa m em movimento sobre uma circunferência de raio r.

produto da corrente pela área do círculo:

$$\mu = IA = I\pi r^2 \tag{2.3}$$

Supondo que T seja o tempo que o elétron leva para descrever uma órbita completa, a corrente é expressa por:

$$I = \frac{q}{T} = \frac{qv}{2\pi r} \tag{2.4}$$

Substituindo a corrente na Equação 2.3 pela Equação 2.4, o momento magnético encontrado é:

$$\mu = IA = \frac{qv}{2\pi r}\pi r^2 = \frac{qvr}{2} \tag{2.5}$$

Da Equação 2.2, tem-se que $vr{=}L/m$. Logo, a Equação 2.5 pode ser escrita como:

$$\mu = \frac{q}{2m}L \tag{2.6}$$

Visto que a carga em questão é o elétron, o momento angular e o momento magnético apontam para direções opostas, uma vez que, $q{=}{-}e$. A massa do elétron é denominada por $m{=}m_e$. Desta forma, a Equação 2.6 expressa na forma vetorial é definida por:

$$\vec{\mu} = -\frac{e}{2m_e}\vec{L} \tag{2.7}$$

A Equação 2.7 mostra a relação entre o momento de dipolo magnético e o momento

angular, também conhecida como paralelismo magnético-mecânico.

No entanto, esta relação foi formulada de maneira clássica. O estudo de sistemas em escala atômica tem maior precisão do ponto de vista quântico. No caso da Equação 2.7, a relação permanece a mesma tanto para o caso clássico quanto para o quântico, exceto para o momento angular de *spin* do elétron. Como o momento angular é quantizado, o momento magnético também é quantizado [2]. O quantum do momento angular é $\hbar = h/2\pi$, onde h é a constante de Planck. Dessa forma, o momento magnético em termos de $\vec{L}/\hbar$ é:

$$\vec{\mu_\ell} = -\frac{e\hbar}{2m_e}\frac{\vec{L}}{\hbar} = -\mu_B\frac{\vec{L}}{\hbar} \tag{2.8}$$

onde $\mu_B = \frac{-e\hbar}{2m_e}$ é denominado **magnéton de Bohr**. O momento magnético do elétron devido ao seu momento angular de *spin*, $\vec{S}$, é dado por:

$$\vec{\mu_s} = -2\mu_B\frac{\vec{S}}{\hbar} \tag{2.9}$$

2.5 Magnetização e Susceptibilidade Magnética

Os materiais podem ser classificados de acordo com o nível de magnetismo adquirido por eles quando estão sob a ação de um campo magnético externo. O nível de magnetismo de um material é medido a partir do momento de dipolo magnético de seus elétrons. O momento de dipolo magnético reage à influência de um campo magnético externo alinhando-se ou permanecendo a uma direção muito próxima a ele e pode ter o mesmo sentido ou sentido contrário ao campo. Essa reação produz o que é chamado de magnetização interna M do material e consiste da soma dos momentos magnéticos das partículas de um sistema atômico. A partir da magnetização interna e do campo magnético externo H define-se matematicamente o nível de magnetismo de um sistema atômico pela relação $\chi = M/H$, onde χ é um valor adimensional obtido para definir o nível de magnetização do material do qual o sistema é formado, chamado de susceptibilidade magnética.

Para $\chi \gg 1$ o material classifica-se como ferromagnético ou ferrimagnético, dependendo da orientação dos *spins* em relação ao campo magnético externo; diamagnético para $\chi < 1$; paramagnético para $\chi > 0$ e antiferromagnético para $\chi \ll 1$.

2.5.1 Paramagnetismo e Ferromagnetismo

Em seus experimentos, o físico Pierre Curie analisou alguns metais alcalinos sob a ação de um campo magnético com variação de temperatura. Curie constatou que, a baixas temperaturas, os momentos magnéticos (*spins*) tendem a se alinhar com o campo, reforçando sua intensidade, ou seja, existe um magnetismo macroscopicamente notável nesta situação. A medida em que a temperatura é elevada, o número de *spins* alinhados diminui até a energia térmica ser suficiente para desordená-los completamente. Neste momento ocorre a desmagnetização do sistema.

A relação proporcionalmente inversa entre temperatura e magnetização ficou conhecida como **Lei de Curie** e a temperatura crítica T_c em que o sistema se desmagnetiza é denominada **Ponto de Curie**.

O processo abrupto de desmagnetização dos átomos após atingirem a T_c é chamado transição de fase. Os materiais na fase paramagnética são aqueles com temperatura acima do Ponto de Curie. Já os materiais abaixo desta temperatura estão na fase ferromagnética.

O paramagnetismo é observado quando a interação entre os momentos magnéticos é fraca. Na ausência de um campo magnético externo, a magnetização interna é nula devido a agitação térmica dos átomos. A energia produzida pelos momentos magnéticos nesta situação é menor do que a energia térmica dos átomos. Por este fato os materiais na fase paramagnética apresentam susceptibilidade magnética positiva, mas pequena.

O ferromagnetismo é causado por uma forte interação de troca de energia entre *spins* de átomos vizinhos. Supondo-se que dois *spins* i e j sejam vizinhos, a energia de troca entre eles é expressa por:

$$E_{ij} = -J[\vec{S}_i \cdot \vec{S}_j] \tag{2.10}$$

onde $J > 0$ é a intensidade da energia de troca.

A aplicação de um campo magnético de pouca intensidade é capaz de produzir um forte grau de alinhamento destes *spins*. Este grau pode ser forte o bastante para manter os *spins* alinhados mesmo que o campo magnético aplicado seja retirado, explicando seu alto valor de susceptibilidade magnética. A influência de um *spin* sobre seus vizinhos leva à formação de domínios magnéticos (Figura 2.2). Um domínio magnético se constitui de uma região microscópica do sistema onde os *spins* constituintes ficam alinhados, mas a

direção de alinhamento varia entre os domínios. Dessa forma, se uma área macroscópica de um material ferromagnético for analisada, a magnetização será mínima. Entretanto o desalinhamento dos *spins* é menor do que o encontrado nos materiais paramagnéticos. Na presença de um campo externo, os domínios alinham-se mais facilmente no mesmo sentido do campo, aumentando a magnetização total do sistema.

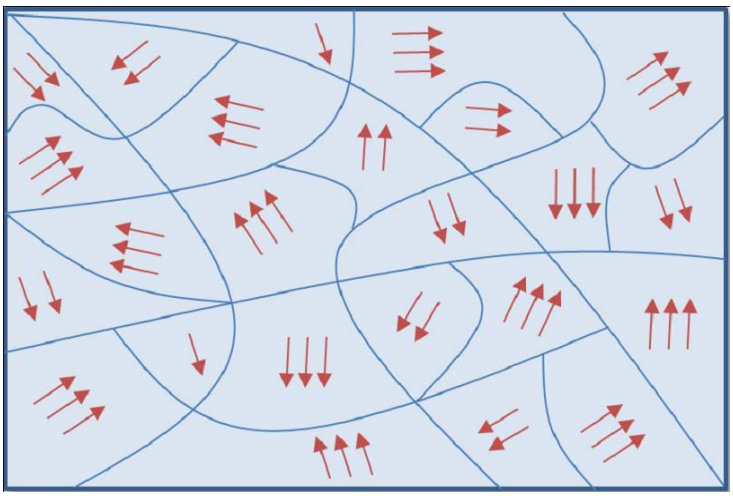


Figura 2.2: Exemplo de formação de domínios magnéticos em materiais ferromagnéticos.

2.6 Energia Potencial de Interação

Quando as partículas (moléculas ou átomos) de um sistema se aproximam uma das outras, dois fenômenos podem ocorrer: elas podem reagir ou elas podem interagir. Ao interagirem, as partículas se atraem ou se repelem entre si sem ocorrer a quebra ou a formação de novas ligações químicas. Estas interações são chamadas de intermoleculares.

As interações intermoleculares têm origem nos fenômenos elétricos e magnéticos e fazem com que uma partícula influencie o comportamento de outra em suas proximidades. Uma vez que estas interações provêm do contato não reativo entre as partículas, pode-se afirmar que a distância de separação entre elas interfere no comportamento das forças intermoleculares, fazendo com que estas forças variem inversamente à distância de separação entre as partículas interagentes, ou seja, entre partículas muito próximas a força de interação será maior. Desse modo, as interações intermoleculares podem ser agrupadas em interações de curto alcance (aquelas que atuam a pequenas distâncias de separação intermolecular) e interações de longo alcance, que atuam a grandes distâncias de

separação intermolecular. Considerando a interação entre duas partículas i e j, a energia intermolecular entre elas é expressa da seguinte forma:

$$E(\text{intermolecular}) = E_{i-j} - (E_i + E_j). \tag{2.11}$$

Isto corresponde a sua decomposição em vários componentes[3]:

$$E(\text{intermolecular}) = E(\text{longo alcance}) + E(\text{curto alcance}). \tag{2.12}$$

Cada componente fornece um tipo de informação a respeito do comportamento do fenômeno observado. Para investigar a magnetização dos materiais ferromagnéticos, a energia potencial de interação adotada neste trabalho possui duas componentes de longo alcance e uma de curto alcance, seguindo o modelo clássico de *spins* de Heisenberg.

Considerando-se um sólido cristalino de geometria tridimensional, a componente de curto alcance referente a energia de troca é:

$$E_f = -J \sum_{i,k=1 \; e \; i \neq k}^{N} \mathbf{S}_i \cdot \mathbf{S}_k \tag{2.13}$$

A primeira componente de longo alcance se refere a energia potencial gerada pela interação entre um dipolo magnético com o campo magnético criado por um segundo dipolo magnético, a interação dipolo-dipolo. O Hamiltoniano que descreve esta energia é dado por:

$$E_{dd} = \frac{A}{2} \sum_{i,j=1 \; e \; i \neq j}^{N} \left\{ \frac{\mathbf{S}_i \cdot \mathbf{S}_j}{|\mathbf{r}_{ij}|^3} - 3 \frac{[\mathbf{S}_i \cdot \mathbf{r}_{ij}]\,[\mathbf{S}_j \cdot \mathbf{r}_{ij}]}{|\mathbf{r}_{ij}|^5} \right\} \tag{2.14}$$

Onde $\mathbf{S}_i$ denota o *spin* da partícula analisada, $\boldsymbol{r}_{ij} = \boldsymbol{r}_i - \boldsymbol{r}_j$ é o vetor posição que separa os átomos i e j, a constante A corresponde à intensidade de interação dipolar.

A segunda componente de longo alcance é o termo Zeeman, que gera uma energia potencial magnética resultante do torque que o campo magnético exerce sobre um momento de dipolo:

$$E_z = -D \sum_{i=1}^{N} (\mathbf{S}_i \cdot \boldsymbol{H}) \tag{2.15}$$

A expressão da energia potencial de interação, objeto deste estudo, é obtida a partir

da união de suas três componentes, sendo portanto escrita como:

$$E_t = \frac{A}{2} \sum_{i,j=1 \; e \; i \neq j}^{N} \left\{ \frac{\mathbf{S}_i \cdot \mathbf{S}_j}{|\mathbf{r}_{ij}|^3} - 3\frac{[\mathbf{S}_i \cdot \mathbf{r}_{ij}]\,[\mathbf{S}_j \cdot \mathbf{r}_{ij}]}{|\mathbf{r}_{ij}|^5} \right\} - J \sum_{i,k=1 \; e \; i \neq k}^{N} \mathbf{S}_i \cdot \mathbf{S}_k - D \sum_{i=1}^{N} (\mathbf{S}_i \cdot \boldsymbol{H}) \quad (2.16)$$

2.7 Métodos Numéricos Aplicados à Simulação Computacional

2.7.1 Método de Monte Carlo

O método de Monte Carlo [4] foi desenvolvido pelos cientistas Stanislaw Ulam, Enrico Fermi, John von Neumann, and Nicholas Metropolis para estudar a difusão dos nêutrons durante a segunda guerra mundial.

O nome Monte Carlo faz alusão ao Grande Casino de Mônaco situado em Monte Carlo. O sorteio aleatório de números e a repetição de procedimentos para se chegar a um resultado são semelhantes à sistemática envolvendo jogos de azar. Os algoritmos computacionais com estas características pertencem à classe dos Métodos de Monte Carlo.

O método trabalha com três bases simples:

1. Definir um domínio de entradas possíveis do sistema estudado.

2. Gerar novas entradas de acordo com o domínio aleatório e fazer um cálculo determinista sobre elas.

3. Agregar os resultados dos cálculos individuais para o resultado final.

Em resumo, os resultados são gerados sobre uma distribuição de probabilidades e a amostra significativa obtida por tentativas aleatórias é usada para aproximar uma função matemática de interesse, isto é, não há a necessidade de reproduzir todas as configurações do sistema. A precisão do resultado final depende em geral do número de tentativas.

A Física Estatística e Computacional utiliza este método em diversas áreas tais como física do estado sólido, mecânica dos fluidos, teoria do campo reticulado, dentre outras.

2.7.2 Distribuição de Boltzmann

O físico austríaco Ludwig Eduard Boltzmann conseguiu relacionar a estatística com a termodinâmica ao encontrar uma distribuição que descreve a probabilidade de um sistema em equilíbrio térmico estar em um determinado estado de energia. Esta distribuição tornou-se útil para estudar algumas propriedades, tais como a entropia e energia interna, a partir da distribuição das partículas de um sistema. Todo sistema tem um número de configurações possíveis para seus elementos e cada uma dessas configurações é considerada um **microestado**. Cada conjunto de microestados que apresentam a mesma energia é definido como um **macroestado** do sistema. Valendo-se da hipótese que todos os estados microscópicos acessíveis a um sistema fechado em equilíbrio são igualmente prováveis, Boltzmann concluiu que a probabilidade P_m de ocorrência de um macroestado m estar a uma certa energia E_m é proporcional ao número de microestados que definem este macroestado, ou seja, o número de microestados que apresentam energia E_m [5]. Boltzmann descreveu a probabilidade P_m como sendo:

$$P_m = \frac{e^{-E_m/k_B T}}{Z} \tag{2.17}$$

onde $k_B = 1,380662 \cdot 10^{-23} J/K$ é a **constante de Boltzmann**, T é a temperatura do sistema e Z é a **função partição** que normaliza a distribuição.

Desde então, a distribuição de Boltzmann proporcionou o desenvolvimento do estudo de sistemas através da Física Estatística e de algoritmos de simulação em Física Computacional. O algoritmo de Metrópolis, explicado a seguir, baseia-se nesta distribuição.

2.7.3 Algoritmo de Metropolis

O algoritmo de Metropolis [4], também criado por Nicholas Metropolis, determina valores esperados de propriedades de um sistema em simulação calculando-se uma média sobre uma amostra (um microestado válido), que é obtida através da geração de números aleatórios. O algoritmo é concebido de modo a se obter uma amostra que siga a distribuição de Boltzmann: o sistema a ser simulado deve se encontrar em temperaturas diferentes de zero e o valor da energia para cada partícula do sistema deve ser conhecido. Entretanto, a função partição Z é difícil de ser calculada, visto que ela depende do

conhecimento de todos os microestados possíveis. Metropolis notou que poderia eliminar esse problema usando uma **cadeia de Markov**[6]. Desse modo, a geração do próximo microestado só dependeria do microestado anterior. Como os eventos são independentes, a probabilidade de transição $P_{m\to n}$ de um microestado m para um novo microestado n pode ser escrita como:

$$\begin{aligned} P_{m\to n} &= \frac{P_n}{P_m} \\ &= \frac{e^{-E_n/k_BT}}{Z} \Big/ \frac{e^{-E_m/k_BT}}{Z} \\ &= e^{-(E_n-E_m)/k_BT} = e^{-\Delta E/k_BT} \end{aligned} \tag{2.18}$$

A Figura 2.3 descreve as etapas do algoritmo de Metropolis durante a simulação computacional de um sistema ferromgnético.

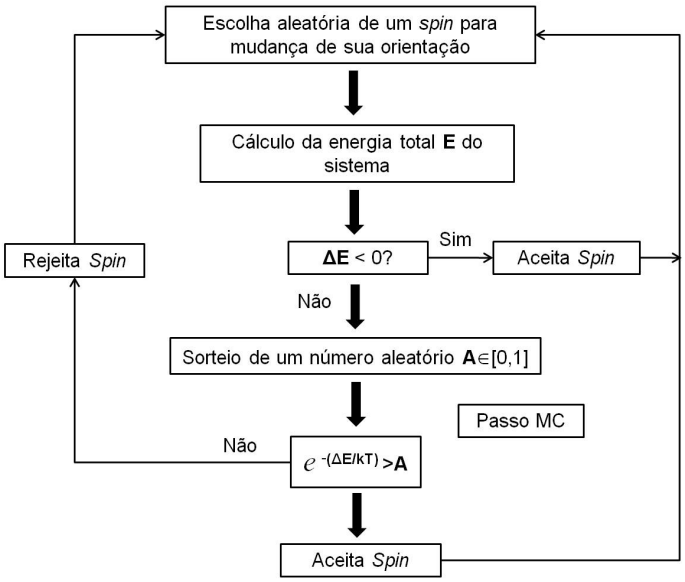


Figura 2.3: Fluxograma exemplo do algoritmo de Metropolis, onde ΔE é a variação de energia obtida (antes e após a mudança do *spin*).

As etapas mostradas no fluxograma acontecem da seguinte forma:

a) Definição das condições físicas iniciais do sistema (arranjo espacial dos *spins*);

b) Escolha arbitrária de um *spin* que terá sua direção alterada;

c) Mudança arbitrária na direção de um *spin*;

d) Cálculo da nova energia total do sistema;

e) Se a variação de energia obtida for menor do que zero, então a configuração dos

spins, modificada na etapa (c), se torna válida para o sistema. Caso contrário, realizam-se mais duas etapas, descritas nos subitens (e1) e (e2) abaixo:

e1) Geração de um número aleatório $\boldsymbol{A}$ no intervalo [0,1].

e2) Se $e^{(-\Delta E/K_B t)} > A$, então a nova configuração é válida. Do contrário, o sistema retorna à configuração anterior.

f) Repetem-se os passos (b), (c), (d), e (e) até que alguma condição de parada seja satisfeita. Cada uma dessas repetições é dita um passo Monte Carlo (MC).

3 NOVO ALGORITMO

Foram apresentadas no capítulo anterior as equações que expressam a energia potencial de interação entre *spins*. Pela equação, podemos facilmente verificar que o custo de sua implementação computacional é de O(N^2), onde N é o número de *spins* do sistema. Esse custo decorre da necessidade de se calcular a contribuição de energia que todos os *spins* do sistema exercem uns sobre os outros. Deste modo, o custo computacional aumenta quadraticamente com o número de *spins* no sistema, o que em última instância limita o tamanho dos sistemas sendo simulados.

Este capítulo apresenta a principal contribuição deste trabalho, um novo algoritmo para calcular a energia do sistema com um custo computacional reduzido em relação ao algoritmo original. Este algoritmo foi batizado de **2xN**.

3.1 Demonstração Matemática da Fórmula 2xN

Como foi apresentado na Figura 2.3, a transição de um microestado para outro está associada a variação de energia(ΔE) produzida com a troca de orientação(**flipagem**) de um *spin* escolhido ao acaso. A energia do sistema é calculada a cada troca de orientação para que se saiba o valor de ΔE em cada iteração da simulação. A revisão da Equação 2.16 revelou que é possível calcular diretamente o ΔE para cada microestado gerado nas iterações subsequentes a partir da energia obtida na primeira iteração. O ΔE nada mais é do que a diferença entre a energia dos dipolos formados com o *spin* cuja orientação foi alterada e a energia dos dipolos formados com este mesmo *spin* antes da troca de orientação. A demonstração a seguir mostra como a Equacao 2.16 foi modificada para produzir ΔE.

Para a demostração a seguir serão adotadas algumas convenções, a saber:

- E^m = energia pré-flipagem;
- E^n = energia pós-flipagem;
- E_i = contribuição de energia de cada *spin*, onde i é o seu índice;
- E_{dd} = energia dipolar magnética;

- E_f = energia ferromagnética;
- E_z = energia de interação *spin*/ campo magnético aplicado;
- S_{ij}, S_{ijij} = primeiro e segundo membro da Equação 2.14, onde i e j são os índices dos *spins* envolvidos.

Considere um sistema de estrutura bidimensional formado por quatro *spins*, como o apresentado na Figura 3.1. Agora suponha que a troca de orientação ocorra para o *spin* número 2. O vetor do *spin* com orientação trocada será referido como S_2*. Nessas condições tem-se que:

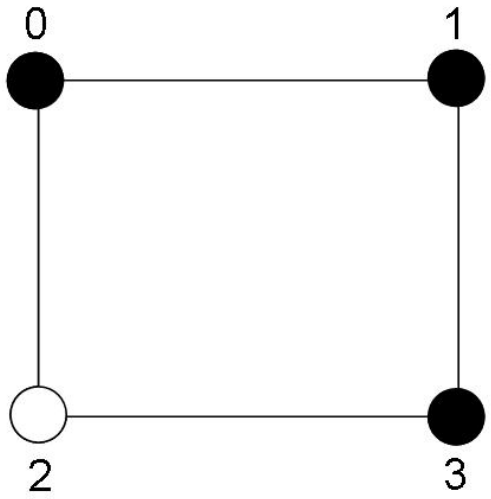


Figura 3.1: Configuração de um sistema bidimensional composto por 4 *spins*. O *spin* com orientação trocada está destacado na parte inferior esquerda da figura.

$$
\begin{aligned}
\Delta E &= E^n - E^m \\
&= \sum_{i=0}^{3} {E^n}_i - \sum_{i=0}^{3} {E^m}_i \\
&= {E^n}_0 + {E^n}_1 + {E^n}_2 + {E^n}_3 - ({E^m}_0 + {E^m}_1 + {E^m}_2 + {E^m}_3) \\
&= {E^n}_0 + {E^n}_1 + {E^n}_2 + {E^n}_3 - {E^m}_0 - {E^m}_1 - {E^m}_2 - {E^m}_3 \\
&= (E^n{}_{dd0} - E^n{}_{f0} - E^n{}_{z0}) + (E^n{}_{dd1} - E^n{}_{f1} - E^n{}_{z1}) + \\
&\quad +(E^n{}_{dd2} - E^n{}_{f2} - E^n{}_{z2}) + (E^n{}_{dd3} - E^n{}_{f3} - E^n{}_{z3}) - \\
&\quad -(E^m{}_{dd0} - E^m{}_{f0} - E^m{}_{z0}) - (E^m{}_{dd1} - E^m{}_{f1} - E^m{}_{z1}) - \\
&\quad -(E^m{}_{dd2} - E^m{}_{f2} - E^m{}_{z2}) - (E^m{}_{dd3} - E^m{}_{f3} - E^n{}_{z3})
\end{aligned} \tag{3.1}
$$

Rearranjando-se os termos da Equação 3.1 de forma que o primeiro esteja em função das energias dipolares, o segundo em função das energias ferromagnéticas e o terceiro em função da energia Zeeman, obtem-se:

$$
\begin{aligned}
\Delta E &= (E^n{}_{dd0} + E^n{}_{dd1} + E^n{}_{dd2} + E^n{}_{dd3} - E^m{}_{dd0} - E^m{}_{dd1} - E^m{}_{dd2} - E^m{}_{dd3}) + \\
&\quad +(-E^n{}_{f0} - E^n{}_{f1} - E^n{}_{f2} - E^n{}_{f3} + E^m{}_{f0} + E^m{}_{f1} + E^m{}_{f2} + E^m{}_{f3}) + \\
&\quad +(-E^n{}_{z0} - E^n{}_{z1} - E^n{}_{z2} - E^n{}_{z3} + E^m{}_{z0} + E^m{}_{z1} + E^m{}_{z2} + E^m{}_{z3})
\end{aligned} \tag{3.2}
$$

Substituindo-se a contribuição de cada *spin* E_i segundo a Equação 2.16, a Equação 3.2 é reescrita como:

$$
\begin{aligned}
\Delta E &= A \cdot [(S_{01} - S_{0101}) + (S_{02*} - S_{02*02*}) + (S_{03} - S_{0303}) + (S_{12*} - S_{12*12*}) + \\
&\quad +(S_{13} - S_{1313}) + (S_{2*3} - S_{2*32*3}) - \\
&\quad -(S_{01} - S_{0101}) - (S_{02} - S_{0202}) - (S_{03} - S_{0303}) - (S_{12} - S_{1212}) - \\
&\quad -(S_{13} - S_{1313}) - (S_{23} - S_{2323})] - \\
&\quad -J \cdot (S_{01} + S_{02*} + S_{10} + S_{13} + S_{2*0} + S_{2*3} + S_{31} + S_{32*} \\
&\quad -S_{01} - S_{02} - S_{10} - S_{13} - S_{20} - S_{23} - S_{31} - S_{32}) - \\
&\quad -D \cdot (S_{0H} + S_{1H} + S_{2*H} + S_{3H} - S_{0H} - S_{1H} - S_{2H} - S_{3H})
\end{aligned}
$$

Observe que as parcelas de energia fornecidas pelos *spins* que não tiveram suas orientações alteradas se cancelam. Logo:

$$\begin{aligned}\Delta E &= A \cdot [(S_{02*} - S_{02*02*}) + (S_{12*} - S_{12*12*}) + (S_{2*3} - S_{2*32*3}) - \\ &\quad -(S_{02} - S_{0202}) - (S_{12} - S_{1212}) - (S_{23} - S_{2323}) - \\ &\quad -J \cdot (S_{02*} + S_{2*0} + S_{2*3} + S_{32*} \\ &\quad -S_{02} - S_{20} - S_{23} - S_{32}) - \\ &\quad -D \cdot (S_{2*H} - S_{2H}) \end{aligned} \tag{3.3}$$

Visto que a interação de um *spin i* para o *spin j* tem a mesma intensidade da interação do *spin j* para o *spin i*, a Equação 3.3 pode ser reescrita como:

$$\begin{aligned}\Delta E &= A \cdot [(S_{02*} - S_{02*02*}) + (S_{12*} - S_{12*12*}) + (S_{2*3} - S_{2*32*3}) - \\ &\quad -(S_{02} - S_{0202}) - (S_{12} - S_{1212}) - (S_{23} - S_{2323})] - \\ &\quad -J \cdot (2S_{02*} + 2S_{2*3} - 2S_{02} - 2S_{23}) - \\ &\quad -D \cdot (S_{2*H} - S_{2H}) \\ \Delta E &= (E^n{}_{dd2} - E^m{}_{dd2}) - (2E^n{}_{f2} - 2E^m{}_{f2}) - (E^n{}_{z2} - E^m{}_{z2}) \\ &= (E^n{}_{dd2} - 2E^n{}_{f2} - E^n{}_{z2}) - (E^m{}_{dd2} - 2E^m{}_{f2} - E^m{}_{z2}) \\ &= E^n{}_2 - E^m{}_2 \end{aligned} \tag{3.4}$$

3.2 Algoritmo 2xN

Após definir as condições iniciais do sistema a ser simulado, o número de iterações do algoritmo de Metropolis e montar a matriz de dados sobre o sistema, o algoritmo está pronto para ser executado.

Na primeira iteração o algoritmo calcula a energia inicial, produzida com a configuração dos *spins* sem a troca de orientação. Esta etapa é feita pelo método tradicional calculando-se as energias individuais de todos os *spins*. A computação das energias individuais é realizada em três etapas. A primeira calcula a energia dipolar, a segunda calcula a energia ferromagnetica e a terceira a energia Zeeman. Para a obtenção

da energia dipolar, o algoritmo itera pela matriz de dados, onde cada posição da matriz representa um *spin* e computa a Equação 2.14 entre o *spin* corrente e os demais. A segunda etapa consiste em determinar a região de vizinhança do *spin* corrente e computar a Equação 2.13. Como o sistema estudado é tridimensional, cada *spin* possui de um a seis vizinhos, dependendo de sua localização geométrica. Por último, a energia Zeeman é calculada. As etapas de cálculo da energia são feitas até que todos os *spins* tenham sido acessados. A contribuição de energia de um *spin* é armazenada na posição da matriz de dados correspondente a ele. Esta contribuição é somada as demais, resultando na energia total inicial ao final da computação. O Algoritmo A.1 apresenta o código correspondente a iteração inicial.

As iterações restantes seguem o algoritmo de Metropolis propriamente dito (conforme descrito na Seção 2.7.3) e a energia é calculada pelo algoritmo 2xN. Neste ponto, há a escolha aleatória de um valor inteiro para cada um dos eixos coordenados onde a probabilidade de escolha deve ser uniforme. Para que esta condição ocorra, utilizamos o algoritmo de geração de números aleatórios Mersenne Twister [7] cuja periodicidade é de 2^{19937} . Os valores escolhidos constituem a posição espacial da partícula cujo valor de *spin* será modificado (passo (b) do algoritmo apresentado na Seção 2.7.3). Escolhida a partícula, os valores de orientação do *spin* são sorteados aleatoriamente, usando também o Mersenne Twister, e armazenados na posição correspondente da matriz (passo (c)). No entanto, o valor antigo também é armazenado, pois se este *spin* levar a formação de uma configuração incorreta do sistema, este será re-atribuído à partícula. O algoritmo 2xN propriamente dito (Algoritmo A.2) se insere no passo (d). Pela Equação 3.4 basta calcular a energia individual do *spin* com seu antigo valor de orientação e a sua energia individual com o valor modificado. Entretanto, esta modificação requer algumas mudanças durante as etapas de cálculo descritas. O algoritmo percorre a matriz de dados, acessando cada *spin* e computando a energia da primeira etapa apenas entre o *spin* corrente e o *spin* com orientação trocada. Neste caso, haverá duas computações desta etapa para cada *spin*: uma para calcular a energia sem a troca de orientação(E^n) e a outra para calcular a energia com a troca de orientação (E^m). Então a energia E^m é subtraída de E^n, fornecendo a parcela ΔE relativa a interação entre o *spin* com orientação trocada e o spin corrente. Ao percorrer a matriz de dados, há uma situação que não ocorre na etapa de cálculo da energia dipolar: é quando o *spin* corrente é o *spin* com orientação trocada. No algoritmo

tradicional nenhuma computação é feita, pois as parcelas ferromagnética e de Zeeman de energia deste *spin* acabam aparecendo na computação dos demais *spins*. Contudo no novo algoritmo as etapas de cálculo da energia ferromagnética e Zeeman são realizadas. As duas etapas também são calculadas duas vezes, produzindo as parcelas $-2E^n{}_f - E^n{}_z$ e $-2E^m{}_f 2 - E^m{}_z$. Estas parcelas são subtraídas e fornecem o ΔE relativo as etapas ferromagnética e Zeeman para o *spin* com orientação trocada. Depois de percorrer toda a matriz de dados, o algoritmo executa a soma de cada um dos ΔE calculados para a obtenção do ΔE final. Somando-se este ΔE a energia total do passo de Monte Carlo anterior, obtém-se a nova energia total do sistema. Com o resultado do ΔE no passo anterior, segue-se para a etapa (e) do algoritmo de Metropolis. Esta etapa é igual ao algoritmo tradicional.

Em termos de complexidade computacional, pode-se dizer que, excluído o primeiro passo de Monte Carlo, o novo algoritmo apresenta complexidade igual a $\mathrm{O}(N)$, onde N é o número de *spins* que formam o sistema, visto que o cálculo é realizado entre um *spin* e todos os demais que constituem o sistema. Entretanto, o primeiro passo de Monte Carlo continua tendo custo da ordem de $\mathrm{O}(N^2)$, visto que o algoritmo tradicional precisa ser utilizado para que seja calculada a energia inicial do sistema.

4 COMPUTAÇÃO PARALELA

4.1 Unidade de Processamento Gráfico de Propósito Geral - GPGPU

Talvez um dos pilares modernos da ciência seja a computação. Os computadores, hoje, tornaram-se ferramentas indispensáveis para realizar descobertas científicas em diversas áreas do conhecimento bem como para impulsionar o desenvolvimento de novas tecnologias.

Grande parte deste impulso que tornou a computação uma ferramenta indispensável nos laboratórios modernos deveu-se ao significativo aumento do poder de processamento das CPUs (*central processing unit* ou unidade central de processamento) nas últimas décadas. Entretanto, apesar destes expressivos aumentos no poder de processamento, diversas aplicações demandam um poder de processamento muito maior do que um único processador sozinho é capaz de prover. Nestes casos, o uso de computação paralela torna-se a única escolha disponível.

Computação paralela é uma forma de computação em que o processamento de diversas instruções ocorre de forma simultânea. Naturalmente que um *hardware* com capacidade de realizar este processamento simultâneo se faz necessário, bem como o emprego de técnicas de programação que levem em consideração esta possibilidade de processamento concorrente. A Figura 4.1 mostra a diferença entre a execução sequencial e a execução em paralelo de uma fila de tarefas de processamento. Os quadros laranjas representam as tarefas destinadas ao processador.

Uma plataforma computacional que vem despertando o interesse dos pesquisadores da área de processamento paralelo é a GPGPU (*General Purpose Graphics Processing Unit* ou unidade de processamento gráfico de propósito geral). Trata-se de uma placa de vídeo com grande poder de processamento e que pode ser utilizada diretamente pelo programador para executar computação de propósito geral, e não apenas relacionado ao processamento gráfico.

A grande vantagem do uso de GPGPUs é a sua relação custo x benefício. Enquanto

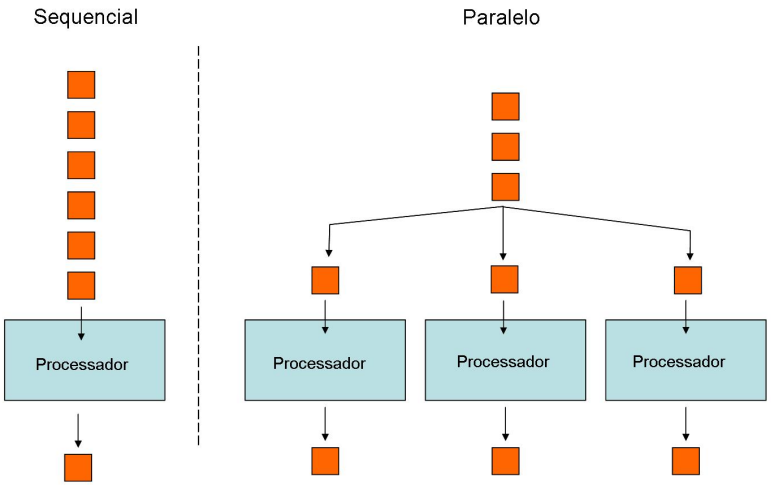


Figura 4.1: Processamento sequencial(à esquerda) e processamento paralelo(à direita) de uma mesma fila de tarefas de processamento.

cada núcleo de processamento de uma CPU moderna possui cerca de 7 unidades funcionais, das quais duas são utilizadas para computação com inteiros e uma com ponto flutuante, as GPGPUs atuais possuem 240 ou mais unidades funcionais (Figura 4.2). O grande número de elementos processadores destas placas permite que várias computações possam ser executadas simultaneamente. Uma única GPGPU equivale assim a quase uma dezena de CPUs, sendo provavelmente hoje o hardware com melhor relação custo x benefício disponível no mercado, tornando-a um ambiente computacional extremamente atraente para realizar computação paralela [8, 9, 10, 11, 12, 13, 14]. Entretanto, para tirar completo proveito de tal arquitetura, as aplicações precisam ter um padrão de computação bem conhecido na área de computação paralela, conhecido como paralelismo de dados[15]: uma mesma computação deve ser executada sobre cada um dos dados a serem processados de modo totalmente independente, de forma que para realizar um processamento não se faz necessária a utilização de resultados de processamentos obtidos com outros dados processados no mesmo passo.

Como este padrão ocorre justamente no problema que tratamos neste trabalho, decidimos empregar GPGPUs em seu processamento. Para um melhor entendimento do processo de paralelização do algoritmo, apresentamos nesta seção uma breve introdução desta plataforma computacional e das ferramentas que podem ser empregadas para o desenvolvimento das aplicações.

Figura 4.2: Componentes da CPU e da GPU. Retirado de [16].

4.2 CUDA (*Compute Unified Device Architecture*)

A tecnologia CUDA [17, 18] ou Arquitetura Unificada de Dispositivos de Computação é uma arquitetura de *hardware* e *software* criada pela NVIDIA. Baseada na extensão da linguagem de programação C, esta arquitetura provê acesso às instruções da GPU e controle da memória de vídeo para explorar o paralelismo encontrado nas placas gráficas atuais. CUDA permite implementar algoritmos que podem ser executados pelas GPUs das placas da série GeForce 8 e de suas sucessoras, GeForce 9, GeForce 200, Quadro e Tesla [19]. CUDA tem como características:

- Ser baseada na linguagem de programação C padrão;
- Possuir bibliotecas padrão para a Transformada de Fourier (FFT) e álgebra linear (BLAS);
- Troca de dados otimizada entre CPU e GPU;
- Interação com APIs gráficas (OpenGL e DirectX);
- Suporte a sistemas operacionais nas plataformas 32- e 64-bits, tais como Windows XP, Windows Vista, Linux e MacOS X;
- Desenvolvimento em baixo nível;
- Livre acesso a todo o espaço de endereçamento da memória da placa gráfica.

Algumas limitações da arquitetura são:

- Não há suporte para funções recursivas;

- As placas das séries 8 e 9 suportam apenas a precisão simples para a aritmética de ponto flutuante;
- Há alguns desvios do padrão IEEE-754;
- A largura de banda entre CPU e GPU pode se tornar um gargalo quando há a transferência de blocos de dados muito extensos;
- É uma arquitetura fechada, isto é, ela foi desenvolvida exclusivamente para placas gráficas da NVIDIA.

Para utilizar CUDA são necessárias três ferramentas: a) *driver* de vídeo, b) CUDA Toolkit e c) CUDA SDK. As duas últimas ferramentas oferecem todas as bibliotecas de CUDA, um guia de programação, o compilador NVCC (NVIDIA CUDA *Compiler*) e vários exemplos de aplicações utilizando os recursos de CUDA.

4.2.1 Arquitetura de uma placa de vídeo NVIDIA

Esta subseção visa mostrar os detalhes do *hardware* gráfico da NVIDIA, tomando como exemplo a arquitetura da GeForce 8800 GTX (Figura 4.3). As placas gráficas sucessoras seguem a mesma arquitetura, apenas com variação em algumas de suas características, como o número de processadores e sua frequência de operação, tamanho e tecnologia da memória, *hardware* de suporte à precisão dupla, etc.

A Figura 4.3 apresenta os elementos formadores do chip G80 e sua comunicação com a memória de vídeo e a CPU, referida na arquitetura CUDA como *Host*. Este *chip* tem seis barramentos de interface com a memória global, cada um de 64 bits e com seu próprio cache de memória L2.

O G80 dispõe de 128 *Scalar Processors* (SP) ou *Stream Processors*, localizados no centro da Figura 4.3. Os SPs são estruturas especializadas no processamento numérico, principalmente no que diz respeito às operações aritméticas de ponto flutuante. Pode-se dizer que estas unidades funcionam como 128 Unidades Lógico-Aritméticas (ULA), dentro de um único *chip*, com clock de 1.35 GHz cada uma. Os SPs operam segundo o modelo de computação SIMD (*Single Instruction Multiple Data*), isto é, executam a mesma instrução sobre diferentes elementos de dados em paralelo.

Na arquitetura do chip G80 ainda podem ser observados 16 *Streaming Multiprocessors* (SM). Cada SM é constituído por 8 SPs que por sua vez compartilham uma área de

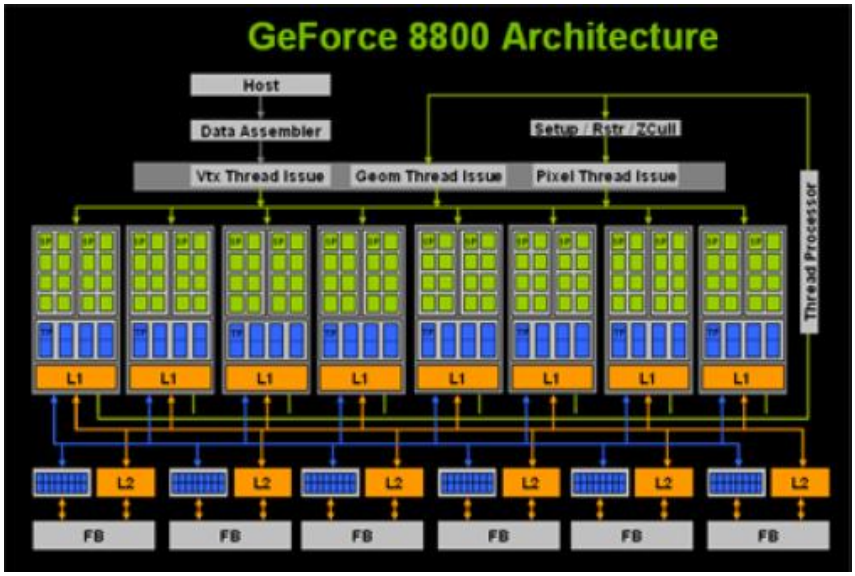


Figura 4.3: Interior do *chip* gráfico da GeForce 8800 GTX. Retirado de [20].

memória de tamanho máximo de 16 KB, 8 unidades de filtragem de texturas (os blocos azuis rotulados TF, textitTexture Filtering), 4 unidades de endereçamento de texturas (não ilustrada na Figura 4.3) e um cache de memória L1.

A placa gráfica conta com uma memória global de vídeo de 768 KB, uma memória somente para leitura, chamada memória constante, de 64 KB. A taxa de transferência de dados entre SPs e memória global de vídeo é de aproximadamente 57 GB/s. Já a taxa referente à transferência de dados da GPU para a CPU é de aproximadamente 950 MB/s. Quando a transferência ocorre na direção inversa, a taxa é de aproximadamente 1.3 GB/s. Por último, há um controlador de emissão de *threads* localizado topo da figura.

4.2.2 Modelo de Programação

Em uma aplicação GPGPU, uma função paralelizável é denominada *kernel*. Quando esta função é implementada em CUDA, ela utiliza uma hierarquia de *threads* definida pelo modelo de programação desta linguagem. Esta hierarquia está diretamente ligada à divisão do *hardware*.

A unidade básica da hierarquia proposta pela arquitetura CUDA é a *thread*. As *threads* desempenham a função de manipular os dados envolvidos no processamento do *kernel* pela GPU. Cada *thread* é executada por um SP. Elas estão agrupadas em blocos de *threads* (*blocks*). Cada bloco de *threads* está associado a um MP. Um conjunto de blocos de

threads forma um *grid*, que é a unidade máxima da hierarquia.

Os blocos podem ser representados como uma matriz, onde o acesso às *threads* é feito utilizando-se a palavra reservada *threadIdx*. Seguindo a hierarquia, cada bloco tem seu próprio índice dentro do *grid*, que também é representado por uma matriz, sendo recuperado através da palavra reservada *blockIdx*. Os índices são atribuídos de acordo com a ordem de escalonamento realizada pelo *hardware*. Como pode ser observado na Figura 4.3, no máximo 8 *threads* por bloco e no máximo 16 blocos são executados simultaneamente pela GPU.

A divisão hierárquica das *threads* também determina como é feito o acesso aos tipos de memória disponíveis na placa gráfica. Todos os blocos de um *grid* têm acesso à memória global e à memória constante. A memória compartilhada de cada MP só pode ser acessada por *threads* pertencentes ao mesmo bloco, sendo que elas não se comunicam com *threads* de blocos distintos. Cada *thread*, atribuída a um SP, acessa a) o seu conjunto de registradores, b) uma pequena memória local (de uso exclusivo de cada SP), c) a memória compartilhada pertencente ao bloco onde o SP se encontra e d) às demais memórias.

Dado o modelo de programação acima, é preciso passar a GPU a configuração escolhida de *grid*, juntamente com o *kernel* que será executado. Ao configurar o *grid* deve ser observada uma limitação com relação às suas dimensões. Um *grid* pode ser representado como uma matriz bidimensional de blocos. O número total de blocos multiplicado pela quantidade de blocos em cada dimensão não pode ultrapassar 65.536. O bloco, por sua vez, pode ser representado como uma matriz tridimensional de *threads*. O número total de *threads* encontrado ao se multiplicar a quantidade de *threads* em cada dimensão não pode ultrapassar 512.

Um *kernel* pode ser de três tipos definidos por CUDA: *host*, *global* ou *device*. O tipo *host* indica que o *kernel* será chamado e executado pela CPU. Este tipo de declaração equivale a uma função comum declarada em linguagem C. O tipo *global* indica que o *kernel* será invocado pela CPU e executado na GPU. Durante a chamada deste tipo de função é que o tamanho do *grid* é informado. Isto será melhor ilustrado no exemplo de código a seguir. O último tipo, *device*, especifica que o *kernel* será chamado e executado pela GPU.

O algoritmo 4.1 ilustra a adição de dois vetores utilizando CUDA. Basicamente a escrita do código segue os seguintes passos:

Algoritmo 4.1: Adição de dois vetores em paralelo usando CUDA.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
__global__ void add_arrays_gpu(float *in1,float *in2,
                               float *out,int Ntot){
    int idx=blockIdx.x*blockDim.x+threadIdx.x;
    if(idx < Ntot)
        out[idx]=in1[idx]+in2[idx];
}
void main(){
    /*pointers to host memory*/
        float *h_in1, *h_in2, *h_out;
        /*pointers to device memory*/
        float *d_in1, *d_in2, *d_out;
        int N=18;
        /*Allocate host arrays*/
        h_in1=(float*)malloc(N*sizeof(float));
        h_in2=(float*)malloc(N*sizeof(float));
        h_out=(float*)malloc(N*sizeof(float));
        /*Allocate device arrays*/
        cudaMalloc((void**)&d_in1,N*sizeof(float));
        cudaMalloc((void**)&d_in2,N*sizeof(float));
        cudaMalloc((void**)&d_out,N*sizeof(float));
        /*Initialize host arrays h_in1 and h_in2 here*/
        /*Copy data from host memory to device memory*/
        cudaMemcpy(d_in1,h_in1,N*sizeof(float),cudaMemcpyHostToDevice);
        cudaMemcpy(d_in2,h_in2,N*sizeof(float),cudaMemcpyHostToDevice);
        cudaMemcpy(d_out,h_out,N*sizeof(float),cudaMemcpyHostToDevice);
        /*Compute the execution configuration*/
        int block_size=8;
        dim3 dimBlock(block_size);
        dim3 dimGrid((N/dimBlock.x)+(!(N%dimBlock.x)?0:1));
        /*Add arrays h_in1 and h_in2, store result in h_out*/
        add_arrays_gpu<<<dimGrid,dimBlock>>>(d_in1,d_in2,d_out,N);
        /*Copy data from device memory to host memory*/
        cudaMemcpy(h_out,d_out,N*sizeof(float),cudaMemcpyDeviceToHost);
        /*Print h_out and free memory*/
}
```

1. Inclusão das bibliotecas necessárias;
2. Declaração de funções;
3. Iniciar o dispositivo;
4. Alocar memória na GPU;
5. Transferir os valores da CPU para a GPU;
6. Chamar o *kernel*;
7. Copiar os resultados de volta para a CPU;
8. Liberar a memória alocada;
9. Finalizar o dispositivo.

4.2.3 Utilizando Múltiplas GPUs Simultaneamente

Da mesma forma que diversas aplicações possuem demandas computacionais que não podem ser atendidas por uma única CPU, antevemos que a grande demanda computacional decorrente da complexidade dos modelos constituídos por dezenas ou centenas de milhões *spins* irá impor o uso de não uma, mas de várias GPGPUs concomitantemente, em um ambiente de agregados de computadores (*clusters* de computadores). Os agregados de computadores são formados por um grupo de computadores autônomos, interligados por uma rede rápida de comunicação.

Uma alternativa para utilizar múltiplas GPUs simultaneamente em um ambiente de agregados de computadores é empregar o modelo de troca de mensagens [21] como mecanismo para realizar a distribuição de dados entre as GPUs, bem como para acessar os resultados da computação. O modelo de troca de mensagens permite que dois ou mais processos se comuniquem através da cópia do dado de um espaço de memória do emissor para o do receptor. Geralmente este modelo é usado quando os processos não compartilham memória, como nos agregados de computadores. Desta forma, nesse modelo os computadores são tratados como uma coleção de processadores, cada um com espaço próprio de memória. Um processador tem acesso somente aos dados e instruções armazenados em sua memória local, o que não impede que qualquer processo possa se comunicar com todos os demais a qualquer tempo.

Um padrão popular para troca de mensagens é MPI (*Messagem Passing Interface*)[22]. Em MPI, destacam-se duas primitivas de comunicação: a comunicação ponto-a-ponto, e a comunicação coletiva. Na comunicação ponto-a-ponto, apenas dois computadores estão envolvidos no processo de comunicação, o processo que envia mensagens e o processo que recebe a mensagem. A comunicação coletiva é definida como um tipo de comunicação que envolve um grupo ou grupos de processos. Na comunicação coletiva, três ou mais processos estão envolvidos na comunicação, sendo que um deles envia a mensagem para os demais, ou um deles recebe múltiplas mensagens simultaneamente.

As principais funções MPI para comunicação são:

- MPI_Send: Envia mensagem para único destinatário. Trata-se de uma primitiva de comunicação ponto-a-ponto.
- MPI_Receive: Recebe mensagem de único remetente. Trata-se de uma primitiva de comunicação ponto-a-ponto.
- MPI_Barrier: barreira de sincronização entre todos os membros de um grupo. Os processos param até que todos cheguem naquele ponto. Trata-se de uma primitiva de sincronização implementada com o uso de mecanismos de comunicação coletiva.
- MPI_Bcast: Envio de uma mensagem de um membro do grupo para todos os demais membros deste. Trata-se de uma primitiva de comunicação coletiva.
- MPI_Reduce: operação de redução global, tal como soma, subtração, mínimo, máximo ou funções definidas pelo usuário. Nesse caso o resultado será acumulado somente no processo mestre. Trata-se de uma primitiva de comunicação coletiva.
- MPI_Allreduce: semelhante à operação MPI_Reduce, com a única diferença que o resultado final da operação é retornado para todos os processos que fazem parte do comunicador.

5 VERSÕES PARALELAS IMPLEMENTADAS

O algoritmo de Metropolis desempenha um papel importante como ferramenta de apoio na validação de um modelo teórico. No contexto das simulações físicas, esta é uma ferramenta amplamente empregada na resolução de problemas que envolvem sistemas de partículas atômicas, sendo perfeitamente aplicável à simulação dos *spins* magnéticos. No entanto, os recursos computacionais disponíveis às vezes são insuficientes para realizar a simulação destas estruturas, conforme a sua complexidade. O tamanho dos sistemas simulados e o número ideal de passos de Monte Carlo contribuem para aumentar o tempo de execução do algoritmo. Isto torna as simulações de alguns sistemas inviáveis, em decorrência dos tempo de simulação muito longos. Para resolver este problema, é comum a adoção de técnicas de computação paralela para garantir que os resultados de uma simulação possam ser produzidos num intervalo de tempo razoável.

Como foi apresentado no Capítulo 2, a representação físico-matemática dos *spins* se dá através da Equação 2.16. A maior parte da computação está concentrada no termo de interação dipolar. O algoritmo de Metrópolis, em sua forma sequencial, executa a cada passo de Monte Carlo a interação entre um *spin* do sistema em relação aos demais, para computar a contribuição energética de cada par formado, e este procedimento é realizado para todos os *spins* do sistema. Como discutido no Capítulo 3, essa forma de cálculo leva a execução sequencial do algoritmo de Metropolis ter complexidade $O(N^2)$, sendo N o número de *spins* simulado.

Neste capítulo apresentamos as duas abordagens utilizadas na paralelização dos algoritmos, visando uma redução em seus tempos de computação. Na primeira abordagem, apenas uma GPU é utilizada para realizar computações. Na segunda abordagem, múltiplas GPUs são empregadas nos cálculos.

Tanto o algoritmo proposto neste trabalho, **2xN**, quanto o algoritmo tradicional, chamado neste texto de **NxN**, foram paralelizados com uma GPU. O algoritmo 2xN também foi paralelizado utilizando múltiplas GPUs. Os códigos das versões paralelas para uma GPU são apresentados no Apêndice A.

5.1 Algoritmo NxN Paralelo

Na primeira tentativa para paralelizar a implementação do algoritmo NxN, a matriz de dados foi organizada de forma contígua na memória global da GPU, ou seja, foi armazenada como um vetor unidimensional. A GPU calculava somente a energia dipolar: o *kernel* era chamado pelo processador para calcular a interação dipolo-dipolo de um *spin* do sistema por vez, chamado de principal. Toda vez que o *kernel* era executado, cada *thread* criada para execução na GPU calculava a energia dipolar entre o *spin* principal e um outro *spin* do sistema, que estava armazenado na matriz localizada na memória global da GPU. Depois que a energia dipolar era calculada, o resultado era copiado de volta para a CPU. O processador calculava os outros termos da Equação 2.16 e completava a execução do código, incluindo a execução do algoritmo de Mersenne Twister.

No entanto, o desempenho desta primeira abordagem foi muito aquém do esperado. Dois fatores distintos contribuíram para o fraco desempenho. O primeiro motivo é que o tamanho da estrutura de dados com as informações de cada *spin* não favorecia um acesso otimizado das *threads* à memoria global da GPU. O *hardware* ficava impossibilitado de agrupar a maior quantidade de dados possíveis para realizar uma transferência única de memória, o que permitiria diminuir o número de requisições à memória. O segundo motivo é que a alocação e dealocação de memória da GPU, bem como a transferência de dados de e para o dispositivo, eram executados em cada passo de Monte Carlo, o que representou uma grande sobrecarga adicional (*overhead*) em termos de tempo de execução. Assim, para melhorar o desempenho, este código foi completamente reestruturado.

A primeira modificação que foi implementada está relacionada com o cálculo da energia do sistema. Enquanto na primeira abordagem a energia de um único *spin* era calculada por vez, nesta abordagem a energia de cada partícula e sua interação com todos os outros *spins* são calculados em paralelo. Nesta segunda abordagem, todas as energias apresentadas nas Equações 2.14, 2.13 e 2.15 são calculadas pela GPU, diferentemente da primeira abordagem, onde somente a energia do dipolo-dipolo era calculada na GPU. Após computar as energias de todos os *spins*, elas são atualizadas na memória global. No final da computação, aproveitando a forma como as energias estão armazenadas na memória global, um *kernel* de redução é executado, produzindo a soma das energias. Então a CPU obtém o total de energia da GPU e passa então e executar os passos finais do algoritmo de Metropolis.

Outra diferença importante entre ambas as abordagens é a forma como os dados são mapeados na memória da GPU. Na primeira abordagem, os dados foram completamente armazenados na memória global. Embora a memória global seja maior, ela é mais lenta do que as outras memórias disponíveis na GPU, como a memória compartilhada, por exemplo. No entanto, a memória compartilhada é menor do que a estrutura de dados utilizada. Assim a memória global foi usada juntamente com a memória compartilhada para armazenar os dados, utilizando a técnica de ***tilling***. Essa mesma técnica foi utilizada na implementação computacional do problema dos N-corpos[23]. Usando *tilling*, os dados são divididos em subconjuntos, de modo que cada *tile*(pedaço) se encaixe na memória compartilhada. No entanto, é importante mencionar que os cálculos realizados pelo *kernel* sobre estes *tiles* devem ser feito independentemente uns dos outros(Figura 5.1).

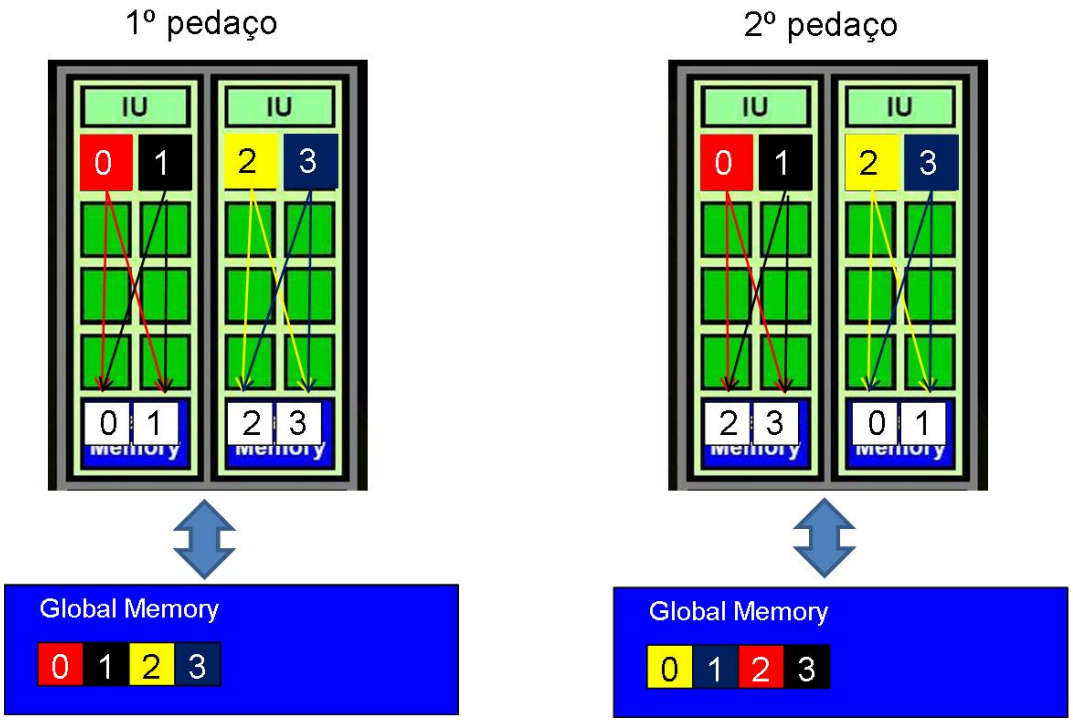


Figura 5.1: Exemplo mostrando as etapas de *tilling* na GPU.

A terceira modificação diz respeito à forma como o *hardware* da GPU é usado. Se o número de *threads* a ser criado é menor do que um determinado limiar, a forma como o cálculo é feito é modificada. Neste caso, duas ou mais *threads* ficam associadas a cada *spin* e colaboram para calcular a sua interação dipolar. As *threads* colaboram de uma forma simples: o *tile* é dividido entre as *threads*, então cada *thread* será responsável por calcular a interação dipolar entre seu *spin* e parte dos *spins* que compõem o *tile*. Por

exemplo, se o algoritmo decide criar duas *threads* por *spin*, então uma será responsável por calcular as interações entre seu *spin* e os *spins* que compõe a primeira metade do *tile*, enquanto a segunda *thread* será responsável pelo cálculo das interações do mesmo *spin* com os *spins* que formam a segunda metade do *tile*. Esta abordagem aumenta o uso da GPU porque mais *threads* são criadas, reduzindo, ao mesmo tempo, o cálculo total feito por uma única *thread*.

A modificação final na primeira abordagem foi a decomposição da matriz de dados em três vetores distintos do tipo *float*, contendo, respectivamente: a) a posição do *spin* no espaço 3D, b) a sua orientação, e c) sua energia. Esta modificação foi inspirada pelo mesmo trabalho que implementa o problema dos N-corpos em GPU[23]. O objetivo da decomposição é otimizar as requisições e transferências de memória, e também evitar conflitos durante o seu acesso. Esse objetivo é alcançado ao se reduzir a possibilidade de duas ou mais *threads* acessarem a mesma posição de memória concorrentemente e ao se melhorar o alinhamento da estrutura de dados na memória. O primeiro vetor foi declarado como *float4*, o segundo como *float3* e o último como *float*. No caso do primeiro vetor, três valores formam as coordenadas e o quarto valor é um índice definido de forma única. Esse índice é usado para evitar a computação da energia de longo alcance da partícula com ela própria.

5.1.1 Geração Automática da Configuração de Execução

Quando o *host* chama um *kernel*, ele deve especificar uma configuração de execução, o que significa definir o número de *threads* paralelas em um bloco e o número de blocos a serem utilizados durante a execução do *kernel* pelo dispositivo CUDA. O programador é responsável por fornecer essas informações. A escolha dos valores de configuração de execução tem um papel importante no desempenho da aplicação.

Na segunda tentativa para paralelizar a implementação do algoritmo NxN, a configuração de execução de um *kernel* foi gerada automaticamente em tempo de execução: o número de *threads* por bloco e o número de blocos são calculados com base no número de *spins* presentes no sistema. Esses valores são gerados de modo a se obter o melhor desempenho possível. Neste sentido, a meta é chegar a uma configuração com o maior número de *threads* por bloco. Para obter este número, alguns aspectos devem ser levados em consideração, tais como as características de *hardware* e da quantidade de

recursos disponíveis para cada *thread*.

O algoritmo começa consultando o dispositivo para obter as suas características. Algumas informações são então extraídas, tais como o número de multiprocessadores disponíveis. Em seguida, alguns valores são calculados, como *mnt*, o número mínimo de *threads* que devem ser criadas para garantir o uso de todos os processadores disponíveis na arquitetura GPGPU. Este valor é igual ao número máximo de *threads* por bloco vezes o número de multiprocessadores. O número máximo de *threads* por bloco é constante, igual a 256, porque este é o valor máximo que permite o lançamento de um *kernel* da implementação NxN. Na sequência, o número de *threads* que será usado durante a computação é calculado. Este valor é obtido em duas etapas. O primeiro passo considera que uma *thread* por *spin* será usada, enquanto a segunda etapa leva em conta o uso de múltiplas *threads* por *spin*. No primeiro passo, o *numero_de_threads* por bloco é definido como 1. Então, é verificado se o número de *spins* é primo: o algoritmo tenta encontrar o Maior Divisor Comum (MDC) entre 1 e a raíz quadrada do número de *spins*, uma vez que esse valor é suficiente para determinar se o número de *spins* é primo ou não. Se o número é primo, o algoritmo cai no pior caso e mantém o número de blocos igual a 1. Durante o cálculo do MDC, o quociente da divisão entre o número de *spins* e o divisor encontrado é armazenado. Em seguida, o algoritmo verifica se o quociente está no intervalo entre 1 e 256. Se isto ocorrer, o quociente é considerado um candidato a ser o *numero_de_threads* por bloco. Do contrário, o divisor é considerado um candidato. A segunda etapa avalia se a utilização de várias *threads* por *spin* é viável. Para isso, o algoritmo compara o número de *spins* com *mnt*. Se o número de *spins* é maior ou igual a *mnt*, o valor obtido na primeira etapa é mantido como *numero_de_threads* por bloco. Caso contrário, o algoritmo tenta organizar os *spins* em uma matriz bidimensional, onde x representa o número de *spins* por bloco, enquanto y representa o número de *threads* por *spin*. A idéia é tentar organizar as *threads* de maneira que as duas dimensões do bloco, x e y, reflitam o tamanho do *warp* e o número de *stream processors* disponível na máquina. A terceira dimensão, z, será igual a um. Se nenhum arranjo de x e y for encontrado, as dimensões do bloco e do *grid* são respectivamente iguais a (*numero_de_threads*, 1, 1) e (*numero_de_spins*/*numero_de_threads*, 1,1). Se houver um arranjo, as dimensões do bloco e do *grid* são respectivamente iguais a (x, y, 1) e (*numero_de_spins*/*numero_de_threads*, 1,1).

5.1.2 Execução do Kernel

O primeiro passo do algoritmo é decompor a matriz de dados em três vetores distintos: a) direção, b) posição e c) energia. Estes vetores são copiados para a memória global da GPU. Então a configuração para a execução do *kernel* é calculada usando o algoritmo descrito na subseção anterior. Em seguida, a CPU chama o *kernel*. Cada *thread* acessa um *spin* particular de acordo com a sua identificação única e copia seus valores de direção e posição na memória local. Quando todas as *threads* do *grid* terminam este passo, elas começam a calcular a interação dipolar: cada *thread* calcula a energia dipolo entre o *spin* armazenado na sua memória local e o subconjunto de *spins* armazenados no *tile* (ou parte deles, no caso de múltiplas *threads* por *spin*), que foi trazido da memória global para a memória compartilhada. Devido a maneira como os dados estão organizados, todas as transferências de memória são feitas sem o bloqueio das *threads*, isto é, cada uma delas está vinculada a uma posição da memória compartilhada onde elas lêem e escrevem dados e as posições atribuídas são distintas. O resultado obtido é adicionado ao resultado parcial armazenado em uma variável local. Este passo é repetido até que todas as *threads* tenham calculado a interação de seu *spin* local e todos os outros *spins* do sistema. Feito isso, cada *thread* calcula a energia ferromagnética e a interação com o campo externo. Estes valores são adicionados ao valor recém calculado da energia dipolar e o resultado final é armazenado na memória global. Terminado este passo, a CPU chama outro *kernel* que computa a redução do vetor de energias. A soma de todos os valores contidos nas posições do vetor de energias representa a nova energia do sistema.

5.2 Algoritmo 2xN Paralelo para uma GPU

A versão paralela do algoritmo 2xN segue a mesma estratégia de raciocínio adotada para a versão sequencial. O cálculo da energia inicial do sistema é feito pelo algoritmo NxN paralelo e, nas iterações seguintes, o algoritmo 2xN paralelo é utilizado.

Assim como na versão sequencial, o *kernel* do algoritmo 2xN paralelo precisa computar duas vezes as energias potenciais de interação: uma vez para efetuar os cálculos considerando os valores de orientação do *spin* escolhido sem a troca de orientação, e outra para efetuar os cálculos com os valores deste *spin* após a troca.

A implementação CUDA tem passos semelhantes ao algoritmo NxN. A estrutura de

dados e a configuração do *grid* permanecem a mesma para ambos os algoritmos. O algoritmo de configuração é o mesmo descrito na Subseção 5.1.2. A única diferença é que o algoritmo 2xN paralelo é executado sempre com uma *thread* por *spin* devido a simplicidade da implementação do *kernel*. Como cada *thread* terá de calcular somente duas interações, um *tile* será composto por dois elementos. Portanto, a criação de várias *threads* por *spin* se faz desnecessária.

Após a configuração do *grid* ser definida, cada *thread* continua acessando um *spin* em particular de acordo com a sua identificação única, copiando seus valores de orientação e posição. Neste ponto, acontece a primeira mudança no *kernel*: cada *thread* guarda também os valores de posição e orientação do *spin* com orientação trocada na memória local. Com isso, todas as *threads* passam a ter as informações necessárias para dar início à computação das energias. Desta forma, cada *thread* precisa realizar duas requisições à memória global durante a execução do *kernel*. A diminuição no número de requisições, aliado ao fato delas serem feitas com a maior otimização possível, minimiza a sobrecarga de computação. Esta sobrecarga aparece por conta dos acessos contínuos à memória global. Isso implica na eliminação do uso da memória compartilhada e faz com que o processamento seja mais rápido, mesmo nas situações em que GPU fica com processadores ociosos.

A CPU chama então o *kernel* do algoritmo, passando para as *threads* os parâmetros usuais e o valor da orientação do *spin* escolhido antes de ocorrer a sua troca. Cada *thread* calcula as energias da mesma forma que o algoritmo sequencial. Se o *spin* manipulado pela *thread* for diferente do *spin* que teve sua orientação trocada, a *thread* calcula a energia dipolar. Caso contrário, ela calcula as duas outras parcelas de energia.

Vale lembrar que as três parcelas de energia são calculadas duas vezes: o primeiro cálculo resulta na energia de interação entre o *spin* vinculado à *thread* e o *spin* antes da troca de orientação e o segundo resulta na energia de interação entre o *spin* vinculado à *thread* e o *spin* depois da troca de sua orientação. As duas energias computadas são subtraídas e o resultado é armazenado no vetor de energias.

5.3 Algoritmo 2xN Paralelo para Múltiplas GPUs

Este algoritmo visa a execução do algoritmo 2xN paralelo em ambientes distribuídos compostas por múltiplas GPUs[24]. O objetivo é usar o poder computacional de um conjunto de GPUs para simular um número de *spins* que seja maior do que a capacidade máxima de uma única GPU. Atualmente, o algoritmo 2xN paralelo é capaz de simular um sistema com até 16.581.375 *spins*, representando uma estrutura cúbica com tamanho de aresta igual a 255. Esta limitação se deve a incapacidade de se gerar uma configuração de *grid* cuja quantidade de blocos fique abaixo do limite estabelecido pela NVIDIA (conforme apresentado na Seção 4.2.2).

Para simular um sistema formado por uma estrutura cúbica de dimensões maiores do que uma GPU possa lidar, o algoritmo 2xN distribuído particiona o espaço desta estrutura e divide as porções de *spins* encontradas entre duas ou mais GPUs disponíveis, que realizam os cálculos. Uma estrutura de dados especial denominada *octree* [25] é responsável pelo processo de particionamento espacial. As GPUs estão interligadas por uma rede e se comunicam utilizando o padrão MPI. É necessário definir quantas máquinas formam a rede antes do início da execução. Um processo MPI é iniciado em cada uma destas máquinas. Um processo se encontra em duas categorias: mestre ou escravo. O processo mestre é atribuído ao primeiro computador encontrado na rede. Este processo coordena a execução do programa, podendo também participar da computação. Os processos escravos são aqueles controlados pelo mestre, ou seja, os computadores restantes sempre aguardam instruções do computador definido com o processo mestre.

Estando a rede devidamente configurada, o algoritmo inicia sua execução. O processo mestre tem a tarefa de receber todos os parâmetros necessários à simulação, iniciar a matriz de dados e enviar ambos por mensagem para os processos escravos. O processo mestre aguarda a confirmação de recebimento da mensagem por todos os processos antes de prosseguir para a próxima instrução. Quando os processos escravos confirmam o recebimento dos dados, todos os processos da rede iniciam o particionamento espacial através da *octree*, como mencionado anteriormente. Cada processo percorre o subespaço obtido pela *octree* e gera uma lista encadeada com os *spins* deste subespaço. Cada processo toma sua lista para executar o algoritmo de Metropolis sobre os *spins* pertencentes à lista. A construção da *octree* em cada máquina se faz necessária para evitar o problema de endereçamento de memória. Sempre que um processo requer um dado que não está

em sua memória local ele deve copiar este dado da máquina que o possui e armazená-lo localmente. A cópia é feita por troca de mensagem. Outra vantagem da construção da *octree* em cada máquina é evitar o reenvio de dados.

A próxima etapa consiste em executar o algoritmo 2xN paralelo sobre a lista de *spins* resultante. Cada GPU calcula a contribuição parcial de energia dos *spins* de sua lista em relação aos demais. Algumas modificações foram feitas para que a implementação em CUDA funcionasse corretamente. São elas:

- Cada processo, ao criar seus vetores de orientação, posição e energia, irá ordená-los de acordo com a sua lista de processamento, ou seja, os *spins* presentes na lista ficam no início dos vetores. Por exemplo, seja 10 o total de spins de uma simulação e a lista de processamento gerada informa ao processo que ele efetuará os cálculos sobre os *spins* 1,3 e 4. Estes *spins* ficarão nas três primeiras posições dos vetores manipulados pela GPU, enquanto os sete *spins* restantes ocuparão as demais posições. Essa abordagem facilita a determinação do término dos cálculos sobre os *spins* de uma lista.

- Como há mais de um processo em execução, a troca de orientação de um *spin* é realizada apenas no processo mestre, que envia a posição que o *spin* com orientação trocada ocupa em cada vetor, juntamente com a sua nova orientação, para os demais processos. Estes atualizam a nova orientação na memória de sua GPU.

- A energia parcial resultante das interações dos *spins* na lista de processamento é enviada ao processo mestre, que por sua vez soma estas energias parciais à sua energia. Dessa forma, a energia total (ou a variação de energia) é obtida.

6 RESULTADOS EXPERIMENTAIS

Neste capítulo são apresentados os resultados computacionais experimentais obtidos pela implementação do algoritmo 2xN, das suas versões paralelas (utilizando uma e múltiplas GPUs), bem como da versão paralela NxN implementada para uma GPU.

Como base para a implementação foi utilizado o simulador Monte Carlo Spins Engine (MCSE) [26]. O simulador implementa a versão sequencial NxN do algoritmo para cálculo da energia potencial de interação entre *spins*. Tal simulador possui uma interface gráfica que permite acompanhar a evolução do sistema a cada passo de Monte Carlo. Entretanto, por questões de desempenho, essa funcionalidade foi desativada nas implementações descritas ao longo deste capítulo.

6.1 Ambiente Experimental

As versões sequencial e paralela foram testadas em uma máquina configurada com sistema operacional Linux Ubuntu 9.04 64-bits, processador Intel Core2 Quad Q9550 2.83 GHz, com 4 GB de memória RAM. A placa de vídeo utilizada nesta máquina foi a NVIDIA GeForce 295 GTX, com 896 MB de memória global. A versão CUDA utilizada foi a 3.0 junto com o *driver* de vídeo versão 256.35.

Para os testes com a versão paralela com múltiplas GPUs, foi utilizado um conjunto de seis máquinas com a seguinte configuração: processador Intel Xeon E5620 2.4 GHz, com 12 GB de memória RAM, e executando o sistema operacional Linux Rocks 5.4 64-bits. As placas de vídeo utilizadas nestes testes foram a NVIDIA Tesla C1060, com 4 GB de memória global. A versão CUDA utilizada foi a 3.2, usada em conjunto com o *driver* de vídeo versão 260.19-29.

6.2 *Métricas de Medida de Desempenho*

Os algoritmos desenvolvidos foram testados em cenários de simulação selecionados previamente. Os cenários possuem um grupo de variáveis que definem as condições iniciais do sistema simulado. As variáveis que compõe os cenários de teste são listadas abaixo:

- Dimensão: a dimensão do sistema se refere ao seu tamanho nos eixos coordenados, x, y e z;
- Geometria: são as formas que a estrutura do sistema pode assumir.
- Parâmetros do Hamiltoniano: são os valores escolhidos para as constantes A, J e D;
- Número de passos: o número de passos de Monte Carlo realizados pelo algoritmo de Metropolis;
- Direção do campo magnético aplicado: são os valores de orientação do campo nos eixos coordenados;
- Temperatura: a temperatura na qual o sistema está submetido durante a simulação;
- Orientação dos *spins*: valor inicial da orientação nos eixos coordenados;
- Número de execuções: se refere ao número de vezes em que a simulação é executada com um cenário.

Os cenários variaram de acordo com o número total de *spins* do sistema, a geometria do objeto e o número de iterações. As demais variáveis receberam um valor que se manteve o mesmo em todos os cenários. Os valores convencionados para as variáveis listadas foram:

- Dimensão: $10 \times 10 \times 10$, $22 \times 22 \times 22$, $64 \times 64 \times 64$, $100 \times 100 \times 100$, $216 \times 216 \times 216$ e $255 \times 255 \times 255$;
- Geometria: cubo sólido, cilindro vazado e esfera;
- Parâmetros do Hamiltoniano: A = 1, J = 5 e D = 50;
- Direção do campo magnético aplicado: ao longo do eixo-z, coordenadas = (0,0,1);
- Número de passos: 5 mil e 100 mil;

- Temperatura: 0.5;
- Orientação dos *spins*: orientação = (-0.57735, -0.57735, -0.57735);
- Número de execuções: 5 vezes

Para cada cenário escolhido, foram computados a média de energia ao final do número de iterações e o tempo médio de execução dos algoritmos. Cada configuração foi executada 5 vezes para guarantir a corretude dos resultados: o desvio padrão percentual dos tempos de execução foi calculado, ficando sempre abaixo de 1%. Nesta seção apresentamos os resultados obtidos para configurações com 5 mil passos de Monte Carlo; os resultados para as configurações com 100.000 passos são apresentados no Anexo B.

6.2.1 Considerações sobre Geometria

Em computação gráfica, existem duas maneiras de se representar um objeto sólido tridimensional[27]:

- Por Bordo: o sólido é representado pela descrição do seu bordo;
- Por Volume: o sólido é representado pela descrição de seu interior.

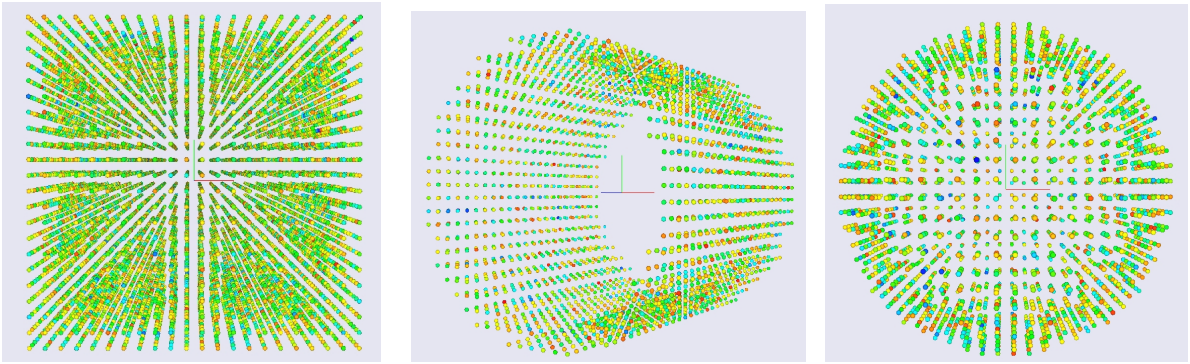
A geometria de um objeto 3D pode ser descrita matematicamente de forma paramétrica ou de forma implícita. Na forma paramétrica, os pontos pertencentes ao objeto são dados diretamente por uma coleção de mapeamentos ou parametrizações. Esses mapeamentos relacionam um espaço de parâmetros para a superfície do objeto de tal forma que há uma correspondência entre os pontos nestes dois espaços. Na forma implícita, os pontos pertencentes ao objeto é dado indiretamente por meio de uma função de classificação de pertinência de pontos. Esta função define a relação dos pontos no espaço ambiente com o objeto[28].O conjunto de pontos que pertencem a um objeto implícito é determinado pelo conjunto de pontos do espaço $S = f^{-1}(0) = X \in \mathbf{R}|f(X) = 0$, onde $f : \mathbf{R}^3 \rightarrow \mathbf{R}$ [28, 29].

Neste trabalho, utilizamos a representação volumétrica com descrição implícita para a construção do objeto simulado. O uso da descrição implícita facilita o acesso aos vizinhos de um ponto, já que a vizinhança é implícita. No entanto a discretização do suporte pode ser custosa.

Tabela 6.1: Número de total de *spins* para cada tipo de objeto.

Dimensão	Cubo	Cilindro	Esfera
10x10x10	1.000	280	272
22x22x22	10.648	2.816	2.600
64x64x64	262.144	69.888	62.904
100x100x100	1.000.000	262.400	239.080
216x216x216	10.077.696	2.633.472	2.404.504
255x255x255	16.581.375	4.345.200	3.956.846

A Figura 6.1 mostra, respectivamente, um cubo sólido(a), uma casca cilíndrica(b) e uma casca esférica(c). Cada ponto representa um *spin* atômico. Cada *spin* possui uma escala de coloração denotando a variação de sua temperatura. Temperaturas baixas são representadas pela cor azul enquanto temperaturas mais elevadas são representadas pela cor vermelha.



(a) Cubo Sólido (b) Casca Cilíndrica (c) Casca Esférica

Figura 6.1: Exemplo de objetos implícitos gerados pelo simulador.

Para a geometria do tipo cubo sólido, o número de *spins* é igual ao produto do número de *spins* em cada dimensão. Contudo deve ser observado que no caso do cilindro vazado e da esfera, o número de *spins* é muito menor que o produto das dimensões. A Tabela 6.1 apresenta o número de *spins* para cada tipo de objeto, considerando diferentes dimensões.

6.3 Comparação entre as Implementações Sequenciais dos Algoritmos

O objetivo desta seção é apresentar a comparação entre o algoritmo NxN e o novo algoritmo 2xN implementado. Assim, apresentamos na Tabela 6.2 os resultados de simulação sequencial obtidos para os algoritmos NxN tradicional e 2xN, com 5.000 passos

de Monte Carlo. Podemos observar o grande crescimento no tempo de computação do algoritmo tradicional, a medida que mais *spins* são acrescentados ao sistema.

Tabela 6.2: Tempo médio de execução dos algoritmos NxN e 2xN sequencial, em segundos.

Dimensão	NxN - Sequencial			2xN - Sequencial			Ganhos		
	Cubo	Cilindro	Esfera	Cubo	Cilindro	Esfera	Cubo	Cilindro	Esfera
10x10x10	96,97	13,76	14,02	0,2	0,09	0,09	484,85	152,83	155,77
22x22x22	10.754,82	1.476,32	1223,68	2,13	0,83	0,77	5.039,75	1.778,70	1.589,19
64x64x64	-	-	-	58,87	22,58	21,25	-	-	-

Devem ser destacadas as restrições impostas pela implementação do algoritmo NxN: dimensões com tamanhos maiores do que $22 \times 22 \times 22$ não puderam ser avaliadas em função do grande tempo de computação necessário para a sua execução.

Ao confrontar os resultados do novo algoritmo com os resultados obtidos para o algoritmo tradicional, pode-se observar que houve um decréscimo considerável nos tempos de execução. Os tempos obtidos para as configurações de dimensão $64 \times 64 \times 64$ são menores do que a menor configuração simulada pelo algoritmo tradicional. Isto significa que a simplificação do código correspondeu às expectativas de tornar a simulação mais rápida. Quando comparamos os resultados obtidos pelas implementações dos dois algoritmos com as mesmas geometrias e dimensões, observamos que a geometria cúbica teve seu tempo de execução reduzido em, respectivamente, 484 e 5.040 vezes. Para a geometria cilíndrica, a redução foi de 153 e 1.778 vezes, respectivamente. Para a esférica obteve-se uma redução de 155 e 1589 vezes, respectivamente. A redução foi maior para a geometria em forma de cubo pelo fato dela estar completamente preenchida, contendo assim um número muito maior de *spins* do que as demais geometrias. Assim, quanto mais *spins*, observamos que maior é o ganho obtido.

6.4 Comparação entre as Implementações Paralelas e Sequenciais dos Algoritmos

6.4.1 Implementação NxN

Os resultados para a execução da versão paralela NxN são apresentados na Tabela 6.3. Devemos observar um aspecto importante que diz respeito ao grande aumento no número de *spins* que puderam ser simulados. Enquanto na versão sequencial o número máximo de

Tabela 6.3: Tempo médio de execução dos algoritmos NxN sequencial e NxN paralelo, em segundos.

Dimensão	NxN - Sequencial			NxN - Paralelo			Ganhos		
	Cubo	Cilindro	Esfera	Cubo	Cilindro	Esfera	Cubo	Cilindro	Esfera
10x10x10	96,97	13,76	14,02	4,45	1,91	1,82	21,82	7,19	7,71
22x22x22	10.754,82	1.476,32	1.223,68	92,14	14,89	13,90	116,72	99,17	88,04
64x64x64	-	-	-	54.861,13	4.197,53	4.621,25	-	-	-
100x100x100	-	-	-	759.083,01	54.924,22	45.106,05	-	-	-

spins simulados foi igual à 10.648 (na configuração cubo sólido, de dimensões $22 \times 22 \times 22$), na versão paralela esse número pode aumentar para 1.000.000 (na configuração cubo sólido, de dimensões $100 \times 100 \times 100$), um aumento de quase 94 vezes.

As acelerações (*speedups*) [21] só puderam ser calculadas para as configurações até $22 \times 22 \times 22$, visto que os tempos sequenciais só estão disponíveis para tais configurações (conforme apresentado na Tabela 6.2). A maior aceleração foi de 117 vezes para a configuração cubo de tamanho $22 \times 22 \times 22$. Novamente, quanto maior a quantidade de *spins* do sistema, maior a aceleração obtida.

6.4.2 Implementação 2xN

A Tabela 6.4 apresenta os resultados das execuções para a versão paralela com uma GPU do algoritmo 2xN. Devem ser destacados dois apectos. O primeiro deles diz respeito ao tamanho dos sistemas sendo simulados, que chegam ao simular 16.581.375 de *spins*. Esse número só pode ser simulado em virtude da redução do tempo de computação proporcionado pelo uso da GPU. Por outro lado, uma restrição imposta pela própria GPU impedem que sistemas maiores sejam simulados: o limite máximo para criação de blocos na GPU foi alcançado.

O segundo aspecto a ser destacado foram as acelerações obtidas. Novamente as acelerações só puderam ser calculadas para as configurações onde foi possível obter os tempos de execução sequencial. Em especial, quando a configuração utilizada é muito pequena (por exemplo, $10 \times 10 \times 10$), podemos observar um *slowdown*, ou seja, um aumento no tempo de execução das configurações paralelas em relação as configurações sequenciais. A medida que sistemas maiores são simulados, os *slowdowns* dão lugar as acelerações, que tendem a crescer a medida que sistemas maiores são simulados. A maior aceleração obtida foi igual a 13.7, novamente na situação em que o maior número de *spins* que foi simulado ($22 \times 22 \times 22$). Tanto os resultados apresentados na Tabela 6.3 quanto na Tabela 6.4

Tabela 6.4: Tempo médio de execução dos algoritmos 2xN sequencial e 2xN paralelo, em segundos.

Dimensão	2xN - Sequencial			2xN - Paralelo			Ganhos		
	Cubo	Cilindro	Esfera	Cubo	Cilindro	Esfera	Cubo	Cilindro	Esfera
10x10x10	0,2	0,09	0,09	0,84	1,04	1,03	0,24	0,09	0,09
22x22x22	2,13	0,83	0,77	0,97	1,02	1,03	2,20	0,81	0,74
64x64x64	58,87	22,58	21,25	4,29	2,04	1,95	13,74	11,04	10,92
100x100x100	-	-	-	13,46	4,33	4,18	-	-	-
216x216x216	-	-	-	117,44	31,88	30,65	-	-	-
255x255x255	-	-	-	200,92	54,10	-	-	-	-

indicam que maiores acelerações poderiam ser obtidas se fosse possível obter tempos de execução sequencial para configurações maiores.

Quando comparamos a contribuição total das implementações apresentadas neste trabalho para o cálculo da energia potencial de interação entre *spins*, chegamos a valores respeitáveis. Por exemplo, ao se comparar os valores da Tabela 6.2 com os valores da Tabela 6.4, chega-se a uma redução no tempo total de computação igual à 11.087 vezes (de $10.754,82$ segundos para $0,97$ segundos). Da mesma forma, quando se analisa o tamanho dos sistemas simulados, verifica-se que as implementações apresentadas neste trabalho permitiram que simulações 1.557 vezes maiores fossem executadas (de 10.648 de *spins* para 16.581.375 de *spins*).

6.5 Comparação entre a Versão com uma GPU e a Versão com Múltiplas GPUs

Esta seção apresenta os resultados da execução da versão 2xN do algoritmo, quando executada com múltiplas GPUs. Os resultados serão comparados com a mesma versão do algoritmo, mas utilizando apenas uma única GPU.

Como o *hardware* utilizado nos testes com múltiplas GPUs difere do usado nos testes apresentados até então, a primeira medida tomada para garantir a justiça nos testes foi re-executar a versão sequencial 2xN no novo *hardware*. Os resultados são apresentados na Tabela 6.5. Nesta tabela apenas a configuração com geometria cúbica foi avaliada, visto que esta é a geometria que apresentou, nos testes anteriores, as maiores demandas computacionais.

Os valores da Tabela 6.4 não podem ser diretamente comparados com os valores apresentados na Tabela 6.5. Essa comparação poderia ser de interessante, por exemplo,

Tabela 6.5: Tempo médio de execução do algoritmo 2xN paralelo, em segundos, quando executado na Tesla C1060.

Dimensão	2xN - Uma GPU	2xN - Multi-GPU		
	Cubo	Cubo		
		2 GPUs	4 GPUs	6 GPUs
22x22x22	3,15	45,3	63	58,7
64x64x64	16,9	65	67,7	71,7
100x100x100	159,3	210,3	146	211,3

para avaliar que placa tem melhor desempenho para esta aplicação, a 295 GTX ou a Tesla C1060. A razão pela qual a comparação não pode ser feita é simples: a forma com que o tempo foi medido em cada uma delas diferiu. No caso da Tabela 6.4 utilizamos o relógio da GPU. Já na Tabela 6.5 foi usado o tempo real medido pelo sistema operacional com o comando *time*, visto que as comparações que serão posteriormente apresentadas nesta seção, e que incluem os custos com as chamadas MPI, requerem que o tempo gasto com operações de entrada e saída sejam também levados em consideração, o que não pode ser computado com o relógio da GPU.

Foram realizadas execuções com 2, 4 e 6 GPUs. Podemos observar que todas as configurações, com exceção da configuração $100 \times 100 \times 100$ em 4 GPUs, apresentaram *slowdowns*. A configuração $100 \times 100 \times 100$ em 4 GPUs obteve uma pequena aceleração, aproximadamente igual a 1.1.

Para melhor compreender os resultados, foram efetuadas instrumentações automáticas no código com a ferramenta VampirTrace[30]. As instrumentações permitiram-nos verificar que o custo da troca de mensagens, efetuada a cada passo de Monte Carlo, é o responsável pelas desacelerações observadas nas configurações que apresentaram *slowdown* na Tabela 6.5, chegando a representar 52% do tempo total de computação, conforme ilustrado na Figura 6.2. Nesta figura, uma única operação MPI, MPI_Bcast, realizada pelo processo 2 dura 163.4s. Com a instrumentação, o mesmo código demora 316.5s para executar.

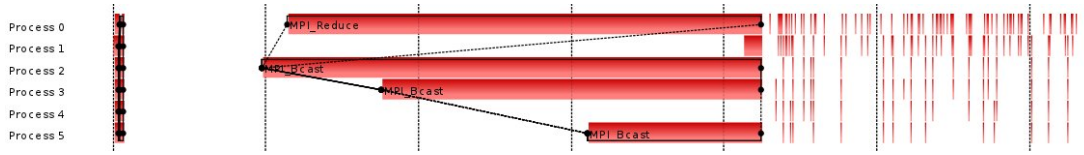


Figura 6.2: Linha do tempo para a execução da configuração 100x100x100 com 6 GPUs.

6.6 Comparação entre os valores de energia obtidos

Esta seção apresenta os valores de energia obtidos pelas versões paralelas e sequenciais dos algoritmos NxN e 2xN. As energias foram obtidas durante a simulação com 5.000 passos de Monte Carlo da geometria cúbica de dimensões $22 \times 22 \times 22$. Foram medidos o valor total de energia e a energia média do sistema a cada 500 passos de Monte Carlo, cujos valores são apresentados respectivamente nas Figuras 6.3 e 6.4.

Um aspecto importante a ser resaltado nas Figuras 6.3 e 6.4 é que as energias mensuradas nas quatro versões apresentam particamente os mesmos valores, com uma pequena diferença sendo observada na terceira casa decimal. Esta pequena diferença decorre da menor precisão numérica dos cálculos realizados pela GPU. Ainda assim podemos considerar que os algoritmos paralelos determinaram os valores de energia do sistema de modo corretamento.

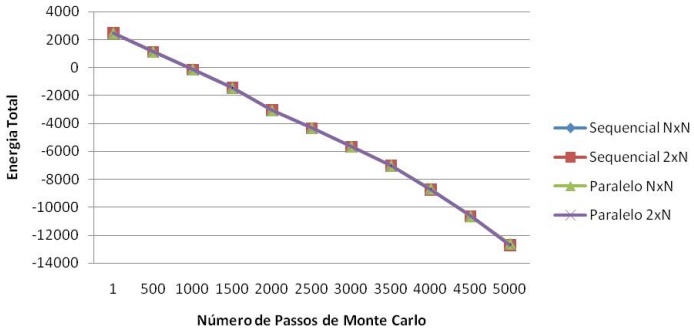


Figura 6.3: Energia total do sistema ao longo da simulação.

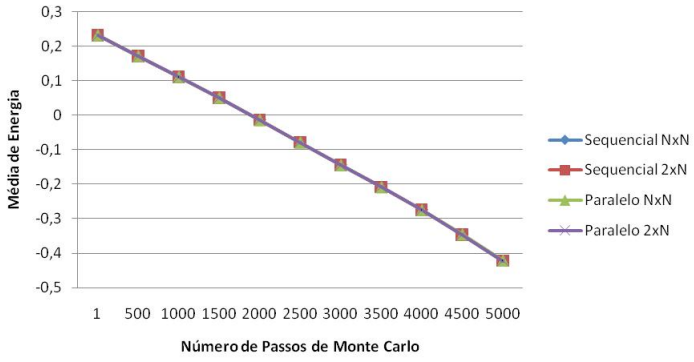


Figura 6.4: Média de energia do sistema ao longo da simulação.

7 TRABALHOS CORRELATOS

Várias propostas para reduzir o tempo total em simulações de Monte Carlo podem ser encontradas na literatura. Uma abordagem baseada em flipagem de grupamentos de *spins* foi introduzido por Swendsen e Wang [31]. Recentemente, Fukui e Todo [32] obtiveram um resultado interessante ao desenvolver um algoritmo $O(N)$ usando essa idéia.

Uma versão paralela do método de Monte Carlo com o modelo de *spin* de Ising[33] foi proposto por Santos *et al.* [34]. As diferenças do modelo proposto por Santos do modelo proposto neste trabalho são: a) a utilização de um espaço bidimensional, e b) a utilização de Ising[33]. Neste trabalho foram utilizados modelos mais complexos: o modelo de Heisenberg foi implementado em um espaço tridimensional. No modelo de *spin* de Ising o *spin* pode adotar apenas dois sentidos: ± 1. O modelo de Heisenberg utilizado neste trabalho é muito menos restritivo e mais realista, visto que o *spin* pode adotar qualquer direção nas coordenadas x, y e z.

Tomov *et al.* [35] desenvolveram uma versão GPU baseada no método de Monte Carlo utilizando o modelo de Ising para simulações ferromagnéticos. Novamente trata-se de um modelo computacionalmente mais simples, visto que o *spin* pode adotar apenas dois sentidos. Adicionalmente, no trabalho de Tomov não utiliza a interação de longo alcance, enquanto o presente trabalho utiliza essa interação. Embora Tomov *et al.* tenham implementado um modelo mais simples, eles não obtiveram uma boa aceleração: foi reportado em seu trabalho um aumento de velocidade de três vezes quando se usa sua versão paralela em GPU.

Uma outra versão GPU interessante para o modelo de Ising foi proposto por Preis *et al.* [36]. Neste trabalho, foram realizadas simulações em sistemas 2D e 3D. Os autores reportam resultados de 60 e 35 vezes mais rápidos, respectivamente, quando comparados com a versão que executa em CPU. Contudo, neste trabalho também não se utiliza o fator de longo alcance. Os resultados apresentados no presente trabalho demonstram que as distintas versões apresentadas foram mais efetivas na melhoria do desempenho da aplicação, mesmo utilizando um modelo mais complexo do que o implementado por Preis *et al.*.

Weigel [37] apresentou uma abordagem do modelo de Ising semelhante a de Preis,

implementando otimizações no acesso a memória compartilhada. Este algoritmo é 5 vezes mais rápido do que o algoritmo implementado na GPU por Preis e até 100 vezes mais rápido do que o algoritmo sequencial.

Block *et al.* também propuseram uma versão multi-GPU [38] para o modelo de Ising usando MPI. Este trabalho é uma expansão do trabalho de Preis e apresenta uma divisão espacial semelhante a apresentada no algoritmo 2xN distribuído. Block *et al.* conseguiram simular sistemas bidimensionais da ordem de 8×10^{10} *spins*. Isto não era possível no trabalho de Preis devido à limitação do tamanho da memória de uma única GPU para simular tais sistemas. Na presente dissertação foi possível simular sistemas com até 16 milhões de *spins*.

8 CONCLUSÕES

Ao longo deste trabalho foi apresentado um novo algoritmo para reduzir o custo computacional para a realização do cálculo da energia potencial de interação entre *spins*. Este novo algoritmo, denominado **2xN**, é assim chamado pela forma como são realizados os cálculos do ΔE, ou seja, da diferença entre a energia dos dipolos formados com o *spin* cuja orientação foi alterada e a energia dos dipolos formados com este mesmo *spin* antes da troca de orientação. Tais cálculos são realizados duas vezes, e apenas entre o *spin* que foi escolhido para ter seu valor de orientação alterado e os demais *spins* que formam o sistema.

Em termos de complexidade computacional, pode-se dizer que, excluído o primeiro passo de Monte Carlo, o algoritmo 2xN apresenta complexidade igual a O(N), onde N é o número de *spins* que constituem o sistema. Entretanto, o primeiro passo de Monte Carlo continua tendo custo da ordem de O(N^2), visto que o algoritmo tradicional precisa ser utilizado para que seja calculada a energia inicial do sistema.

Ainda com a necessidade de se utilizar o algoritmo tradicional no primeiro passo, o algortimo proposto apresenta uma forte redução no tempo total de computação, permitindo ganhos de desempenho até 1.160 vezes, quando comparado ao algoritmo tradicional NxN. Ganhos adicionais de desempenho foram obtidos com o auxílio de GPGPUs: neste caso, os ganhos apresentados neste trabalho foram da superiores a 11.000 vezes. Adicionalmente, o uso conjunto do novo algoritmo e da plataforma para computação de alto desempenho permitem que sistemas com mais de 16 milhões de *spins* sejam simulados, muito maiores que os sistemas simulados até então com o algoritmo NxN, na casa de 50.000 spins.

Deve-se ainda destacar, como contribuição deste trabalho, a proposição da geração automática da configuração de execução de um *kernel* CUDA. Para esta finalidade, o número de *spins* no sistema, bem como a quantidade total de memória utilizada por cada *thread*, são levados em consideração para o cálculo da configuração de execução. A geração da configuração automática de execução do *kernel* pode, no entanto, ser refinada para a obtenção de um melhor desempenho. Sugere-se, como trabalho futuro, um estudo detalhado dos impactos no desempenho quando distintas configurações são utilizadas.

Por fim, como contribuições secundárias deste trabalho, destacamos a implementação da versão sequencial do algoritmo 2xN, bem como a implementação paralela deste e do algoritmo NxN na plataforma CUDA. Ainda foi proposta e implementada uma versão para múltiplas GPUs do algoritmo 2xN, empregando neste caso também o protocolo MPI para a comunicação. A avaliação de desempenho realizada através da instrumentação do código da aplicação proporcionou um entendimento detalhado das causas para o baixo desempenho desta última versão e certamente permitirão, em um futuro próximo, a modificação dos algoritmos para que ganhos maiores de desempenho possam ser obtidos. Assim, como trabalho futuro, propõe-se uma maior investigação de alternativas para reduzir a necessidade de trocas de mensagens na versão com múltiplas GPUs.

A implementação 2xN para GPU possui hoje uma limitação em relação a quantidade de *spins* que podem ser simulados. Essa restrição se deve ao fato da configuração utilizada ter chegado ao limite no número de blocos que a GPU pode simular. Como trabalho futuro, propõe-se a solução desta limitação pelo uso de múltiplos *spins* por *thread*, ou seja, que uma única *thread* calcule os termos da equação de interação dipolar para vários *spins*, e não para apenas um, como é feito hoje.

Por fim, como trabalho futuro, sugere-se uma otimização no código *assembly* gerado pelo compilador nvcc, de modo que instruções desnecessárias por ele geradas sejam eliminadas[39].

APÊNDICE A - Códigos dos Algoritmos

Algoritmo A.1: Algoritmo tradicional para a computação das energias.

```
float computeEnergyNxN(float A, float J, float D, VECTOR3 externalField,
    VECTOR3 magnetization) {

    unsigned int i, j, k;// índices nos eixos x, y e z respectivamente
    float totalenergy = 0.0;

     for(k = 0; k < length_Z; k++)
       for(i = 0; i < length_X; i++)
         for(j = 0; j < length_Y; j++) {
           unsigned int ind = PARTICLEADDRESS(i, j, k); //calcula a posição
               do spin na matriz de dados
           matrix[ind].energy = (A*0.5) * computeDipoleDipoleEnergy(i, j, k)
               - J * computeMagnecticFactor(i, j, k) - (D * matrix[ind].spin
               * externalField);
           totalenergy += matrix[ind].energy;
           /*atualiza magnetização*/
         }

     return totalenergy;
}
```

Algoritmo A.2: Algoritmo 2xN sequencial para a computação das energias.

```
float computeEnergy2xN(float A, float J, float D, VECTOR3 externalField,
    VECTOR3 magnetization) {

     unsigned int i, j, k;
     float totalenergy = 0.0;
     //indexFlipped contém a posição em cada eixo coordenadodo spin com
         orientação trocada
     unsigned int indFlipped = PARTICLEADDRESS(indexFlipped[0], indexFlipped
         [1], indexFlipped[2]);
```

```
    for(k = 0; k < length_Z; k++)
      for(i = 0; i < length_X; i++)
        for(j = 0; j < length_Y; j++) {
          unsigned int ind = PARTICLEADDRESS(i, j, k);
          matrix[ind].energy = A * computeDipoleDipoleEnergy2xN(i, j, k,
              indexFlipped[0], indexFlipped[1], indexFlipped[2]);
          if(ind == indFlipped){
                matrix[ind].energy = ((-(2 * J) * computeMagnecticFactor(i
                    , j, k, matrix[ind].spin)- (D * matrix[ind].spin *
                    externalField))-(-(2 * J) * computeMagnecticFactor(i, j
                    , k, flippedSpin)- (D * flippedSpin * externalField)));
            }
          totalenergy += matrix[ind].energy;
          /*atualiza magnetização*/
        }

    return totalenergy;
}
```

Algoritmo A.3: Algoritmo NxN paralelo para a computação das energias.

```
template<bool multithreadSpins>
__global__ void
integrateSpins(float A, float J, float D, unsigned int length_X, unsigned
    int length_Y, unsigned int length_Z, float4* spinsArray,
                float* energiesArray, float4* positionsArray, int numSpins,
                    float3 externalField)
{

    unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;

    float4 currentSpin         = spinsArray[index];
    float4 currentSpinPosition = positionsArray[index];

    float energy = 0.0; energy = (0.5 * A) * computeDDEnergy<
        multithreadSpins>(currentSpin, currentSpinPosition, spinsArray,
        positionsArray, numSpins);
    __syncthreads();
```

```
15      if(threadIdx.y == 0){

            //Calcula a influencia do campo externo
            float dotCURSPIN_EXTFLD = ((currentSpin.x * externalField.x)+(
                currentSpin.y * externalField.y)+(currentSpin.z * externalField.
                z));

20          energy = energy - ((J * computeMagnecticFactor(currentSpin,
                currentSpinPosition, nX, nY, nZ, spinsArray, positionsArray))+(D
                * dotCURSPIN_EXTFLD));
            // Armazena a contribuicao de energia do spin no vetor de energias
            energiesArray[index] = energy;
        }
    }
```

Algoritmo A.4: Algoritmo 2xN paralelo para a computação das energias.

```
__global__ void
integrateSpins2xN(float A, float J, float D, unsigned int length_X,
    unsigned int length_Y, unsigned int length_Z, float4* spinsArray,
                float* energiesArray, float4* positionsArray, int numSpins,
                    float3 externalField, float4 oldOrientation, unsigned int
                     idxFlipped)
{
5
    unsigned int index = blockIdx.x *blockDim.x + threadIdx.x;

    float4 currentSpin         = spinsArray[index];
    float4 currentSpinPosition = positionsArray[index];
10  float4 flippedSpin         = spinsArray[idxFlipped];
    float4 flippedSpinPosition = positionsArray[idxFlipped];

    float energyOld = 0.0; energyOld = A * dipoleDipoleInteraction(
        energyOld, oldOrientation, flippedSpinPosition, currentSpin,
        currentSpinPosition);
    float energyNew = 0.0; energyNew = A * dipoleDipoleInteraction(
        energyNew, flippedSpin, flippedSpinPosition, currentSpin,
        currentSpinPosition);

15

    __syncthreads();
```

```
    if(currentSpin.w == flippedSpin.w){

        energyOld = 0.0; energyNew = 0.0;

        //Calcula a influencia do campo externo
        float dotCURSPIN_EXTFLD_OLD = ((oldOrientation.x * eF0)+(
            oldOrientation.y * eF1)+(oldOrientation.z * eF2));

        energyOld = energyOld - ((2 * J * computeMagnecticFactor(
            oldOrientation, currentSpinPosition, nX, nY, nZ, spinsArray,
            positionsArray))+(D * dotCURSPIN_EXTFLD_OLD));
        //Calcula a influencia do campo externo
        float dotCURSPIN_EXTFLD_NEW = ((currentSpin.x * externalField.x)+(
            currentSpin.y * externalField.y)+(currentSpin.z * externalField.
            z));
        energyNew = energyNew - ((2 * J * computeMagnecticFactor(
            currentSpin, currentSpinPosition, nX, nY, nZ, spinsArray,
            positionsArray))+(D * dotCURSPIN_EXTFLD_NEW));

    }
       __syncthreads();
    // Armazena a contribuicao de energia do spin no vetor de energias
     energiesArray[index] = energyNew - energyOld;


}
```

APÊNDICE B - Outros Resultados Experimentais

Os resultados abaixo foram obtidos para simulações com 100.000 passos de Monte Carlo.

Tabela B.1: Tempo médio de execução dos algoritmos NxN e 2xN sequencial, em segundos.

Dimensão	NxN - Sequencial			2xN - Sequencial			Ganhos		
	Cubo	Cilindro	Esfera	Cubo	Cilindro	Esfera	Cubo	Cilindro	Esfera
10x10x10	1.941,5	294,65	266,65	4,08	1,88	1,82	23,76	156,73	146,91
22x22x22	215.166,92	29.546,88	23.666,87	42,73	16,56	15,41	5.035,74	1.784,77	1.536,31
64x64x64	-	-	-	1.071,1	449,19	424,88	-	-	-

Tabela B.2: Tempo médio de execução dos algoritmos NxN sequencial e NxN paralelo, em segundos.

Dimensão	NxN - Sequencial			NxN - Paralelo			Ganhos		
	Cubo	Cilindro	Esfera	Cubo	Cilindro	Esfera	Cubo	Cilindro	Esfera
10x10x10	1.941,5	294,65	266,65	89,69	37,46	35,15	21,65	7,86	7,59
22x22x22	215.166,92	29.546,88	23.666,87	1.843,38	295,68	275,57	116,72	99,93	85,88

Tabela B.3: Tempo médio de execução dos algoritmos 2xN sequencial e 2xN paralelo, em segundos.

Dimensão	2xN - Sequencial			2xN - Paralelo			Ganhos		
	Cubo	Cilindro	Esfera	Cubo	Cilindro	Esfera	Cubo	Cilindro	Esfera
10x10x10	4,08	1,88	1,82	17,59	19,91	20,07	0,23	0,09	0,09
22x22x22	42,73	16,56	15,41	19,94	20,31	20,33	2,14	0,82	0,76
64x64x64	1.071,05	449,19	424,88	85,74	40,86	38,10	12,49	10,99	11,15
100x100x100	-	-	-	270,02	-	-	-	-	-
216x216x216	-	-	-	2.351,21	-	-	-	-	-
255x255x255	-	-	-	4.009,09	-	-	-	-	-

REFERÊNCIAS

[1] HEISENBERG, W., "Zur Theorie des Ferromagnetismus", *Z. Phys*, v. 49, pp. 619–636, 1928.

[2] TIPLER, P. A., *Física*. 4th ed., v. 3. Editora LTC, 2000.

[3] ROCHA, W. R., "Interações Intermoleculares", *Cadernos Temáticos de Química Nova na Escola*, v. 4, pp. 31–36, 2001.

[4] METROPOLIS, N., ROSENBLUTH, A. W., ROSENBLUTH, M. N., TELLER, A. H., TELLER, E., "Equation of State Calculations by Fast Computing Machines", *Journal of Chemical Physics*, v. 21, pp. 1087–1092, 1953.

[5] CASTRO, L. L., *Simulação Monte Carlo de Fluidos Magnéticos*, Master's Thesis, Universidade de Brasília, 2005.

[6] DOOB, J. L., *Stochastic Processes*. John Wiley and Sons: New York, 1990.

[7] MATSUMOTO, M., NISHIMURA, T., "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator", *ACM Trans. Model. Comput. Simul.*, v. 8, n. 1, pp. 3–30, 1998.

[8] CHE, S., BOYER, M., MENG, J., TARJAN, D., SHEAFFER, J. W., SKADRON, K., "A performance study of general-purpose applications on graphics processors using CUDA", *J. Parallel Distrib. Comput.*, v. 68, pp. 1370–1380, October 2008.

[9] ANDERSON, J. A., LORENZ, C. D., TRAVESSET, A., "General purpose molecular dynamics simulations fully implemented on graphics processing units", *J. Comput. Phys.*, v. 227, pp. 5342–5359, May 2008.

[10] KIPFER, P., SEGAL, M., WESTERMANN, R., "UberFlow: a GPU-based particle engine". In: *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pp. 115–122, ACM Press: New York, NY, USA, 2004.

[11] YANG, J., WANG, Y., CHEN, Y., "GPU accelerated molecular dynamics simulation of thermal conductivities", *J. Comput. Phys.*, v. 221, n. 2, pp. 799–804, 2007.

[12] GEORGII, J., ECHTLER, F., WESTERMANN, R., "Interactive Simulation of Deformable Bodies on GPUs". In: *Proceedings of Simulation and Visualisation 2005*, pp. 247–258, 2005.

[13] BOLZ, J., FARMER, I., GRINSPUN, E., SCHRÖODER, P., "Sparse matrix solvers on the GPU: conjugate gradients and multigrid", *ACM Trans. Graph.*, v. 22, pp. 917–924, July 2003.

[14] GOVINDARAJU, N., GRAY, J., KUMAR, R., MANOCHA, D., "GPUTeraSort: high performance graphics co-processor sorting for large database management". In: *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, *SIGMOD '06*, pp. 325–336, ACM: New York, NY, USA, 2006.

[15] MATTSON, T. G., SANDERS, B. A., MASSINGILL, B. L., *Patterns for Parallel Programming*. Addison Wesley: Westford, 2005.

[16] NVIDIA, *NVIDIA CUDA Programming Guide*, Tech. rep., NVIDIA Corporation, 2007.

[17] SANDERS, J., KANDROT, E., *CUDA by Example: An Introduction to General-Purpose GPU Programming*. 1st ed. Addison-Wesley Professional, 2010.

[18] KIRK, D. B., HWU, W.-M. W., *Programming Massively Parallel Processors: A Hands-on Approach*. 1st ed. Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 2010.

[19] LINDHOLM, E., NICKOLLS, J., OBERMAN, S., MONTRYM, J., "NVIDIA Tesla: A Unified Graphics and Computing Architecture", *IEEE Micro*, v. 28, pp. 39–55, March 2008.

[20] NVIDIA, *NVIDIA GeForce 8800 GPU Architecture Overwiew*, Tech. rep., 2006.

[21] WILKINSON, B., ALLEN, M., *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. 2nd ed. Prentice Hall, 2004.

[22] PACHECO, P. S., *Parallel programming with MPI*. Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 1996.

[23] NYLAND, L., HARRIS, M., PRINS, J., "Fast N-Body Simulation with CUDA", In: NGUYEN, H. (ed), *GPU Gems 3*, chap. 31, Addison Wesley Professional, August 2007.

[24] DE ALMEIDA, R. B., "Paralelização Utilizando Múltiplas GPUs de Elementos e Compostos Magénticos Baseado no Método de Monte Carlo", 2010.

[25] BENTLEY, J. L., "Multidimensional divide-and-conquer", *Commun. ACM*, v. 23, pp. 214–229, April 1980.

[26] PEÇANHA, J., CAMPOS, A., PAMPANELLI, P., LOBOSCO, M., VIEIRA, M., DANTAS, S., "Um Modelo Computacional para Simulação de Interações de Spins em Elementos e Compostos Magnéticos", *XI Encontro de Modelagem Computacional*, 2008.

[27] J. GOMES, C. HOFFMANN, V. S., VELHO, L., "Modeling in Graphics", *SIGGRAPH' 93 Course Notes*, 1993.

[28] LUIZ VELHO, JONAS GOMES, L. H. D. F., *Implicit Objects in Computer Graphics*. Springer-Verlag, 2002.

[29] IWANO, T. M., *Uso da Aplicação Normal de Gauss na Poligonização de Superfícies Implícitas*, Master's Thesis, Universidade Federal de Campina Grande, 2005.

[30] DRESDEN, T. U., *VampirTrace 5.10.1 User Manual*, Tech. rep.

[31] SWENDSEN, R. H., WANG, J.-S., "Nonuniversal critical dynamics in Monte Carlo simulations", *Physical Review Letters*, v. 58, n. 2, pp. 86+, January 1987.

[32] FUKUI, K., TODO, S., "Order-N cluster Monte Carlo method for spin systems with long-range interactions", *Journal of Computational Physics*, v. 228, n. 7, pp. 2629 – 2642, 2009.

[33] ISING, E., "Beitrag zur Theorie der Ferromagnetismus", *Z. Physik*, v. 31, pp. 253–258, 1925.

[34] SANTOS, E. E., RICKMAN, J. M., MUTHUKRISHNAN, G., FENG, S., "Efficient algorithms for parallelizing Monte Carlo simulations for 2D Ising spin models", *J. Supercomput.*, v. 44, n. 3, pp. 274–290, 2008.

[35] TOMOV, S., MCGUIGAN, M., BENNETT, R., SMITH, G., SPILETIC, J., "Benchmarking and implementation of probability-based simulations on programmable graphics cards", *Computers and Graphics*, v. 29, n. 1, pp. 71 – 80, 2005.

[36] PREIS, T., VIRNAU, P., PAUL, W., SCHNEIDER, J. J., "GPU accelerated Monte Carlo simulation of the 2D and 3D Ising model", *Journal of Computational Physics*, v. 228, n. 12, pp. 4468 – 4477, 2009.

[37] WEIGEL, M., "Simulating spin models on GPU", *ArXiv e-prints*, Jun 2010.

[38] BLOCK, B., VIRNAU, P., PREIS, T., "Multi-GPU Accelerated Multi-Spin Monte Carlo Simulations of the 2D Ising Model", *ArXiv e-prints*, Jul 2010.

[39] LIONETTI, F. V., MCCULLOCH, A. D., BADEN, S. B., "Source-to-source optimization of CUDA C for GPU accelerated cardiac cell modeling". In: *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part I*, *EuroPar'10*, pp. 38–49, Springer-Verlag: Berlin, Heidelberg, 2010.