Alessandra Matos Campos

Implementação Paralela de Um Simulador de Spins em Uma Unidade de Processamento Gráfico

Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Juiz de Fora

2008

Monografia submetida ao corpo docente do Instituto de Ciências Exatas da Universidade Federal de Juiz de Fora como parte integrante dos requisitos necessários para obtenção do grau de bacharel em Ciência da Computação.

Aprovado em 02 de Dezembro de 2008.

BANCA EXAMINADORA

Prof. Marcelo Lobosco, D. Sc.
Orientador

Prof. Marcelo Bernardes Vieira, D. Sc.
Co-Orientador

Prof. Sócrates de Oliveira Dantas, D. Sc.

Sumário

Lista de Figuras

1 Introdução

Os primeiros relatos sobre a existência do magnetismo terrestre vêm da antiguidade, por volta do ano 800 a.C. Desde a descoberta da magnetita, um mineral capaz de atrair fragmentos de ferro, o homem vem aprofundando seu conhecimento sobre os fenômenos magnéticos e empregando-o em seu cotidiano.

Os fenômenos magnéticos ganharam uma dimensão ainda maior a partir de 1820, quando os trabalhos de elaborados por Hans Christian Oersted, André Marie Ampère e Michael Faraday apontaram sua correlação com a eletricidade. Mais tarde, Albert Fert e Peter Grünberg descobriram, em trabalhos simultâneos e independentes, o efeito de magnetoresistência gigante. A evolução no conhecimento dos fenômenos magnéticos através destas e outras descobertas permitiu a criação de várias aplicações tecnológicas como o motor e o gerador elétrico, transformadores, fitas cassete, cartões de crédito e os *hard disks* (HDs).

Os materiais magnéticos desempenham um papel fundamental nas aplicações tecnológicas e são utilizados em três categorias principais de aplicação: como ímãs permanentes, capazes de criar um campo magnético constante, como materiais permanentes doces, que se magnetizam e desmagnetizam facilmente e produzem um campo magnético muito maior do que o campo criado por um solenóide, e como dispositivos de gravação magnética.

A magnetização dos materiais envolvidos nos fenômenos magnéticos é determinada pelo momento de *spin* de seus elétrons. O *spin* reflete o movimento de rotação dos elétrons em torno do seu próprio eixo. Cada *spin* está associado a um momento magnético e tende a se alinhar na direção de um campo magnético externo quando submetido a ele.

O momento de *spin* é uma propriedade interna dos átomos, cuja escala é nanométrica. Por isso, a simulação computacional é uma ferramenta imprescindível na elaboração de experimentos que posssibilitem aos cientistas fazer uma análise comportamental desta propriedade em riqueza de detalhes e consequentemente surjam novas descobertas.

1.1 Objetivos

O estudo do momento de *spin* dos elétrons de um material magnético tem por base a energia produzida pela interação entre todos os momentos de *spin* pertencentes à estrutura molecular do material. A energia é determinada por um modelo físico-matemático. O foco deste trabalho está voltado para os materiais ferromagnéticos. Estes materiais possuem como característica um forte grau de magnetismo mesmo com a ausência de campo magnético e em temperaturas elevadas.

A estrutura molecular de qualquer substância possui milhares de átomos. Logo, calcular a energia produzida pela interação dos elétrons desta estrutura e simular o seu comportamento em diversas situações é uma tarefa árdua e requer uma alta capacidade computacional. As simulações demoram um tempo considerável dependendo da estrutura molecular analisada. Dessa forma, o objetivo deste trabalho é fazer com que as funções que determinam a energia de interação entre os *spins* de uma estrutura molecular ferromagnética sejam calculadas mais rapidamente, diminuindo o tempo de simulação. Para isto, foi concebido um simulador de *spins* que adota uma unidade de processamento gráfico para efetuar os cálculos necessários já que este *hardware* suporta a execução de uma mesma operação em paralelo, isto é, simultaneamente. A programação de unidades de processamento gráfico necessita de uma ferramenta apropriada para a manipulação do *hardware*. Neste trabalho, optou-se pela tecnologia CUDA.

1.2 Visão Geral

As etapas de construção do simulador de *spins* serão apresentadas nos próximos capítulos de acordo com a estrutura a seguir.

No Capítulo 2 faz-se uma abordagem sobre as técnicas de simulação para o modelo físico-matemático dos *spins*, apresentando o conceito de Dinâmica Molecular e Simulação Computacional. Além disso, também é mostrado o modelo físico-matemático que provê a energia de interação entre os *spins*, o principal foco de estudo para o desenvolvimento da aplicação.

O Capítulo 3 explica a computação de propósito geral em unidades de processamento gráfico (GPGPU). Nele, será apresentada a evolução das GPUs, o que fornece a base para o entendimento desta técnica. Logo após, menciona-se por que a GPGPU tem sido amplamente empregada pela comunidade científica. O final do capítulo se destina

a tecnologia CUDA, descrevendo a arquitetura da placa gráfica NVIDIA GeForce 8800 GTX e como o modelo de hardware e de programação do CUDA se aplica a esta placa.

O Capitulo 4 trata da aplicação propriamente dita, a Monte Carlo Spin Engine. Primeiramente serão mostradas as características da MCSE: quais módulos a compõe e como o problema foi paralelizado. O tópico subseqüente expõe os resultados de simulação obtidos para a aplicação, além de avaliar o desempenho obtido ao paralelizar a aplicação.

No último capítulo são apresentadas as considerações finais sobre este trabalho.

2 *Técnicas de Simulação e Modelo Teórico dos Spins*

2.1 Dinâmica Molecular e Simulação Computacional

A Dinâmica Molecular é um método proposto por Alder e Wainwright (ALDER; WAINWRIGHT, 1959) utilizado para simular o comportamento mecânico de sistemas moleculares. Os elementos mais relevantes para tal simulação são o conhecimento do potencial de interação entre as partículas e as equações de movimento que controlam a sua dinâmica. A mecânica Newtoniana é comumente empregada para descrever o movimento das partículas. O potencial adotado para uma simulação pode ser simples, como, por exemplo, o potencial gravitacional em interações de corpos macroscópicos, ou possuir mais de um termo, como o que descreve as interações entre átomos e moléculas. Com estes elementos obtêm-se uma série de informações para todas as partículas (ou átomos) e para o sistema como um todo, a partir das quais várias propriedades físicas podem ser calculadas.

A Mecânica Estatística e a Física da Matéria Condensada, comumente empregam o método da Dinâmica Molecular na solução de problemas relacionados a essas áreas da física. A Física da Matéria Condensada é o campo da física que trata das propriedades físicas da matéria. Na fase "condensada"que aparece sempre que o número de constituintes de um sistema (átomos, elétrons, etc.) é extremamente grande e as interações entre os constituintes são fortes. Já a Mecânica Estatística (ou física estatística) é o ramo da física que estuda as propriedades de sistemas com elevado número de entidades (átomos, moléculas, íons, entre outros) a partir do comportamento coletivo destas entidades.

Alguns problemas em Mecânica Estatística são exatamente solúveis. São os problemas triviais, em que a partir das propriedades macroscópicas 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 das partículas é um exemplo do conjunto de problemas da Mecânica Estatística considerado não trivial devido à complexidade de 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 é algo extremamente difícil ou quase impossível visto que os sistemas dessa classe são altamente complexos de serem analisados.

A idéia da simulação computacional é que se a função matemática do problema é conhecida, neste caso a função de interesse corresponde à do potencial de interação intermolecular, pode-se então fazer a evolução temporal (ou mapeamento configuracional) até que o sistema atinja o equilíbrio. Então podemos utilizar os conceitos da mecânica estatística para obtermos as propriedades termodinâmicas macroscópicas do sistema em equilíbrio. As duas grandes técnicas de simulação utilizadas hoje são a dinâmica molecular e o método de Monte Carlo.

Devido à abordagem deste trabalho ser a análise comportamental de um sistema molecular em termos da energia potencial entre as partículas, o fator mecânico será desprezado. O potencial de interação entre as partículas acima mencionado será adotado, sendo este composto das energias potenciais de interação dipolar magnética, de estado de magnetização do sistema e a de interação com o campo magnético externo aplicado. Também será levado em conta o tipo de magnetização do material que constitui as moléculas (ou átomos) do sistema. Cada um dos termos relatados serão apresentados a seguir, bem como os métodos numéricos utilizados na modelagem computacional do problema.

2.2 Energias Potenciais de Interação

Como foi visto anteriormente, o cálculo da energia potencial de interação entre as partículas de um sistema molecular é um elemento fundamental no estudo destes sistemas através da dinâmica molecular ou do método de Monte Carlo. Quando moléculas ou átomos aproximam-se uns dos outros, dois fenômenos podem ocorrer: (i) eles podem reagir ou (ii) eles podem interagir. Uma interação química significa que as moléculas se atraem ou se repelem entre si, sem que ocorra a quebra ou formação de novas ligações químicas. Estas interações são freqüentemente chamadas interações intermoleculares.

As interações intermoleculares surgem devido às forças essencialmente de natureza

elétrica ou magnética (chamadas de forças intermoleculares), que fazem com que uma partícula influencie o comportamento de outra partícula em suas proximidades. Uma vez que estas forças intermoleculares se originam do contato não reativo entre duas 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: $\boldsymbol{E}$ (intermolecular) = $\boldsymbol{E_{i\text{-}j}}$ - ($\boldsymbol{E_i}$ + $\boldsymbol{E_j}$). Isto corresponde à sua decomposição em vários componentes: $\boldsymbol{E}$ (intermolecular) = $\boldsymbol{E}$ (longo alcance) + $\boldsymbol{E}$ (curto alcance).

Nesta seção, serão apresentadas componentes que fornecem energia intermolecular ou energia potencial de interação, cuja fórmula é expressa por:

$$E_t = \frac{A}{2} \sum_{i,j=1\ e\ i\neq j}^{N} \left\{ \frac{\mathbf{S}_i \cdot \mathbf{S}_j}{|\mathbf{r}_{ij}|^3} - 3\frac{[\mathbf{S}_i \cdot \mathbf{r}_{ij}]\,[\mathbf{S}_j \cdot \mathbf{r}_{ij}]}{|\mathbf{r}_{ij}|^5} \right\} - J \sum_{i,k=1\ e\ i\neq k}^{N} \mathbf{S}_i \cdot \mathbf{S}_k - \sum_{i=1}^{N} D(\mathbf{S}_i \cdot \boldsymbol{H}) \quad (2.1)$$

2.2.1 Energia de Interação Dipolar Magnética

Um dipolo magnético é um sistema constituído por dois pólos magnéticos iguais, mas de sinais opostos, separados por uma pequena distância finita. Pela teoria eletromagnética, elétrons que circulam ao redor de núcleos atômicos, de seus próprios eixos e de núcleos atômicos carregados positivamente são todos dipolos magnéticos. A soma destes efeitos pode se cancelar, de forma que um determinado tipo de átomo pode não ser um dipolo magnético. Se eles não se cancelam completamente, o átomo é um dipolo magnético permanente, como são, por exemplo, os átomos de ferro. Muitos milhões de átomos de ferro, espontaneamente, se mantém no mesmo alinhamento para formar um domínio ferromagnético, constituindo também um dipolo magnético. As agulhas de bússolas magnéticas e imãs de barra são exemplos de dipolos magnéticos macroscópicos. A interação dipolar magnética fornece o nível de energia presente em cada um dos possíveis dipolos formados entre as partículas de um sistema molecular. A energia atribuída à

interação dipolar é dada pela fórmula:

$$E_{DD} = \frac{A}{2} \sum_{i,j=1\ e\ i\neq j}^{N} \left\{ \frac{\mathbf{S}_i \cdot \mathbf{S}_j}{|\mathbf{r}_{ij}|^3} - 3\frac{[\mathbf{S}_i \cdot \mathbf{r}_{ij}]\,[\mathbf{S}_j \cdot \mathbf{r}_{ij}]}{|\mathbf{r}_{ij}|^5} \right\} \tag{2.2}$$

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

2.2.2 Energia de Interação Ferromagnética e Energia de Interação com o Campo Magnético Externo

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 qual está relacionado com seu momento de *spin* (Figura 1). Entende-se por momento de *spin* como o momento de rotação do átomo em torno do seu eixo. Em mecânica quântica, o *spin* de um átomo, dado por um vetor tridimensional $\boldsymbol{S}$, refere-se as possíveis orientações que as partículas subatômicas (prótons, elétrons, nêutrons e alguns núcleos atômicos) possuem quando estão sob ação, ou não, de um campo magnético externo. Os *spins* podem ser representados matematicamente com vetores orientados, nos permitindo, com isso, incrementá-los em formulações que nos retornam algumas características do sistema molecular estudado.

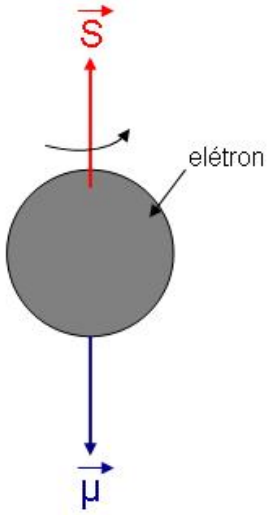

Figura 1: Momento de *Spin*.

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 nos fornece o que chamamos de magnetização interna $\boldsymbol{M}$ do material. A partir dela e do campo magnético externo $\boldsymbol{H}$ podemos então definir matematicamente o nível de magnetismo de um sistema molecular pela relação: $\boldsymbol{X} = \boldsymbol{M}/\boldsymbol{H}$, onde $\boldsymbol{X}$ é valor obtido para definir o nível de magnetização do material do qual o sistema é formado, chamado de susceptibilidade magnética.

Para ($\boldsymbol{X} \gg 1$) dizemos que o material é ferromagnético ou ferrimagnético, dependendo da orientação dos *spins* em relação ao campo magnético externo, diamagnético para ($\boldsymbol{X} < 1$), paramagnético para ($\boldsymbol{X} > 0$) e antiferromagnético para $\boldsymbol{X}$ pequeno. Os materiais ferromagnéticos serão os principais objetos de estudo deste trabalho.

O ferromagnetismo é causado por uma forte interação entre os elétrons pertencentes a uma camada incompleta de átomo ou entre os elétrons de átomos vizinhos. Esta interação, considerada de curto alcance, faz com que a energia entre dois átomos seja menor caso seus *spins* possuam a mesma orientação.

Os materiais ferromagnéticos, como ferro, cobalto e níquel, por exemplo, apresentam um alto valor positivo de susceptibilidade magnética. Quando um campo magnético externo é aplicado a esses materiais, os momentos magnéticos de cada partícula desses materiais tendem a alinhar-se na mesma direção e sentido do campo magnético externo.

Uma parcela da energia potencial de um sistema molecular é obtida com a interação dos momentos magnéticos com o campo magnético externo e com sua vizinhança através da equação:

$$E_{FCE} = -J \sum_{i,k=1 \; e \; i \neq k}^{N} \mathbf{S}_i \cdot \mathbf{S}_k - \sum_{i=1}^{N} D(\mathbf{S}_i \cdot \boldsymbol{H}) \tag{2.3}$$

O primeiro termo fornece a energia de interação ferromagnética, onde i representa uma partícula do sistema molecular, k representa cada partícula pertencente à vizinhança de i, sendo i sempre diferente de k, visto que uma partícula não interage com ela mesma, e $\boldsymbol{J}$ é denominada de constante ferromagnética.

2.3 Métodos Numéricos Aplicados à Simulação Computacional

2.3.1 Método de Monte Carlo

O método de Monte Carlo(FERNANDES; RAMALHO, 1989)foi desenvolvido por Nicholas Metropolis e Stanislaw Ulam durante a segunda guerra mundial para estudar a difusão de nêutrons durante o fenômeno de fissão nuclear. O método explora as propriedades estatísticas de números gerados aleatoriamente para assegurar que o resultado correto é computado da mesma maneira que num jogo de cassino para se certificar de que a "casa" sempre terá lucro. Por esta razão, a técnica de resolução de problemas é chamada de método de Monte Carlo. Com ele, é possível obter aproximações numéricas para funções complexas através da conversão de um modelo físico ou matemático em um modelo estatístico. Os resultados são gerados sobre uma distribuição de probabilidades e a amostra significativa obtida por tentativas aleatórias é usada para aproximar a função de interesse, isto é, não há necessidade de reproduzir todas as configurações do sistema. A precisão do resultado final depende em geral do número de tentativas.

2.3.2 Algoritmo de Metropolis

Antes de apresentar o algoritmo de Metropolis, é necessário mencionar a distribuição de Boltzmann, uma vez que ela é a base para o algoritmo.

O físico austríaco Ludwig Eduard Boltzmann, que deu nome a essa distribuição de probabilidade, escreveu diversos artigos nos anos 1870 e estabeleceu as bases da mecânica estatística, mostrando que a segunda lei da termodinâmica poderia ser explicada através da aplicação nos átomos das leis da mecânica e da teoria de probabilidades.

A distribuição de Boltzmann descreve a probabilidade de se encontrar o sistema em um determinado estado de energia.

O sistema tem um número de configurações possíveis para seus elementos e cada uma dessas configurações é considerada um microestado do sistema. Cada conjunto de microestados que apresentam a mesma energia é definido como um macroestado do sistema. Adotando-se a hipótese de que todos os microestados têm a mesma probabilidade de ocorrência, é possível afirmar que a probabilidade $\boldsymbol{p_m}$ de ocorrência de um macroestado m, que corresponda a uma certa energia $\boldsymbol{E_m}$, é proporcional ao número de microestados que esse macroestado contém, ou seja, o número de microestados que apresentam energia

$\boldsymbol{E_m}$.

De maneira geral, o algoritmo de Metropolis determina valores esperados de propriedades de um sistema em simulação calculando-se uma média sobre uma amostra, 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: assim, 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. A Figura 2 descreve as etapas do algoritmo de Metropolis durante a simulação computacional da energia total de um sistema molecular.

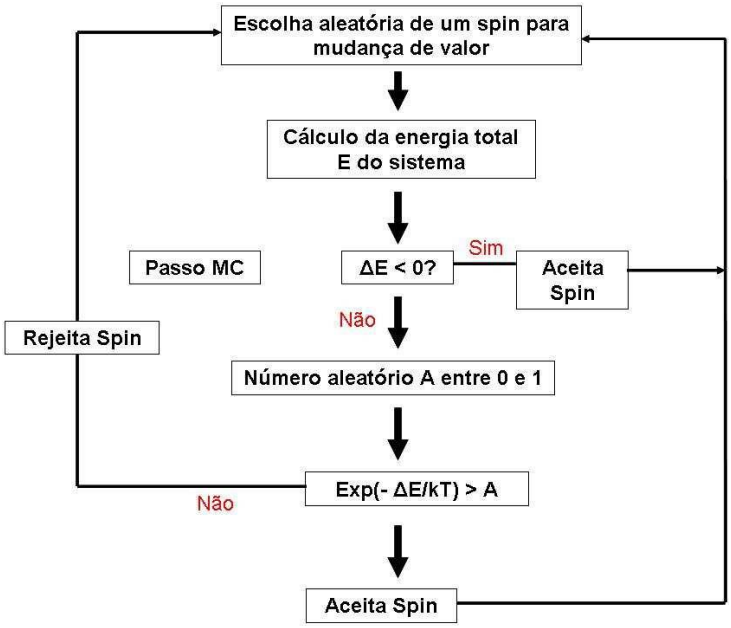

Figura 2: Fluxograma do algoritmo de Metropolis onde ΔE é a variação de energia obtida (antes e após a mudança do *spin*).

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* a mudar de direção

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

d) Cálculo da nova energia total do sistema;

e) Se a variação de energia obtida for < 0, 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 numero aleatório $\boldsymbol{A}$ no intervalo[0 ,1]. Este número corresponde

à probabilidade pm apresentada no início desta subseção.

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

f) Repete-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 *Computação de Propósito Geral nas GPUs*

GPGPU significa *General Purpose Graphics Processing Unit*, isto é, unidade de processamento gráfico de propósito geral. Esta nova técnica de programação foi idealizada com base na evolução do pipeline gráfico, que será explicado a seguir. As seções seguintes tratam, respectivamente, sobre a GPGPU de fato, por que esta técnica vem se tornando cada vez mais adotada e última seção descreve brevemente a tecnologia CUDA.

3.1 Evolução do *Pipeline* Gráfico

A historia das placas de vídeo começa nos anos 60, quando as telas passaram a ser utilizadas no lugar de impressoras como elemento de visualização nos computadores de grande porte. A partir dessa idéia e da evolução na arquitetura dos computadores surgiu, em 1981, a primeira placa gráfica para computadores pessoais lançada pela IBM com o modelo IBM-PC. A placa possuía 4 KB de memória de e suportava apenas uma cor. A saída gráfica era textual e era mostrada em um monitor MDA (*Monochrome Display Adapter*). Desde então o investimento em novas tecnologias para placas gráficas expandiu rapidamente acompanhado do lançamento dos vídeo games de console.

Nesta época, surgiram alguns padrões de interface de vídeo como o CGA(*Color Graphics Adapter*) que possuía 16KB de memória e suporte ao modo gráfico, com resolução máxima de 640 x 200, o EGA (*Exchange Graphics Array*) que suportava a geração de imagens em segunda dimensão (2D), o VGA (*Vídeo Graphics Array*) que foi uma melhoria do EGA(*Enhanced Graphics Adapter*) e o SVGA (*Super Vídeo Graphics Array*) resultado do aprimoramento do padrão VGA. Os últimos padrões tornaram-se os mais utilizados, pois proviam melhor qualidade na interface gráfica.

Porém o lançamento do Microsoft Windows, no início dos anos 90, fez com que as placas gráficas existentes não conseguissem efetuar o processamento gráfico de maneira

correta. Isto acontecia devido á interface gráfica do novo sistema operacional ser mais aprimorada, exigindo mais recursos das placas gráficas. No entanto, elas eram capazes apenas de exibir o conteúdo de sua memória na tela. A construção das imagens pixel a pixel ficava a cargo das unidades de ponto flutuante (FPUs) dos processadores (CPUs). Desta forma, a performance dos computadores ficou prejudicada, pois o processamento gráfico ocupava a CPU na maior parte do tempo.

A solução para o problema veio em 1994 quando apareceram as placas aceleradoras gráficas 2D. Elas foram as primeiras a ter um chip dedicado ao processamento gráfico em segunda dimensão em PCs: a GPU ou Unidade de Processamento Gráfico. Este chip é um microprocessador otimizado para calcular operações de ponto flutuante que são a base para construção de imagens. As placas NVIDIA TNT, NVIDIA TNT2 e ATI Rage Fury são exemplos desta primeira geração de placas com GPU.

Como o nível de realismo dos jogos de vídeo game e de computadores aumentou consideravelmente, os fabricantes incorporaram funções mais elaboradas as placas gráficas 2D, permitindo a geração de imagens em terceira dimensão (3D). Porém, o desempenho das novas placas não correspondeu às expectativas dos fabricantes e elas não conseguiram evoluir de forma desejável.

Em 1999, a NVIDIA lançou no mercado a GeForce 256, o primeiro hardware dedicado a geração de imagens 3D. Isto caracterizou a segunda geração de GPUs da qual também fazem parte as placas 3dfx Voodoo 2 e 3dfx Voodoo 3. Quanto à arquitetura, as GPUs desta geração ainda possuíam um *pipeline* fixo (Figura 3, isto é, os dados de entrada da GPU eram processados por uma série de funções pré-definidas pelo fabricante até que a imagem final se formasse. O processo de formação de imagens ocorre na ordem mostrada a seguir:

a) Aplicação (*Application*): Prepara e carrega os dados a serem utilizados pela placa gráfica e emite comandos através de uma API;

b) Comandos (*Commands*): Inserem os dados no *pipeline* gráfico;

c) Operações de Geometria (*Geometry*): Esta etapa é responsável pelo cálculo das coordenadas de tela do conjunto de vértices vindos da etapa anterior, geração das coordenadas de textura e iluminação dos vértices, que corresponde à transformação dos vértices, e a montagem dos vértices transformados em primitivas geométricas (triângulos quadriláteros, linhas ou pontos), operação chamada *Primitive Assembly*.

d) Rasterização (*Rasterization*): Determina o conjunto de *pixels* cobertos por uma

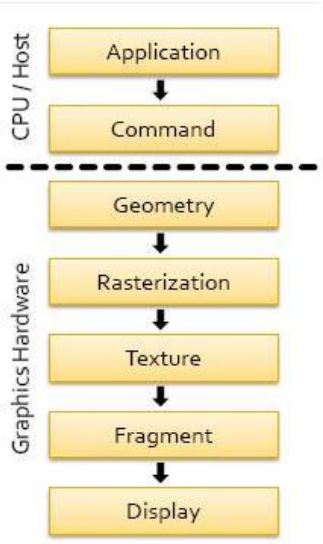

Figura 3: Etapas do *pipeline* fixo

primitiva geométrica. Esta etapa produz um fragmento correspondente para cada pixel. O conjunto de atributos de cada pixel é calculado por interpolação dos atributos dos vértices da primitiva.

e) Inserção de Textura (*Texture*): Busca a textura na memória através das coordenadas calculadas na etapa (c). Aplica a textura nos dados provenientes da etapa (d).

f) Operações de Fragmento (*Fragment*): Aplica uma função a cada fragmento para calcular sua cor final e sua profundidade.

g) Composição final da Imagem (*Display*): Nesta etapa comparam-se os valores dos atributos de cada fragmento com os valores correspondentes no framebuffer, realizando-se uma série de testes. Após a comparação, se os valores forem válidos, o fragmento é escrito no framebuffer. O framebuffer consiste em um conjunto de *pixels* (elementos da figura) dispostos em um arranjo bidimensional. Ele armazena os *pixels* que serão mostrados na tela (Figura 4).

Em 2001 as GPUs alcançaram a terceira geração. O lançamento das APIs gráficas, OpenGL e DirectX, e o lançamento da placa GeForce 3 introduziram a programabilidade do estágio de processamento de vértices do *pipeline* com a criação do processador de vértices (*Vertex Processor*). Isto permitiu aos programadores ajustar as funções pré-definidas para esse estágio a fim de produzir resultados sob medida para cada aplicação gráfica. Os programas criados através das APIs citadas são denominados shaders. As

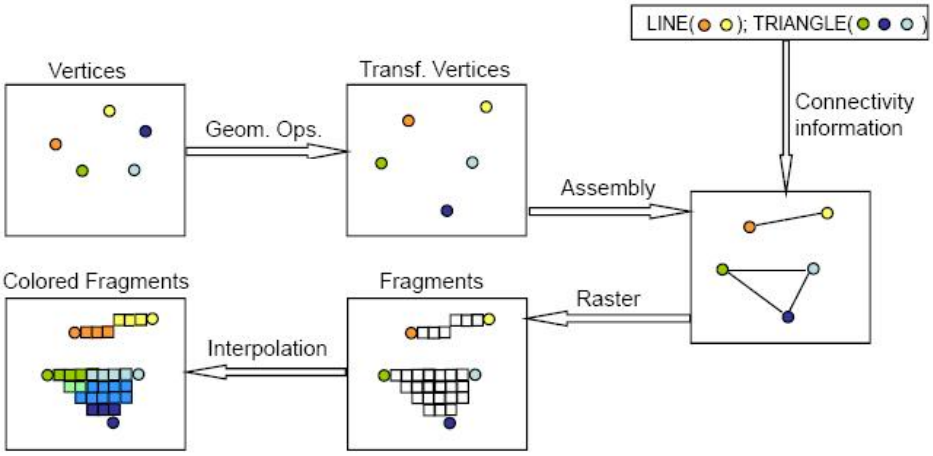

Figura 4: Passos para a construção de uma imagem seguindo o modelo de *pipeline* fixo

placas GeForce 4, ATI Radeon 7500 e ATI Radeon 8500 também são exemplos desta geração.

A quarta geração de GPUs foi apresentada ao mundo em 2002. Fazem parte desta geração as placas gráficas GeForce FX, ATI Radeon 9600, ATI Radeon 9700 e ATI Radeon 9800. Estas placas tinham como característica marcante a programação ao nível de *píxel*, com a transformação do estágio de fragmentação em uma unidade programável chamada Fragment Processor, ampliando ainda mais a programabilidade do hardware (Figura 5). Alguns programadores, baseados nesta característica e na alta capacidade de processamento paralelo do chip, experimentaram mapear aplicações não-gráficas nas APIs existentes, de modo que a GPU desempenhasse a mesma função da CPU. Embora o processo de mapeamento fosse altamente complexo, os programadores obtiveram êxito. Daí surgiu o termo GPGPU e com ela algumas linguagens de programação tais como Cg (C *for Graphics*), Sh, BrookGPU e HLSL (*High-Level Shader Language*), baseadas nas APIs gráficas existentes.

Ao observar a expansão da GPGPU, as fabricantes NVIDIA e ATI passaram a investir em tecnologias que oferecessem um suporte mais adequado a GPGPU. A ATI lançou a CTM (*Close To Metal*) que consiste em algumas unidades de hardware incorporadas as suas placas gráficas para controle de execução de programas e uma série de instruções de baixo nível que potencializam o uso das unidades de ponto flutuante das GPUs. Já a NVIDIA lançou uma série de placas denominada NVIDIA GeForce 8. Foram as primeiras placas gráficas da quinta geração. As GPUs desta série trouxeram uma nova arquitetura de hardware e de software onde os programadores puderam de fato desenvolver aplicações

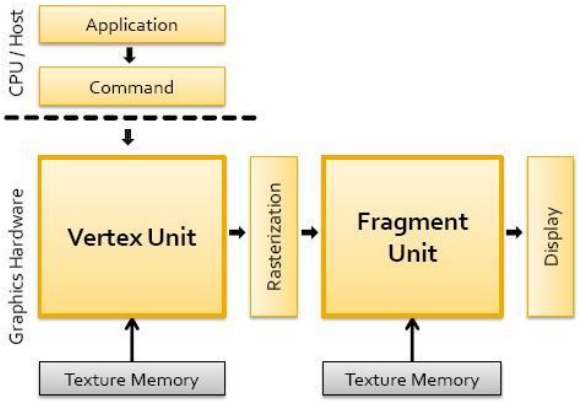

Figura 5: *Pipeline* Programável.

GPGPU em um ambiente de programação próximo ao utilizado na programação para CPUs . A nova arquitetura foi chamada de CUDA. O CUDA será visto em detalhes nas próximas seções. Além disso, introduziu o conceito de *shaders* unificados (Figura 6). Estas novas unidades de hardware foram criadas com o intuito de aumentar o aproveitamento do hardware durante a execução do *pipeline*. O uso dos shaders unificados resolveu o problema do desperdício de recursos em situações onde há uso intenso de Vertex Shader e pouco Pixel Shader ou o contrário, aumentando bastante o desempenho. Com *Vertex Shaders* e *Pixel Shaders* que estão designados a executar um tipo exclusivo de operação, em situações de uso intenso de um, o hardware responsável pelo outro é sub-utilizado (fica ocioso) e o desempenho não é ideal. Os Shaders Unificados podem executar ambos os tipos, aproveitando melhor as unidades disponíveis e atingindo um desempenho ótimo.

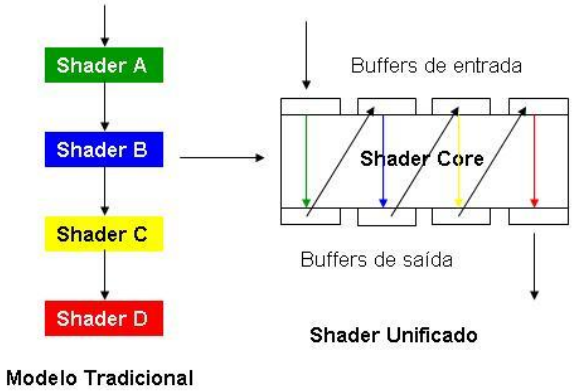

Figura 6: Arquitetura de *Shaders* Unificados.

3.2 Por que usar a GPGPU?

Aplicações que requerem maior processamento de dados fazem parte do que se chama *High Performance Computing* (HPC) ou Computação de Alta Performance. Tradicionalmente, este grupo especial de aplicações necessita de processadores com maior capacidade computacional ou recorrem ao processamento distribuído para a obtenção de resultados em menor tempo, a fim de melhorar o desempenho. À medida que as aplicações HPC tornam-se mais complexas, maior é a capacidade de processamento de dados exigida. Aumentar a capacidade de processamento das CPUs significava até pouco tempo diminuir o tamanho dos transistores com o intuito de que um chip pudesse acomodá-los em maior número.

É conhecido pela Lei de Moore (KOCH, 2005) que o número de transistores por circuito integrado cresce exponencialmente a cada dois anos. No entanto, quanto maior o número de transistores, maior é o calor produzido pelo chip. Além disto, as memórias não acompanharam a evolução dos processadores na mesma escala. Com base nisto, tecnologias como a Hyper-Threading da Intel e Multi-Core foram concebidas para amenizar este problema.

Embora as CPUs com estas tecnologias fossem utilizadas pela HPC, a GPGPU tornou-se mais atrativa devido ao fato das GPUs possuírem mais de unidades de ponto flutuante do que as CPUs e portanto serem capazes de efetuar cálculos complexos com maior rapidez, executando-os em paralelo (Figura 7). Desta forma, as placas gráficas podem trabalhar em conjunto com os processadores executando todo o processamento ou funcionando como um co-processador, dividindo a carga de execução com os processadores. Uma outra forte característica provida pela GPGPU é a taxa de poder computacional por treal investido. O que sai mais caro: Uma 8800GT, que custa 500 reais no mercado legal e consome no máximo 160w em carga máxima ou 100 processadores Xeon com pelo menos 2.8 GHz baseados no Core2 consumindo no mínimo mais de 65w cada um?

Vale lembrar que uma placa de vídeo *desktop* 8800GT da NVIDIA utilizando o CUDA tem capacidade de processamento suficiente parae substituir os 100 processadores Xeon citados, os quais são baseados no Core2, uma das mais recentes plataformas da Intel, em cálculo de ponto flutuante.

A seguir encontram-se algumas aplicações não-gráficas, incluindo aplicações HPC, produzidas através da GPGPU:

- Clusters de computadores (utilizando a tecnologia GPU cluster) para tarefas de

Figura 7: Componentes da CPU e da GPU.

cálculos altamente intensivos:

- Simulação física e motores físicos (geralmente baseados em modelos físicos Newtonianos)
- Segmentação - 2D e 3D
- Tomografia Computacional
- Processamento de Sinais de Áudio
- Processamento de Imagens Digitais
- Processamento de Video
- *Ray tracing*
- Computação Geométrica - *Constructive Solid Geometry* (CSG), detecção de colisões, geração de sombras
- Computação Científica
  - Meteorologia
  - Estudo do clima (incluindo estudo do aquecimento global)
  - Modelagem molecular
  - Mecânica quântica
- Bio-informática
- Computação financeira
- Imagem Médica
- Operações em Base de Dados
- Criptografia e Criptoanálise

3.3 CUDA (Compute Unified Device Architecture)

A tecnologia CUDA 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, que provê acesso às instruções da GPU e ao controle da memória de vídeo para explorar o paralelismo encontrado nas placas gráficas atuais[ref progaming guide]. O 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. O CUDA tem como características:

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

3.3.1 Arquitetura da GeForce 8800 GTX

Esta subseção visa mostrar a arquitetura da placa gráfica GeForce 8800 GTX (Figura 8). Embora outras placas sejam compatíveis com o CUDA os detalhes arquiteturais deste modelo, em particular, se tornam interessantes para este trabalho, o que será explicado no capítulo 4.

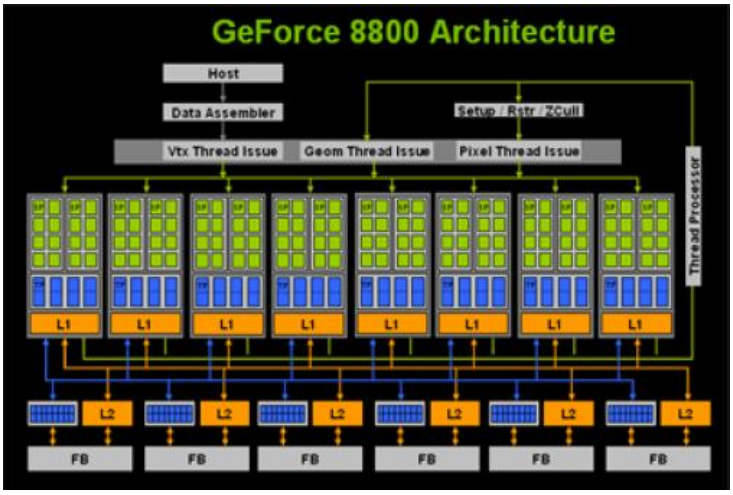

Figura 8: Interior do *chip* gráfico da GeForce 8800 GTX.

A figura anterior mostra os elementos formadores do chip G80 e sua comunicação com a memória de vídeo e a CPU, aqui referida 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*, destacados pela cor verde. 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 cento e vinte e oito ULAs, dentro 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.

Os SPs da placa gráfica GeForce 8800 GTX não possuem função específica dentro

do *pipeline* gráfico, pois eles foram concebidos de modo a serem *Shaders Unificados*, mencionados anteriormente. Tanto do ponto de vista gráfico quanto do ponto de vista da GPGPU isto significa o uso completo do hardware e consequentemente maior desempenho e maior eficiência no processamento.

Na arquitetura do chip G80 ainda podem ser observados 16 *Streaming Multiprocessors* (SM). Cada SM é constituído por oito SPs que por sua vez compartilham uma área de memória de tamanho máximo de 16 KB, oito unidades de filtragem de texturas (os blocos azuis rotulados TF, *Texture Filtering*), quatro unidades de endereçamento de texturas (não desenhadas na Figura 9) e um cache de memória L1 em destaque na cor laranja.

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* identificado pela cor cinza no topo da figura.

3.3.2 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 pelo 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 (Figura 9).

Os blocos podem ser representados como uma matriz, onde o acesso as *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 8, no máximo oito *threads* por bloco e no máximo dezesseis blocos são executados

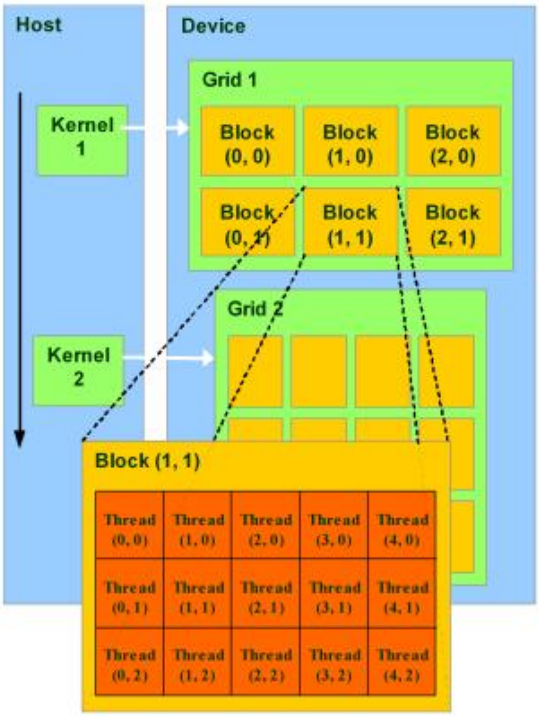

Figura 9: Esquema de representação do modelo de programação da tecnologia CUDA

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 o seu conjunto de registradores, uma pequena memória local (de uso exclusivo de cada SP), a memória compartilhada pertencente ao bloco onde o SP se encontra e 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. Um *kernel* pode ser três tipos definidos pelo 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 *grid* é informado. Isto será explicado melhor no exemplo de código a seguir. O último tipo, device especifica que o *kernel* será chamado e executado pela GPU.

O exemplo abaixo mostra a adição de dois vetores utilizando o CUDA. (Figura 10).

Basicamente a escrita do código segue os seguintes passos:

```
#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];}

int main(){

/* pointers to host memory */
float *a, *b, *c;
/* pointers to device memory */
float *a_d, *b_d, *c_d;

int N=18;
int i;

/* Allocate arrays a, b and c on host*/
a =         (float*)    malloc(N*sizeof(float));
b =         (float*)    malloc(N*sizeof(float));
c =         (float*)    malloc(N*sizeof(float));

/* Allocate arrays a_d, b_d and c_d on device*/
cudaMalloc ((void **) &a_d, sizeof(float)*N);
cudaMalloc ((void **) &b_d, sizeof(float)*N);
cudaMalloc ((void **) &c_d, sizeof(float)*N);
/* Initialize arrays a and b */
/* Copy data from host memory to device memory */
cudaMemcpy(a_d, a, sizeof(float)*N, cudaMemcpyHostToDevice);
cudaMemcpy(b_d, b, sizeof(float)*N, 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 a and b, store result in c */
add_arrays_gpu<<<dimGrid, dimBlock>>>(a_d, b_d, c_d, N);

/* Copy data from deveice memory to host memory */
cudaMemcpy(c, c_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
/* Print c */
/* Free the memory */
free(a); free(b); free(c);
cudaFree(a_d); cudaFree(b_d);cudaFree(c_d);}
```

Figura 10: Código CUDA mostrando a adição de dois vetores.

1. Inclusão das bibliotecas necessárias
2. Declaração de funções;
3. No método principal:
4. Iniciar o dispositivo;
5. Alocar memória na GPU;
6. Transferir os valores da CPU para a GPU;
7. Chamar o *kernel*;

8. Copiar os resultados de volta para a CPU;
9. Liberar a memória alocada;
10. Finalizar o dispositivo.

4 Monte Carlo Spins Engine

A técnica de Simulação Computacional mostrada no capítulo 2 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. Para resolver este conflito, é comum a adoção estratégias de computação paralela a fim de garantir que os resultados de uma simulação possam ser produzidos num intervalo de tempo viável e de forma correta. Tendo em vista a necessidade de uma ferramenta que combinasse uma simulação realista do modelo teórico dos *spins* magnéticos e a computação paralela, foi proposta uma aplicação voltada à Simulação Computacional baseada no Método de Monte Carlo e no Algoritmo de Metropolis, batizada de Monte Carlo Spins Engine ou MCSE.

A Monte Carlo Spin Engine é um simulador de sistemas moleculares magnéticos que provê informações acerca da mudança de estados do sistema, ocasionada pela alteração dos *spins*, ao longo do "tempo" (passos de Monte Carlo). A aplicação se divide em duas partes distintas. A visualização dos resultados e a manipulação dos parâmetros que modelam as condições do sistema ficam a cargo da interface gráfica. Já o processamento numérico envolvido nos cálculos das equações fornecidas pelo modelo físico-matemático fica a cargo do núcleo de simulação. Este é dividido em três módulos distintos: Seqüencial, Multithread e CUDA.

4.1 Interface Gráfica

A interface gráfica da MCSE é a responsável pelo fluxo de entrada e saída de dados da aplicação. A saída de dados ou visualização consiste em disponibilizar graficamente os resultados gerados pelo núcleo de simulação. A interface de visualização (Figura 11)

permite ao usuário analisar o sistema como um todo, coletando informações relacionadas à posição de todos os *spins* no espaço tridimensional, energia das partículas, orientação da magnetização interna do sistema (seta vermelha), orientação do campo magnético externo aplicado (seta preta), o número de iterações/s (utilizado como métrica desempenho) e o número de *spins* pertencentes à grade. A energia é mostrada pela coloração do vetor de orientação de cada *spin*, onde a cor azul indica energias mais baixas e a cor vermelha energias mais altas. O número de *spins* pertencentes à grade nem sempre corresponde ao tamanho da grade. Isto acontece quando se define implicitamente no código a forma do objeto. No caso da Figura 11 o objeto, representando o sistema, possui a forma de um cubo o qual está completamente preenchido.

Dois gráficos complementam as informações acima. Um diz respeito ao nível de magnetização (Figura 13) em cada eixo coordenado da grade enquanto o outro traça o histórico do cálculo de energia do sistema nos eixos coordenados x, y e z (Figura 12) bem como a média dos valores de energia encontrados.

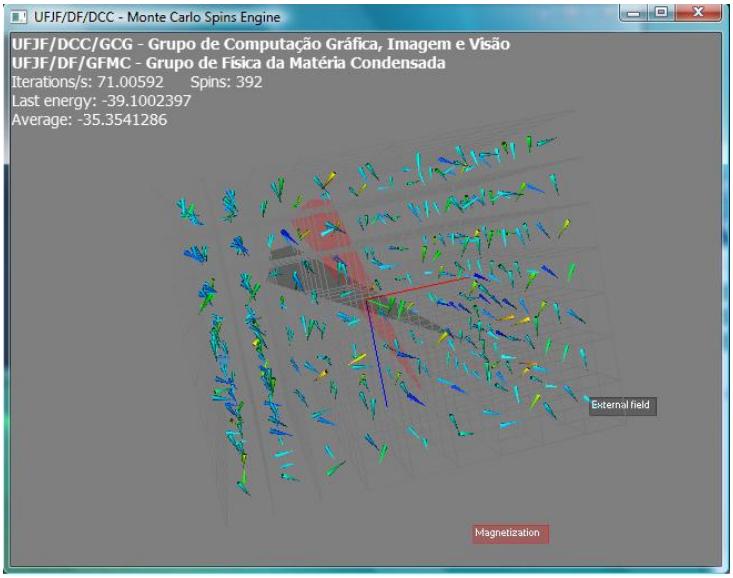

Figura 11: Representação dos *spins* dentro de uma grade de simulação regular, os dois vetores no centro mostram o campo externo e a magnetização do arranjo.

A entrada de dados é feita utilizando-se uma interface de interação com o usuário (Figura 14) onde é possível configurar uma condição inicial para o sistema modificando-se os valores dos parâmetros da Equação 2.1. Além disso, o usuário tem liberdade para modificar o tamanho da grade, mudar a orientação do campo externo aplicado ao sistema, habilitar e desabilitar as linhas da grade, iluminação do vetor de *spins* e a visualização dos gráficos. O usuário também escolhe o modo de simulação (Seqüencial, Multithread,

Figura 12: Gráfico que informa a média das energias e seu somatório total

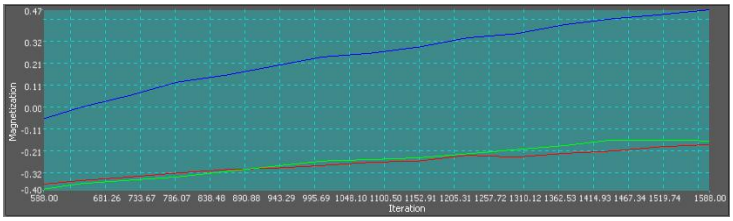

Figura 13: Desenvolvimento da magnetização ao longo do número de iterações concluidas. Cada linha indica como ela evolui pelos três eixos coordenados (x,y,z)

CUDA). As modificações dos parâmetros podem ser feitas a qualquer momento no decorrer da simulação.

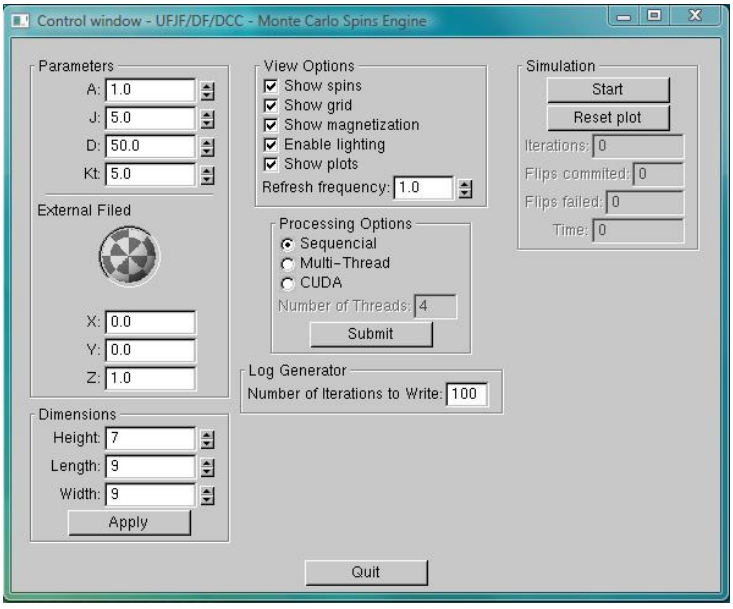

Figura 14: Interface da MCSE

4.2 Núcleo de Simulação

O Algoritmo de Metropolis, o Método de Monte Carlo e a Equação 2.1 são implementados nesta parte da MCSE. O núcleo de simulação apresenta três abordagens diferentes para a simulação dos *spins* chamadas de módulos. Os módulos de simulação serão descritos nas subseções a seguir. As operações são realizadas sobre uma estrutura de dados comum aos três módulos denominada *matrix*, que corresponde à grade tridimensional mostrada pela saída gráfica. Os *spins* constituem os elementos da grade, onde uma posição da matrix corresponde a um *spin*. Para cada *spin*, são armazenados os valores de energia, posicionamento na grade (coordenadas x, y e z), orientação espacial, dada pelos valores de um vetor tridimensional chamado *spin*, e se ele faz parte do objeto.

4.2.1 Módulo Sequencial

Este é o módulo mais simples da MCSE, pois sua implementação segue o modelo de programação tradicional, no qual apenas uma instrução é executada por vez em seqüência. A expansão para os módulos restantes parte da implementação deste módulo.

Os passos do Algoritmo de Metropolis, do item (b) em diante, descrito no Capítulo 2 estão divididos nas seguintes funções: *flipRandomParticle*, *computeEnergy* e *sequentialIterate*.

O Algoritmo inicia sua execução quando o usuário ajusta as condições iniciais do sistema via interface gráfica (item (a)), escolhe o modo de simulação (neste caso, o seqüencial) e pressiona o botão *Start*, dando início à simulação. O método principal inicializa a matrix de acordo com o valor dos parâmetros e repassa a estrutura atualizada aos três módulos. Logo após, a função *sequentialIterate*, chamada de *multithreadIterate* e *cudaIterate* nos módulos multithread e CUDA respectivamente é acionada. Esta é a função responsável pelo laço formado pelos passos (b) até (e). O Algoritmo avança para os passos (b) e (c) através da execução da função *flipRandomParticle*.

Neste ponto, há a escolha aleatória de um valor para cada um eixos coordenados, utilizando-se o algoritmo de geração de números aleatórios Mersenne Twister. Estes valores constituem o índice da partícula cujo valor de *spin* será modificado (passo (b)). Feito isso, verifica-se se o índice escolhido é válido, ou seja, se ele faz parte do objeto. Caso esta condição não se verifique, um novo índice é sorteado. Este procedimento é feito até que seja encontrado um índice. Escolhida a partícula, os valores de orientação do spin são sorteados aleatoriamente e armazenados no vetor spin (passo (c)). No entanto,

o valor antigo fica armazenado em uma variável chamada *flippedSpin*, pois caso este spin leve a formação de uma configuração incorreta do sistema, o valor antigo é re-atribuído à partícula.

O passo (d) é realizado pela função *computeEnergy* que é responsável pelo cálculo da energia total do sistema. A função é subdividida de modo que cada parte esteja ligada ao cálculo de uma das energias potenciais de interação descritas na seção 2.2. O que ocorre então é que a função *computeEnergy* percorre a grade através de um laço e a cada iteração a energia individual da partícula correspondente a essa iteração é calculada pelas funções, chamadas de *computeDipoleDipoleEnergy* e *computeMagneticFactor*. Ambas as funções recebem como parâmetros o índice da partícula acessada pelo laço da função *computeEnergy*, chamada de partícula fixada. A primeira função calcula o fator da energia de interação dipolar magnética, Equação 2.2 desta partícula em relação a todas as outras partículas do sistema. Já a segunda função, acessa apenas as partículas vizinhas à partícula fixada, calculando a energia de interação ferromagnética (Equação 2.3). As coordenadas da vizinhança são calculadas pela soma dos índices da partícula fixada com os respectivos vetores de vizinhança neighborX, neighborY e neighborZ. Por último, a energia de interação com o campo magnético externo, segundo termo da Equação 2.3 é calculada diretamente dentro do laço da função *computeEnergy*. O novo resultado para a energia individual é armazenado na matrix e adicionado à energia total do sistema.

Com o resultado da energia total do sistema calculada no passo anterior, segue-se para a etapa (e) do algoritmo de Metropolis e posteriormente o resultado final é transmitido para a interface gráfica pela função *callback*.

4.2.2 Módulo Multithread

O módulo Multithread é a primeira expansão do módulo anterior sendo o primeiro a empregar uma abordagem de computação paralela para a execução do Algoritmo de Metropolis. A idéia do módulo é dividir o processamento numérico entre várias *threads* para que ele seja efetuado mais rapidamente e explore ao máximo os recursos disponíveis na CPU. O módulo foi criado com o objetivo de dar suporte a máquinas com múltiplas CPUs. O fato de uma máquina possuir múltiplas CPUs se refere tanto ao caso em que uma placa-mãe acomoda mais de uma CPU (máquina do tipo multiprocessador) quanto ao caso em que a CPU possui mais de um núcleo de processamento (processadores do tipo multicore). Vale ressaltar que uma única máquina pode ser dos dois tipos simultaneamente caso possua mais de uma CPU multicore instalada na mesma placa-mãe.

Uma *thread* (ou linha de execuções em português) é definida como um conjunto de instruções que pode ser programado de forma independente. Tanto em máquinas multiprocessadoras quanto em máquinas multicores as linhas de execução (*threads*) podem ser realizadas de forma simultânea, isto é em paralelo.

O módulo Multithread utiliza a biblioteca POSIX Pthreads para a implementação do Algoritmo de Metropolis. Esta biblioteca trata-se de um conjunto de procedimentos e tipos definidos para se programar de forma paralela na linguagem C.

A principal mudança em relação ao módulo Seqüencial acontece na função computeEnergy. Foi adicionada ao módulo uma nova função denominada *computeEnergyThread* que corresponde à função executada simultaneamente por cada *thread*. A configuração do número de *threads* depende do número de CPUs disponíveis na máquina onde cada *thread* é atribuída a uma CPU para sua execução. A estrutura matrix foi subdividida entre as *threads* de maneira que cada uma calcule a energia de todas as partículas pertencentes a um subconjunto da grade. Neste caso, cada *thread* acessa um plano da grade de cada vez e o acesso aos planos é controlado por uma variável compartilhada denominada *planes*, responsável pelo mecanismo de *lock*/*unlock*, evitando a concorrência das *threads* por um mesmo plano.

Ao iniciar o passo (d) do algoritmo, as *threads* são criadas utilizando-se os procedimentos da biblioteca Pthreads e iniciadas. A partir daí passam a executar a função *computeEnergyThread*. Nesta função a *thread* verifica qual o número do plano corrente armazenado em planes e usa este valor como índice local. Em seguida, o número do plano é incrementado em um e a variável é liberada para o acesso das *threads* restantes. Feito isso, a função calcula a energia das partículas de seu subconjunto empregando o mesmo raciocínio do módulo Seqüencial e armazena o resultado como informação da própria *thread*.

Quando todas as *threads* terminam a execução do cálculo de energia, a energia total é atualizada somando-se cada parcela de energia obtida pelas *threads*. A Figura 15 mostra basicamente como ocorre a execução da função em paralelo.

4.2.3 Módulo CUDA

A segunda estratégia adotada para paralelizar o cálculo da Energia de Interação Potencial entre os spins foi o uso do CUDA. Conforme visto no capítulo anterior esta arquitetura é uma ferramenta atraente para resolução de problemas que exigem muitas operações

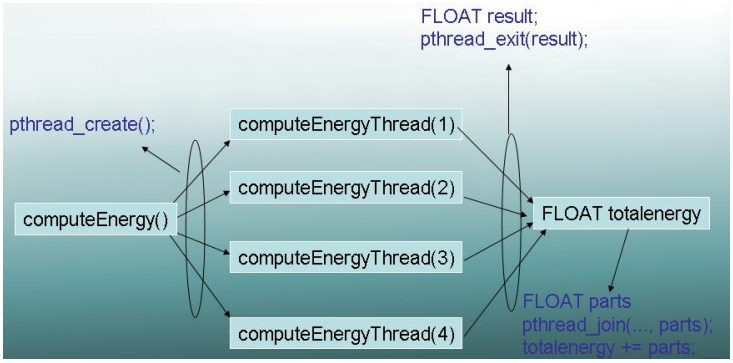

Figura 15: Esquema de funcionamento da biblioteca PThreads no módulo Multithread

aritméticas sobre um volume de dados e tem sido muito utilizada em aplicações de perfil científico. A obtenção da energia do sistema simulado pela MCSE também se enquadra nesta categoria.

O módulo CUDA aborda o problema do cálculo da energia do sistema de forma completamente diferente do módulo Multithread. Alguns aspectos da programação em PThreads, invisíveis ao programador, passam a ser relevantes para a modelagem do problema em CUDA. Os pontos mais delicados para a implementação deste módulo são: a divisão da estrutura de dados entre as *threads* utilizadas pelo CUDA, o controle dos recursos disponíveis na placa gráfica, e a maneira como os dados serão atualizados para a CPU.

Novamente, as alterações caem sobre a função *computeEnergy*. As funções *computeDipoleDipoleEnergyDevice* e *computeMagnecticFactorDevice* que calculam cada fator da Equação 2.1 em separado utilizarão a GPU e serão chamadas por *computeEnergy*. A segunda diferença da função em relação aos outros dois módulos está na duplicação de alguns dados. Todos os dados que são modificados pela GPU precisam ter uma cópia de uso exclusivo do dispositivo. As variáveis destinadas à cópia dos dados são alocadas na memória global de vídeo pelo CUDA. Em seguida, o conteúdo da variável original na CPU é transferido para a variável de cópia na GPU. Por último, a memória global de vídeo é liberada. Este procedimento é feito a cada chamada da função *computeEnergy*.

Ao contrário do módulo Mulithread, o número de *threads* designadas para a computação da energia fornecida pela Equação 2.2 é fixo. A escolha deste número foi feita levando-se em consideração o número máximo de *threads* em que a MCSE foi executada sem causar falhas no driver da placa gráfica. Ao todo são 16384 *threads* divididas em um grid contendo 64 blocos de 256 *threads* cada um. O *kernel* que implementa a Equação 2.3 utiliza apenas uma *thread* para a computação dos resultados visto que as operações são

relativamente simples.

Após a devida configuração dos dados e dos recursos, os *kernels* podem ser executados. O *kernel computeMagneticFactorDevice* é executado seqüencialmente visto que ele utiliza apenas uma *thread*. Uma diferença em relação aos demais módulos é o fato de que o *kernel* calcula o fator ferromagnético e o fator do campo externo. Além disso, esta parcela de energia é adicionada a energia individual de cada elemento somente após o calculo do fator dipolo-dipolo foi feito para a grade inteira. Nos outros módulos todas as parcelas de energia relativas a cada elemento eram calculadas e a energia final do elemento era atualizada no momento da iteração relativa ele. Já o *kernel computeDipoleDipoleEnergyDevice* utiliza a configuração de 16384 *threads*. As *threads* deste *kernel* são responsáveis por manipular um ou mais elementos dependendo das dimensões da grade. No caso de a grade ter até 16384 elementos haverá uma *thread* para cada elemento. Do contrário, haverá um laço onde os elementos que satisfazem a condição indiceDaThread = numeroDeElementos mod indiceDoElemento serão manipulados pela mesma *thread*. Relembrando o capítulo anterior, o número máximo de *threads* executadas simultaneamente pela GPU é 128. Isto significa que para cada elemento fixo, a GPU calcula a energia deste elemento em relação a outros 128 em paralelo. As Figuras 16 e **??** mostram como seria a manipulação de uma matriz com 4 linhas e 2 colunas.

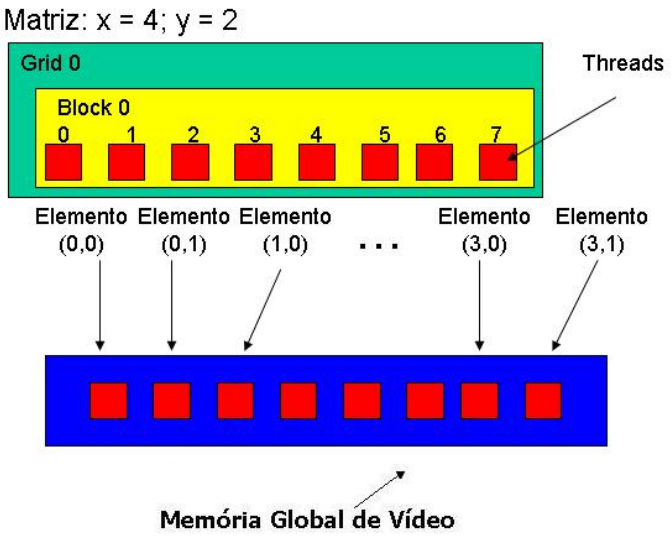

Figura 16: Correspondência entre *threads* e dados

A última diferença é a atualização da energia total do sistema. Foi visto que o módulo Seqüencial faz esta atualização a cada iteração e o módulo Multithread soma os resultados parciais obtidos por cada *thread* após o término de sua execução. A atualização podia ser feita da mesma forma que o módulo Seqüencial. No entanto, a transferência de dados

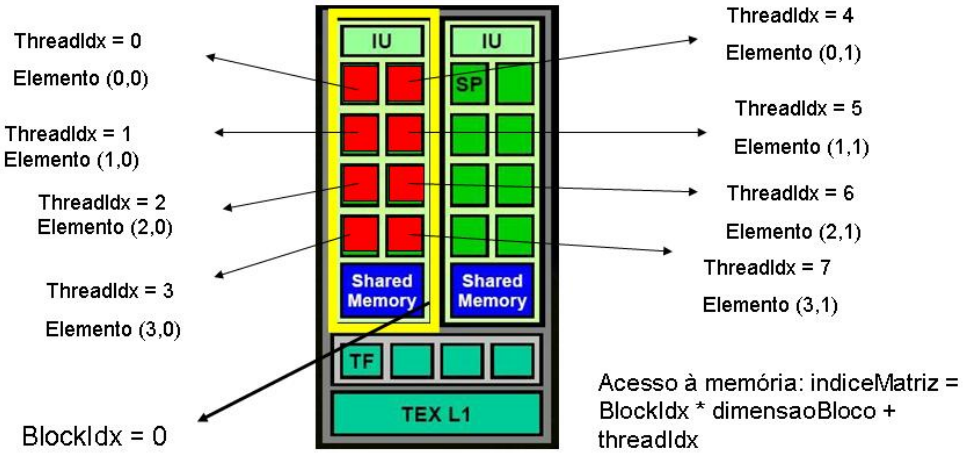

Figura 17: Divisão da Matriz-Exemplo entre os *Stream Processors*

entre a memória global de vídeo e a memória principal é bastante lenta, o que diminuiria drasticamente o desempenho da aplicação. A solução utilizada então foi esperar a execução dos dois *kernels* para todos os elementos da grade, copiar a estrutura matrix de volta para a CPU e somar as energias de cada elemento através de um laço.

4.3 Discussão de Resultados

Esta seção tem como objetivo mostrar os resultados iniciais obtidos para os três módulos da Monte Carlo Spins Engine. Os testes foram realizados nas máquinas listadas na Figura 18 executando-se duas versões distintas da aplicação. A primeira versão foi desenvolvida na plataforma 32-bits tendo somente os módulos Seqüencial e Multithread implementados. O módulo CUDA foi desenvolvido através dos recursos oferecidos para a plataforma 64-bits. Logo, o módulo não faz parte dessa versão devido à sua incompatibilidade com os demais módulos. A segunda versão foi desenvolvida implementando-se os três módulos na plataforma 64-bits.

	Máquina 1	**Máquina 2**
Processador	Intel Xeon E5410 2.33 GHz	Intel Core 2 Quad Q6600 2.4 GHz
Memória RAM	4 GB	4GB
Placa Gráfica	NVIDIA GeForce 8800 GTX	NVIDIA GeForce 8800 GTX
Sistema Operacional	Windows Vista Ultimate 64-bits	Windows Vista Ultimate 64-bits

Figura 18: Ambiente de Teste.

A aplicação foi testada modificando-se os valores do tamanho da grade e o módulo

de simulação em ambas as versões. As configurações escolhidas foram iguais para as duas máquinas de teste.A análise dos resultados foi feita levando-se em conta o número de iterações por segundo (it/s).

Os resultados iniciais para cada arquitetura podem ser observados na Figura 19.

	Máquina 1		Máquina 2	
	32-bits	64-bits	32-bits	64-bits
Seqüencial	11 it/s	14 it/s	9,5 it/s	14 it/s
Multithread	39 it/s	42 it/s	34 it/s	51 it/s
CUDA	–	13 it/s	–	13 it/s

Figura 19: Resultados experimentais em it/s.

Os resultados confirmam que o desempenho alcançado pelo módulo Multihtread é maior em relação ao módulo Seqüencial quando o número de spins é aumentado. No entanto, os resultados esperados para o módulo CUDA ainda estão abaixo do desejado.

O baixo desempenho pode ser atribuído primeiramente à forma como os dados são manipulados na GPU. Todas as variáveis que residem na GPU têm de ser alocadas antes da chamada aos *kernels* e desalocadas após a execução destes. Estas operações acabam acrescentando um tempo extra de execução, pois são realizadas de maneira excessiva pela aplicação.

Em segundo lugar, o custo da transferência de dados entre a memória global de vídeo e a memória principal também se torna um gargalo para a aplicação. Neste caso, o problema esta na diferença de largura de banda entre as memórias. A taxa alcançada na transferência de dados da CPU para GPU chega a aproximadamente 1.3 GB/s, enquanto a taxa medida para o sentido inverso é de aproximadamente 950 MB/s a largura.

O último fator a contribuir com o baixo desempenho do módulo CUDA é o acesso freqüente dos *kernels* às variáveis alocadas na memória global de vídeo. De acordo com (BELLEMAN; BÉDORF; PORTEGIES, 2008), esta operação leva de 400 a 600 ciclos para ser realizada.

Além do desempenho, foram observados os valores individuais de energia dos spins. Observou-se uma diferença nos resultados causada pelo desvio na precisão numérica que a placa gráfica possui.

5 Conclusões

Observa-se que os fenômenos magnéticos impulsionam a evolução do setor tecnológico, especialmente na área da informática. À medida que ele avança, as pesquisas se aprofundam e as experimentações adquirem maior complexidade. Por isso, a simulação computacional tem grande importância neste contexto.

Com este trabalho, foi possível desenvolver uma ferramenta bastante útil ao estudo e a análise da magnetização nos materiais ferromagnéticos. O aprimoramento da Monte Carlo Spins Engine trouxe como forte característica a flexibilidade da aplicação. Os três módulos, Sequencial, Multithread e CUDA, se adaptam às diferentes arquiteturas de computadores conhecidas, obtendo a máxima utilização dos recursos disponibilizados por um computador.

A tecnologia CUDA tem se mostrado muito eficiente para as aplicações científicas em diversas áreas de conhecimento como a física, a biologia e a matemática, por exemplo. No entanto, os resultados apresentados pela MCSE, por hora, demonstram o oposto. Isto se deve à dificuldade na manipulação dos dados e no controle das memórias RAM e de vídeo pela GPU e pela dificuldade de adaptação a esta nova abordagem da computação paralela. As falhas apresentadas tanto pelo hardware gráfico quanto pelo ambiente de programação em CUDA precisam ser corrigidas para se tornarem mais amigáveis ao programador.

Mesmo com estes problemas, a utilização do CUDA é viável haja vista a relacão positiva entre custo e benefício que ela oferece. Futuramente, pretende-se melhorar o módulo CUDA da MCSE, corrigindo os erros de precisão apontados e diminuindo as transferências de dados entre a memória RAM e a de vídeo, fato que contribuiu para que o desempenho deste módulo não alcançasse as expectativas previstas. É importante também garantir a portabilidade da aplicação, desenvolvendo-se novas versões compatíveis com outros sistemas operacionais. E por fim aumentar a sua flexibilidade possibilitando sua execução em ambientes distribuídos.

Referências

ALDER, B. J.; WAINWRIGHT, T. E. Studies in molecular dynamics. i. general method. *The Journal of Chemical Physics*, AIP, v. 31, n. 2, 1959. Disponível em: <http://dx.doi.org/10.1063/1.1673845>.

BELLEMAN, R. G.; BÉDORF, J.; PORTEGIES. High performance direct gravitational n-body simulations on graphics processing units ii: An implementation in cuda. *New Astronomy*, v. 13, n. 2, p. 103–112, February 2008. Disponível em: <http://dx.doi.org/10.1016/j.newast.2007.07.004>.

BOLITHO, M. Lecture 3: Introduction to cuda. 2008. Disponível em: <http://www.cs.jhu.edu/ bolitho/Teaching/GPGPU/Lectures/Lecture3.pdf>.

DESPRÈS, P. et al. Stream processors: a new platform for monte carlo calculations. *J. Phys.: Conf. Ser.*, Institute of Physics Publishing, v. 102, n. 1, 2008. ISSN 1742-6596. Disponível em: <http://dx.doi.org/10.1088/1742-6596/102/1/012007>.

FERNANDES, A. R. Glsl - introdução e programação da aplicação opengl. 2007. Disponível em: <sim.di.uminho.pt/disciplinas/cg/0607/at/cg05-introShaders.pdf>.

FERNANDES, F. M. S. S.; RAMALHO, J. P. P. Simulação computacional. i.fundamentos do método de monte carlo. *Revista da Ciência*, 1989.

GRZYBOWSKI, A.; GWOZDZ, E.; BRODKA, A. Ewald summation of electrostatic interactions in molecular dynamics of a three-dimensional system with periodicity in two directions. *Physical Review B*, v. 61, n. 10, 2000.

HIMAWAN, B.; VACHHARAJANI, M. Deconstructing hardware usage for general purpose computation on gpus. 2006.

KNOBEL, M. Aplicações do magnetismo. *Ciência Hoje*, v. 36, n. 215, 2005.

KOCH, G. Discovering multi-core: Extending the benefits of moore's law. *Technology@Intel Magazine*, 2005.

NVIDIA. *NVIDIA GeForce 8800 GPU Architecture Overview*. [S.l.], 2006.

NVIDIA. *NVIDIA CUDA Programming Guide*. 1.1. ed. [S.l.], 2007.

OWENS, J. D. et al. Gpu computing. *Proceedings of the IEEE*, v. 96, n. 5, p. 879–899, 2008. Disponível em: <http://dx.doi.org/10.1109/JPROC.2008.917757>.

SENA FILHO, R. F. *Teoria Microscópica de Ondas de Spin em Nanofios Magnéticos*. Dissertação (Mestrado) — Universidade Federal do Ceará, 2007.