Rafael Barra de Almeida

Paralelização Utilizando Múltiplas GPUs de Um Simulador de Elementos e Compostos Magnéticos Baseado no Método de Monte Carlo

Juiz de Fora

2010

Rafael Barra de Almeida

Paralelização Utilizando Múltiplas GPUs de Um Simulador de Elementos e Compostos Magnéticos Baseado no Método de Monte Carlo

Orientador:
Marcelo Lobosco

Co-orientadores:
Marcelo Bernardes Vieira
Sócrates de Oliveira Dantas

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

Juiz de Fora

2010

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.

Prof. Marcelo Lobosco, D. Sc.
Orientador

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

Prof. Sócrates de Oliveira Dantas, D. Sc.
Co-Orientador

Agradecimentos

Aos meus colegas do Grupo de Computação Gráfica pela compreensão durante a fase de testes, em especial a Alessandra Matos Campos e João Paulo Peçanha Navarro de Oliveira pela grande ajuda.

Aos orientadores desse projeto pelo tempo e paciência dedicados.

Agradeço também ao CNPq pelo auxílio financeiro e à FAPEMIG pelos equipamentos disponibilizados.

Sumário

Lista de Figuras

Resumo

Os fenômenos magnéticos tem sido amplamente utilizados no desenvolvimento de novas tecnologias, sendo assim é muito importante uma boa compreensão destes. O estudo desses fenômenos é facilitado com a utilização de modelos físicos que podem ser simulados computacionalmente com o objetivo de analisar o comportamento de alguns materiais. Porém, a complexidade é um grande problema na resolução matemática desses modelos. Devido a isso, esse trabalho apresenta um modelo computacional paralelo e distribuído, baseado na utilização de placas gráficas(GPUs) que visa reduzir o tempo gasto com a simulação. Com os resultados obtidos nos testes observou-se que o modelo é capaz de resolver o modelo físico, porém são necessários alguns aperfeiçoamentos para evitar problemas referentes à comunicação, que são característicos desse tipo de ambiente.

Palavras-chave: Simulação Computacional, paralelização, multi-GPU, Compostos Ferromagnéticos, Simulação Distribuída

1 Introdução

Os fenômenos magnéticos são muito presentes no nosso dia a dia. Tais fenômenos são amplamente utilizados no desenvolvimento de novas tecnologias, com aplicações em várias áreas, dentre elas a geração e distribuição de energia, armazenamento de dados, transportes e telecomunicações. Por isso um melhor entendimento das propriedades magnéticas dos materiais é de extrema importância.

A origem do magnetismo está associada a duas importantes propriedades dos elétrons em escala atômica: o momento angular orbital e o momento de *spin*. Sempre que os átomos estão próximos e sob a ação de um campo magnético, eles interagem magneticamente e podem formar estruturas em nanoescala.

Com a modelagem computacional podemos estudar melhor os fenômenos magnéticos. Lançando mão do uso de simuladores podemos analisar com mais clareza os modelos, tendo a possibilidade de testar, por exemplo, novas hipóteses sem que sistemas físicos precisem ser construídos. Cientistas podem usar o simulador para criar ou modificar diferentes propriedades físicas de um sistema magnético, gerando dados visuais e numéricos que irão ajudar a entender os processos físicos associados ao fenômeno magnético.

Em trabalhos anteriores, foi desenvolvida uma versão sequencial do simulador, chamada MCSE(*Monte Carlo Spin Engine*) (PEÇANHA et al., 2009). Seu objetivo principal é prover uma ferramenta para analisar o comportamento dos *spins* de um objeto sob a ação de um campo magnético externo uniforme. Essa versão simula *spins* em estruturas tridimensionais genéricas formadas por átomos com propriedades magnéticas. Porém, com essa versão, notamos que a simulação do modelo envolve um alto custo computacional, diretamente ligado a quantidade de elementos presentes no sistema simulado.

1.1 Definição do Problema

A interação entre os *spins* ocorre de uma maneira similar ao problema dos n-corpos, sendo assim a complexidade desse problema é igual a $\boldsymbol{O}(n^2)$, onde n é o número de *spins* presentes no sistema. Deste modo, quanto maior o número de *spins* simulados, maior o custo computacional, em termos de tempo, da simulação. Devido a essa complexidade, surgiu a necessidade de desenvolver uma versão paralela do simulador que nos permitisse trabalhar com sistemas maiores, e que portanto se aproximam mais da realidade, ao mesmo tempo em que o impacto deste aumento no tamanho do sistema, em termos de tempo de computação, fosse reduzido. Com tal objetivo em mente, foi desenvolvida uma versão paralela do simulador (CAMPOS, 2008) utilizando unidades gráficas de propósito geral (*GPGPUs*). Com essa versão foi possível alcançar speedups maiores que 100.

Mas, apesar do grande aumento de desempenho alcançado pela versão paralela, ainda há a necessidade de se aumentar a quantidade de *spins* simulados pelo sistema, de modo a torná-lo ainda mais próximo ao número de *spins* encontrados em um sistema real. Assim, propomos uma nova versão do simulador paralelo. Nesta nova versão, mais GPGPUs serão utilizadas, ainda que estas se encontrem presentes em distintos computadores. Portanto, a idéia é que o processamento seja distribuído entre as GPGPUs, de modo que elas operem sobre partes distintas do conjunto de dados. Para realizar a comunicação entre as distintas máquinas, uma rede de comunicação será utilizada.

1.2 Objetivos

O objetivo principal desta monografia é apresentar uma ferramenta que seja capaz de simular o comportamento de uma estrutura molecular qualquer e calcular a energia produzida pela interação dos *spins* dessa estrutura utilizando as GPGPUs de vários computadores. Para alcançar tal objetivo utilizaremos as tecnologias CUDA e MPI, que nos permitem controlar o processamento nas GPGPUs e distribuir os trabalhos entre várias máquinas ligadas a uma rede local, respectivamente.

2 Modelo Físico

Na física, o magnetismo é a área que estuda as propriedades dos materiais magnéticos e os fenômenos pelos quais tais materiais têm a capacidade de influenciar outros, atraindo ou repelindo. Além disso, estuda o comportamento dos materiais quando expostos a um campo magnético externo ou quando já possuem algum magnetismo intrínseco.

Um bom entendimento das propriedades magnéticas dos materiais e seus fenômenos é de grande importância para o desenvolvimento de novas tecnologias, como geração e distribuição de energia, armazenamento de dados, transportes, telecomunicações e até mesmo na medicina.

A origem do magnetismo está associada a duas importantes propriedades dos elétrons em escala atômica: a) o momento angular, associado ao movimento dos elétrons ao redor do núcleo atômico e b) o momento de *spin*, que refere-se às possíveis orientações que as partículas subatômicas possuem quando estão sob ação de um campo magnético externo. Sempre que os átomos estão próximos e sob a ação de um campo magnético externo, eles interagem magneticamente e podem formar estruturas em escalas nanométricas.

Para dar uma visão geral sobre o tema, nesse capítulo iremos definir momento de *spin*, discutir algumas fases magnéticas dos materiais como o diamagnetismo, paramagnetismo, ferromagnetismo e antiferromagnetismo, além de apresentar detalhes sobre a simulação computacional do sistema e do método de Monte Carlo.

2.1 Momento de Spin

O *spin* de um átomo, definido por um vetor tridimensional, é utilizado para designar uma propriedade intrínseca das particulas elementares. Surgiu inicialmente para descrever um estado de rotação das partículas, que originava um campo magnético em torno de si. Porém, hoje em dia é encarado como um fenômeno exclusivamente quântico, do qual não é possível fazer uma interpretação clássica, deixando essa definição sem sentido. Contudo,

na falta de um termo mais apropriado para definí-lo, continua a ser utilizada a rotação.

2.2 Fases Magnéticas

As fases magnéticas podem ser classificadas de acordo com as origens microscópicas de sua magnetização e conforme seu nível de magnetização quando exposto a um campo magnético externo, chamada susceptibilidade magnética $\chi_m = M/B$ (onde B é o campo externo e M a grandeza macroscópica que representa a magnetização do um materia) (RIBEIRO, 2000). A seguir serão discutidos os principais tipos de fases magnéticas que são diamagnetismo, paramagnetismo, ferromagnetismo e antiferromagnetismo.

2.2.1 Diamagnetismo

O diamagnetismo é caracterizado pela suscebilidade negativa $\chi_m < 0$. Esse valor ocorre devido ao campo contrário criado no material opondo-se à variação de um campo magnético externo ao qual o material está submetido. Esse efeito está presente em todo material, porém somente é percebido quando não há outros comportamentos magnéticos o sobrepondo.

Materiais diamagnéticos não possuem campo magnético intrínseco, ele é induzido pelo campo magnético externo.

2.2.2 Paramagnetismo

Materiais paramagnéticos possuem momentos magnéticos instrínsecos não interagentes entre si, ou seja, quando esse tipo de material não estiver exposto a um campo magnético externo, sua magnetização é nula. Esses materiais tem susceptibilidade positiva com ordem de grandeza entre 10^{-5} e 10^{-3}. Quando um campo externo é aplicado ao material, os dipólos tendem a se alinhar na direção do campo, porém, essa tendência encontra forte oposição na agitação térmica, sendo assim a susceptibilidade paramagnética terá grande depêndencia da temperatura (T), diminuindo quando T aumenta.

2.2.3 Ferromagnetismo

Os ferromagnetos são materiais que apresentam grande susceptibilidade magnética, temos como exemplos desse tipo de material o ferro, cobalto e o níquel. A principal dife-

rença entre os ferromagnéticos e os paramagnéticos está no fato de que sua magnetização interna não dependente da existência de um campo externo, possuem momentos de dipolo magnético intrínsecos fortemente interagentes que se alinham paralelamente entre si.

O comportamento dos ferromagnetos depende, além do material, do tratamento térmico e magnético aos quais foi previamente submetido. Devido a isso diz-se que esse tipo de material possui uma memória. Uma aplicação dessa memória é a gravação de mídia magnética, memórias permanentes, dispositivos eletrônicos entre outras. Por isso a grande importância desse tipo de material no nosso estudo.

Essas características de um ferromagnéto se mantêm até uma certa temperatura, chamada **temperatura de Curie**. A partir dessa temperatura o alinhamento interno se quebra e os *spins* tomam direções aleatórias, descaracterizando o comportamento de um material ferromágnetico. A partir desse momento, o material passa a se comportar como um paramagnético. A esse fenômeno damos o nome de transição de fase(quando um material passa a se comportar como outro).

Heisenberg em 1927 propôs uma explicação que possibilitou um melhor entendimento do campo interno dos ferromagnetos. Seus estudos concluíram que o ferromagnetismo é em grande parte devido à interação entre os *spins* dos elétrons. Assim, somados, o número de *spins* por unidade de volume do material determina a sua magnetização total.

2.3 Simulação Computacional do Modelo e o Método de Monte Carlo

As duas principais técnicas de simulação utilizadas para sistemas moleculares são: a) o método de dinâmica molecular e b) o método de Monte Carlo (FERNANDES; RAMALHO, 1989b). A dinâmica molecular geralmente é usada quando há interesse em estudar o comportamento mecânico do sistema; já o método de Monte Carlo é usado quando o estudo se foca na análise comportamental do sistema, sem se importar com o fator mecânico.

A mecânica estatística geralmente emprega o método da dinâmica molecular na solução de problemas na área, onde os sistemas possuem um grande número de entidades e são estudados a partir do comportamento coletivo dessas entidades. Alguns desses problemas são possíveis de se resolver exatamente, são os chamados problemas triviais, em que a partir de propriedades microscópicas pode-se calcular analiticamente e sem aproximações as suas propriedades macroscópicas e, assim, estudar seu comportamento. Porém existem problemas não triviais, dos quais muito poucos são exatamente solúveis e, por isso,

precisam de algum outro modelo que permita a sua resolução. O cálculo do potencial de interação entre as partículas é um exemplo de problema não trivial da mecânica estatística devido à dificuldade de se encontrar soluções analíticas para as equações que o modelam. Nesse caso, a simulação passa a ser uma poderosa ferramenta pois permite a obtenção de resultados difíceis ou até mesmo impossíveis de serem obtidos com experimentos (ALDER; WAINWRITE, 1959).

Tendo conhecimento do modelo que, com base no Hamiltoniano do sistema, permite obter as equações que o definem, podemos utilizar a simulação para evoluí-lo no tempo até se que atinja o equilíbrio. Assim conseguimos alcançar o objetivo da mecânica estatística que é transformar informações moleculares em informação termodinâmica (temperatura, energia interna, etc).

Neste trabalho será usado o método de Monte Carlo já que o foco está na análise comportamental de um sistema em termos da energia potencial entre as partículas, desconsiderando o fator mecânico. O potêncial de interação entre as partículas é obtido através das energias potênciais de interação dipolar magnética, de estado de magnetização do sistema e a de interação com o campo magnético externo. O tipo de magnetização do material também será levado em conta. A seguir, apresentaremos cada um desses termos.

2.3.1 Energias Potenciais de Interação

Sabemos que quando moléculas se aproximam, elas podem reagir ou interagir. Quando ocorre uma interação química, as moléculas podem se atrair ou se repelir, sem ocorrer quebra ou formação de novas ligações. Estas interações são chamadas interações intermoleculares.

Esse comportamento é determinado pela origem dos seus dipolos magnéticos e pela natureza da interação entre eles. Apesar de termos que considerar as contribuições de todos os elementos do sistema, a sua energia total de interação é definida na maior parte pela interação entre as partículas que estão mais próximas, visto que, quando as forças moleculares se originam do contato não reativo entre as partículas, elas variam inversamente à distância.

Nas próximas subseções serão apresentadas cada uma das energias descritas pelo hamiltoniano $\mathcal{H}$, definindo a interação dipolo-dipolo, ferromagnética e a interação com o campo magnético externo entre seus *spins*:

$$\mathcal{H} = \frac{A}{2}\sum_{\substack{i,j=1\\ i\neq j}}^{n}\omega_{ij} - J\sum_{\substack{i,k=1\\ i\neq k}}^{n}\vec{S}_i\cdot\vec{S}_k - \sum_{i=1}^{n}D(\vec{S}_i\cdot\vec{B}) \tag{2.1}$$

onde ω_{ij} é definido por:

$$\omega_{ij} = \frac{\vec{S}_i\cdot\vec{S}_j}{\left|\vec{r}_{ij}\right|^3} - 3\frac{\left(\vec{S}_i\cdot\vec{r}_{ij}\right)\left(\vec{S}_j\cdot\vec{r}_{ij}\right)}{\left|\vec{r}_{ij}\right|^5} \tag{2.2}$$

sendo $\vec{S}_i$ o clássico *spin* de Heisenberg na posição $\vec{r}_i = (x_i, y_i, z_i)$ e $\vec{r}_{ij} = \vec{r}_i - \vec{r}_j$ o vetor posição que separa os átomos i e j. As constantes A, J e D correspondem às interações dipolares, ferromagnéticas e campo externo, respectivamente.

2.3.1.1 Energia de Interação Dipolar Magnética

Chama-se de interação dipolar magnética a interação entre duas nanopartículas, decorrente da magnetização espontânea. A energia produzida por essa interação varia de acordo com a orientação e posição dos *spins* dos dipolos, separados por uma distância $\vec{r}_{ij} = \vec{r}_i - \vec{r}_j$ (GRZYBOWSKI; GWOZDZ; BRóDKA, 2000).

Em materiais ferromagnéticos, os átomos são dipolos magnéticos permanentes, ou seja, não se cancelam totalmente. Esses átomos se mantém alinhados formando um domínio ferromagnético. Desse modo, todo o material se torna também um dipolo magnético. Quando exposto a um campo magnético externo uniforme, a força resultante que atua no dipolo é nula. Nesse caso somente existirá um movimento de rotação no dipolo para fazer com que o seu momento magnético se alinhe com o campo externo. A parte da Eq.2.1 que define a energia de interação dipolar é a seguinte:

$$E_{DD} = \frac{A}{2}\sum_{\substack{i,j=1\\ i\neq j}}^{n}\left\{\frac{\vec{S}_i\cdot\vec{S}_j}{\left|\vec{r}_{ij}\right|^3} - 3\frac{\left(\vec{S}_i\cdot\vec{r}_{ij}\right)\left(\vec{S}_j\cdot\vec{r}_{ij}\right)}{\left|\vec{r}_{ij}\right|^5}\right\} \tag{2.3}$$

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

A partir da interação dos momentos magnéticos de cada partícula com o campo magnético externo e com sua vizinhança podemos obter a outra parcela da energia potencial do sistema, definida pela equação:

$$E_{FC} = -J \sum_{\substack{i,k=1 \\ i \neq k}}^{n} \vec{S}_i \cdot \vec{S}_k - \sum_{i=1}^{n} D(\vec{S}_i \cdot \vec{B}) \tag{2.4}$$

A constante J, denominada constante ferromagnética, define o tipo de interação entre os *spins*. Se positivo, representa uma interação ferromagnética, caso contrário, uma interação antiferromagnética. Com o primeiro termo da Eq.2.4 temos a energia de interação ferromagnética, sendo i uma partícula do sistema e k as partículas vizinhas a ele, com i sempre diferente de k.

O segundo termo nos dá a energia de interação entre cada *spin* com o campo magnético externo.

2.3.2 O Método de Monte Carlo

O método de Monte Carlo (FERNANDES; RAMALHO, 1989a) foi desenvolvido inicialmente por von Neumann, Ulam e Metropolis. O método consiste, basicamente, em aproximar numéricamente funções complexas, convertendo um modelo físico ou matemático em um modelo estatístico. Vários fenômenos físicos podem ser explorados com esse método, em modelos que podem ser discretizados naturalmente ou através de aproximações. Para simular tais modelos não é necessário considerar todas as suas configurações pois, através da utilização de números aleatórios e probabilidade, pode-se aproximar a função de interesse. Quanto maior o tamanho da amostra melhor a aproximação.

Um dos métodos de Monte Carlo mais usados na física atualmente é o **Algoritmo de Metropolis** desenvolvido por Nicholas Metropolis. Obtendo uma média sobre uma amostra, esse algoritmo é capaz de determinar valores esperados de propriedades do sistema simulado. É necessário que a amostra obtida siga a **distribuição de Boltzmann**. Essa distribuição define a probabilidade de um sistema em equilíbrio térmico estar num determinado estado de energia. O algoritmo irá privilegiar a energia cuja probabilidade de ocorrência no sistema seja maior. O algoritmo de Metropolis segue os seguintes passos:

1. Configuração inicial do sistema e cálculo da primeira energia.
2. Escolha aleatória de um *spin* e mudança da sua direção.
3. Calculo da nova energia do sistema.
4. Se a variação da energia(ΔE) for menor que zero, aceita a configuração e pula para o passo (7). Caso contrário, executa os passos (5) e (6).

5. Gera número aleatório A entre 0 e 1. Esse número corresponde à probabilidade do sistema estar com uma certa energia.

6. Se exp(-$\Delta E/kT$) $>$ A, então a configuração é aceita. Caso contrário, rejeita a configuração atual e volta o *spin* à posição anterior.

7. Seguir os passos (1), (2), (3), (4) até que se atinja o critério de parada definido. Cada repetição é chamada de **passo de Monte Carlo**.

A seguir é apresentado o fluxograma do algoritmo:

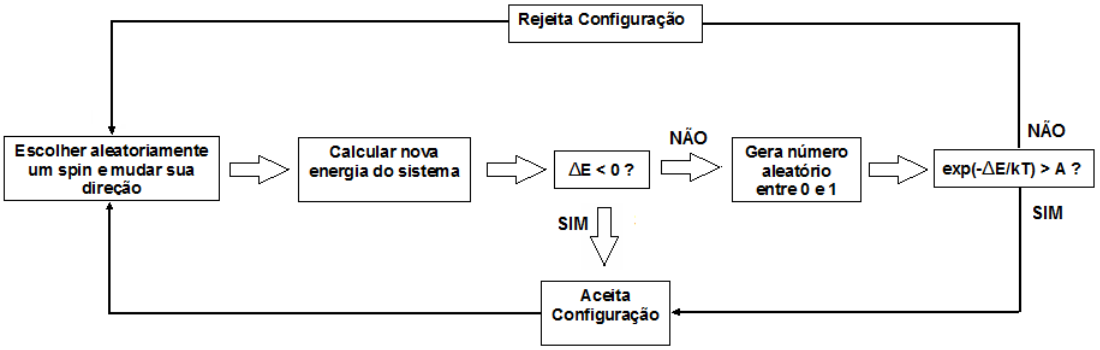

Figura 1: Fluxograma do algoritmo de Metropolis.

3 Paralelismo de dados e Técnicas Utilizadas

A computação tornou-se uma ferramenta indispensável para o desenvolvimento da ciência. Simulações computacionais permitem que uma grande gama de hipóteses seja testada *in silico*, sem que sejam necessárias, por exemplo, cobaias, montagem de sistemas físicos, ou contato com substâncias perigosas.

Neste cenário, a computação paralela é frequentemente utilizada, visto que as simulações podem necessitar que grandes volumes de dados sejam processados, o que pode levar a longos tempos de execução. Em muitas dessas aplicações é encontrado um padrão de computação bem conhecido na área de computação paralela: uma mesma computação é executada sobre cada um dos dados a serem processados de uma maneira totalmente independente, de modo que para realizar um processamento não se faz necessário a utilização de resultados de processamentos com outros dados processados anteriormente. Tal independência é chamada de paralelismo de dados (PALMER; PRINS; WESTFOLD, 1995).

Assim, paralelismo de dados se refere à possibilidade de um programa executar várias operações simultaneamente em distintas porções de dados, sem que uma operação afete os dados ou resultados de outra. A Figura 2 ilustra esse conceito, adotando uma operação de multiplicação de matrizes como exemplo. Cada elemento da matriz resultante P é gerado através do produto escalar de uma linha da matriz de entrada M e de uma coluna da matriz de entrada N. Note que as operações de produto escalar que geram cada um dos elementos de P podem ser realizados simultaneamente, ou seja, nenhum desses produtos afetará o resultado dos demais. Cada linha de M e coluna de N vão gerar um elemento de P em posição diferente.

Nesse capítulo serão apresentadas definições e conceitos importantes sobre as técnicas utilizadas na implementação computacional do modelo físico apresentado no capítulo anterior.

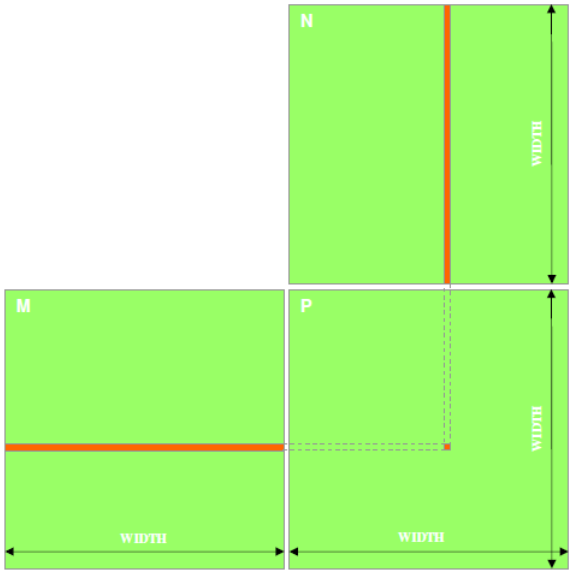

Figura 2: Paralelismo de Dados na Multiplicação de Matrizes.

3.1 Divisão dos dados

Antes de pensar em distribuir o processamento entre várias máquinas, é necessário decidir qual técnica de divisão dos dados utilizar. Os dados a serem computados representam a distribuição de *spins* em uma região do espaço tridimensional. A técnica de divisão de dados portanto está intimamente ligada a forma como o espaço será particionado. Um aspecto importante é que a técnica escolhida deve ser capaz de fazer a divisão do espaço de maneira rápida e precisa, nos permitindo um acesso eficiente aos dados. Uma das estruturas de dados mais utilizadas nesse tipo de situação são as árvores de particionamento espacial. Dessas, as mais comumente utilizadas são as as Kd-Trees, Árvores BSP e as *Octree*, que são definidas a seguir:

- Kd-tree: a kd-tree é uma estrutura de dados utilizada para o particionamento de espaço que tenta organizar pontos em um espaço de k-dimensões. A divisão dos espaço em planos tem de seguir a direção dos eixos principais (ZHOU et al., 2008). Nesse tipo de estrutura, encontrar a menor árvore possível é um problema np-completo, por isso não cabe à toda aplicação.

- Árvores BSP: as árvores BSP podem ser consideradas uma generalização das Kd-Trees, ou estas como uma especialização das BSP-Trees. A principal diferença entre ambas está no modo como são feitos os cortes; numa BSP os planos não precisam seguir a direção dos eixos principais.

Para realizar a divisão do espaço em nosso trabalho escolhemos a árvore de partição *octree*, pois esta nos permite fazer a divisão de forma rápida e sem grande custo de memória. Representamos o espaço a ser dividido como uma matriz, onde cada posição se refere a um átomo do sistema. A seguir são dados mais detalhes dessa técnica.

3.1.1 Octrees

As *Octrees*, ou *Quadtree* se o espaço considerado for bidimensional, envolvem a subdivisão recursiva do espaço em pequenas células, chamadas octantes (espaço 3D), ou quadrantes (espaço 2D). A subdivisão é feita até que se chegue numa profundidade máxima ou até que todas as suas folhas tenham alcançado a granularidade de vértices determinada (AYALA et al., 1985). Nesse trabalho, o espaço de simulação considerado é tridimensional.

Após feita toda a divisão, a raiz da árvore gerada possuirá todo o espaço, enquanto cada nó representará um sub-espaço que pode ser subdividido em mais 8 octantes caso a granularidade seja maior que a determinada, caso contrário esse nó será chamado folha (SAMET; WEBBER, 1988). Serão esses nós que armazenaram separadamente os átomos do sistema, de acordo com o escopo desse trabalho.

3.1.1.1 Construção da Octree

Para iniciar a construção da octree precisamos definir um objeto que envolva totalmente o espaço a ser dividido. Geralmente esse objeto é um cubo cujas dimensões são potências de 2, maiores ou iguais aos limites do espaço a ser dividido. A partir daí inicia-se a divisão do cubo que envolve o espaço em 8 octantes. A cada divisão feita conta-se a quantidade de átomos presentes em cada nó. Caso o nó tenha granularidade maior que a definida, esse nó é chamado de não-folha e ele é novamente dividido em 8 octantes(Figura 3). Isso é feito para todos os nós até que se chegue a um estado onde todas as folhas da árvore possuam uma quantidade de átomos menor ou igual à determinada.

Terminada a construção da árvore, inicia-se o processo de poda onde todo sub-espaço nulo (que não possui átomos) é desconsiderado. A figura 4 ilustra o critério de parada e a poda na árvore quando uma *quadtreee* é utilizada. Nela, o nó R é a raíz, os nós brancos com letra N indicam nós intermediários (não-folha). Os nós verdes numerados indicam os nós folhas e os nós em forma de quadrado pretos indicam nós nulos, sem átomos. O critério de parada neste exemplo foi ser atingida granularidade igual a 1.

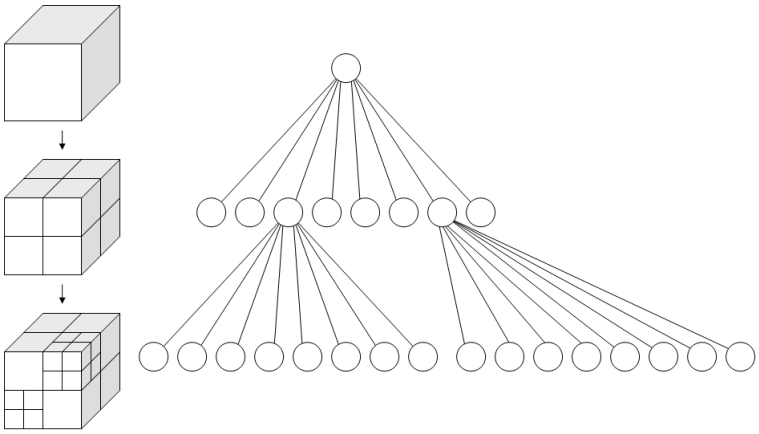

Figura 3: Construção da Octree - cada sub-espaço é sempre dividido em 8 partes.

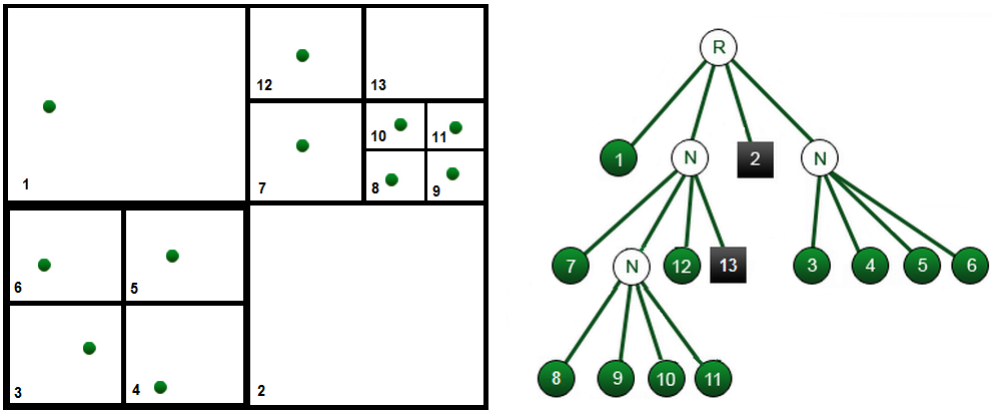

Figura 4: Representação de uma quadtree simples.

O fato de se desconsiderar as regiões nulas do espaço permite otimizar o cálculo da interação entre os *spins*, pois desse modo não perdemos tempo acessando regiões de memória que não guardam valores importantes para o cálculo.

3.2 MPI (*Message Passing Interface*)

Neste trabalho foi utilizado MPI para realizar a comunicação e a distribuição do processamento entre as máquinas. MPI é uma biblioteca de comunicação por passagem de mensagens (FORUM, 2009) muito popular no desenvolvimento de aplicações paralelas.

3.2.1 O Modelo de Passagem de Mensagem

Esse modelo 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 é usado quando os processos não compartilham memória, como quando uma arquitetura multicomputador é empregada. 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.

Ao iniciar a execução de um programa, o usuário especifica o número de processos participantes, número este que permanece durante todo o tempo de execução. Todos os processos executam o mesmo código. Porém é possível fazer com que cada processo tome um caminho diferente neste código, visto que cada processo receberá um número de identificação próprio.

3.2.2 Terminologia MPI

Essa subseção explica alguns dos termos usados ao longo desse trabalho. São eles:

- **Não Bloqueante**: um procedimento é dito não bloqueante se pode-se retornar de sua invocação antes que a operação se complete e antes que todos os processos sejam capazes de utilizar os recursos especificados na chamada.
- **Bloqueante**: um procedimento é bloqueante se retorna-se de sua invocação somente após todos os processos serem capazes de utilizar os recursos especificados na chamada.
- **Local**: um procedimento é local se sua execução total depende somente do processo local.
- **Não Local**: um procedimento é não local se sua execução total depende de algum outro procedimento MPI em outro processo. Tal operação deve precisar de algum tipo de comunicação entre os processos.
- **Coletivo**: um procedimento é coletivo se todos os processos em um grupo de processos precisam invocar o procedimento. Uma chamada coletiva pode ou não ser sincronizada e deve ser feita na mesma ordem por todos os membros que estão sob um mesmo comunicador.

- **Tipo predefinido**: é um tipo com um nome definido. A Figura 5 mostra os tipos de dados predefinidos no MPI.

Tipo MPI	Tipo C
MPI_CHAR	char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT	signed long long int
MPI_LONG_LONG (as a synonym)	signed long long int
MPI_SIGNED_CHAR	signed char
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_WCHAR	wchar_t
MPI_C_BOOL	_Bool
MPI_INT8_T	int8_t
MPI_INT16_T	int16_t
MPI_INT32_T	int32_t
MPI_INT64_T	int64_t
MPI_UINT8_T	uint8_t
MPI_UINT16_T	uint16_t
MPI_INT32_T	int32_t
MPI_INT64_T	int64_t
MPI_UINT8_T	uint8_t
MPI_UINT16_T	uint16_t
MPI_UINT32_T	uint32_t
MPI_UINT64_T	uint64_t
MPI_C_COMPLEX	float _Complex
MPI_C_FLOAT_COMPLEX (as a synonym)	float _Complex
MPI_C_DOUBLE_COMPLEX	double _Complex
MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
MPI_BYTE	
MPI_PACKED	

Figura 5: Tipos de dados MPI (ref MPI: A Message-Passing Interface Standard).

3.2.3 Diretivas de Comunicação

3.2.3.1 Comunicação ponto a ponto

O mecanismo básico de comunicação no MPI é o envio e o recebimento de mensagens. As duas operações básicas são o MPI_Send e MPI_Recv. A Figura 6 exemplifica seu uso.

No exemplo da Figura 6, os processos escravos ($rank > 0$) enviam uma mensagem ao processo zero, dito mestre, através da operação MPI_Send. Essa operação especifica um *buffer* na memória do processo emissor de onde a mensagem é retirada. O local, tamanho e tipo do *buffer* são especificados pelos três primeiros parâmetros da operação. Para definir o destino da mensagem, utiliza-se um "envelope", que contêm também informações que podem ser utilizadas pela operação **receive** para selecionar uma mensagem particular. Esse envelope é especificado pelos últimos três parâmetros da operação MPI_Send.

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc, char* argv[]) {
        int rank;               /* meu rank (ou id) */
        int tag = 0;            /* tag (ou rótulo) da mensagem */
        char message[100];
        MPI_Status status;      /* status de retorno do receive */

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        if (rank != 0) {
                sprintf(message, "Olá, Mundo! Processo no. %d!", rank);
                MPI_Send(message, strlen(message)+1, MPI_CHAR, 0, tag, MPI_COMM_WORLD);
        }
        else {
                for (source = 1; source < p; source++) {
                        MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
                        printf("%s\n", message);
                }
        }
        MPI_Finalize();
}
```

Figura 6: Exemplo de código MPI.

O processo zero (rank = 0), recebe os parâmetros através da operação MPI_Recv. A mensagem recebida é selecionada de acordo com o envelope e os dados são gravados no *buffer* do receptor. Os três primeiros parâmetros da operação definem localização, tamanho e tipo do *buffer* no receptor. Os três parâmetros seguintes são usados para selecionar a mensagem que chega e o último parâmetro é usado para retornar informação sobre o estado da mensagem que acabou de chegar.

3.2.3.2 Comunicação Coletiva

Comunicação coletiva é definida como um tipo de comunicação que envolve um grupo ou grupos de processos. As funções desse tipo providas pelo MPI e usadas nesse trabalho são:

- MPI_BARRIER: barreira de comunicação entre todos os membros de um grupo. Os processos param até que todos cheguem naquele ponto.
- MPI_BCAST: Envio de uma mensagem de um membro do grupo para todos os demais membros.
- MPI_REDUCE: operação de redução global, tal como soma, subtração, mínimo,

máximo ou funções definidas pelo usuário. Nesse caso o resultado fica acumulado somente no processo mestre.

- 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.

3.2.3.3 Compilando e Executando um Código MPI

Para compilar um programa MPI podemos usar as ferramentas *mpicc* (código em C) ou *mpicxx* (código em C++). O comando para compilação de uma forma geral tem a seguinte forma: *mpicxx nomedoprograma.c -o nomedoprograma.out*. Já para executar um programa MPI utilizamos a ferramenta *mpirun*, que dispara a execução do programa nos diversos nós. O comando básico para executar o programa é *mpirun -np X nomedoprograma.out -machinefile machinesList*, onde X é o número de processos e *machinesList* o arquivo que lista as máquinas disponíveis para o processamento (KARNIADAKIS; KIRBY, 2003).

3.3 Computação de Propósito Geral nas GPUs

A grande quantidade de unidades de processamento existentes nas GPUs (Unidades de Processamento Gráfico) atraiu a atenção da comunicade de computação paralela.

As primeiras GPUs, apesar de possuírem grande capacidade de processamento, não permitiam ao programador aproveitar plenamente essa capacidade. As funções utilizadas para realizar o processamento eram todas pré-definidas pelo fabricante, impedindo, assim, que o programador alterasse o modo como os dados eram processados. Com a evolução das GPUs e o lançamento das APIs gráficas OpenGL e DirectX, e o lançamento da placa GeForce 3, as GPUs passaram a ser programáveis, permitindo aos programadores ajustar algumas funções pré-definidas pelo fabricante para que se adequassem às suas aplicações gráficas.

Conforme evoluíam, as placas permitiam cada vez mais a interferência do programador em suas funções básicas, ao ponto de chegar a permitir a execução de aplicações não gráficas. A partir desse momento surgiu o nome GPGPU (*General Purpose Graphics Processing Unit*, ou unidade de processamento gráfico de propósito geral.

Observando esse avanço, a fabricante NVIDIA passou a investir em tecnologias que

oferecessem suporte mais adequado à GPGPU, lançando a série de placas chamada NVIDIA GeForce 8. Com uma nova arquitetura de hardware e software, essas GPUs passaram a permitir realmente que programadores desenvolvessem aplicações de uma maneira próxima à programação para CPUs. Essa arquitetura foi chamada de CUDA.

3.3.1 CUDA (*Compute Unified Device Arquitecture*)

CUDA (Arquitetura Unificada de Dispositivos de Computação) é a arquitetura de hardware e software para computação paralela desenvolvida pela NVIDIA. Com um novo modelo de computação paralela e arquitetura de conjunto de instruções, conseguiu alavancar a computação paralela nas GPUs NVIDIA para resolver vários problemas computacionais complexos de uma maneira mais eficiente que na CPU (NVIDIA, 2010).

CUDA teve como base a linguagem de programação C, o que facilitou o processo de aprendizagem do programador, por partir de uma linguagem popular e já amplamente difundida, e consequentemente permitiu que programas fossem escritos com facilidade para executar no dispositivo.

3.3.1.1 Estrutura Básica de Um Programa CUDA

O compilador C da NVIDIA (NVCC) separa bem os códigos que devem ser executados na CPU (*host*) e na GPU (*device*). O código da CPU é compilado com o compilador C padrão, enquanto o código do dispositivo é escrito usando uma extensão à linguagem C para nomear funções paralelas, chamadas *kernels*, e suas estruturas de dados associadas (KIRK; WHU, 2009).

Os *kernels* geram um grande número de *threads* para explorar o paralelismo de dados. No exemplo de multiplicação de matrizes da Figura 2, a computação da multiplicação pode ser implementada como um *kernel*, onde cada *thread* seria reponsável por calcular um elemento de saída da matriz P.

Quando um *kernel* é invocado, ele é executado como um *grid* de *threads* paralelas. Na Figura 7, quando o *kernel* 1 é invocado cria-se o *Grid* 1. Cada *grid* geralmente contêm, a cada invocação do *kernel*, de centenas a milhares de *threads*.

As *threads* em um *grid* são organizadas numa hierarquia de dois níveis, como mostrado na Figura 7. No nível mais alto, cada *grid* consiste de um ou mais blocos de *threads*. Tais blocos possuem o mesmo número de *threads*.

Cada bloco é organizado como um vetor tridimensional de *threads*, com um máximo de 512 *threads*. As coordenadas da *thread* em um bloco são únicas e definidas por três índices: *threadIdx.x*, *threadIdx.y*, e *threadIdx.z*.

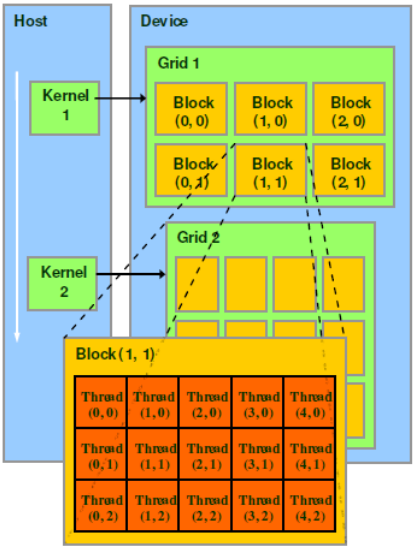

Figura 7: Organização das Threads.

A Figura 8 mostra um exemplo de código em CUDA para realizar a multiplicação de matrizes.

3.3.1.2 O Modelo de Memória CUDA

Cada dispositivo CUDA possui memórias que podem ser usadas pelos programadores para que consigam alcançar alta velocidade de execução dos seus *kernels*. A Figura 9 mostra como essas memórias são organizadas na arquitetura da placa GeForce 8800GTX. Na parte inferior da figura está localizada a memória constante e a memória global. Essas são as memórias que o *host* pode ler e escrever. O *device* possui apenas permissão de leitura na memória constante, que possui acesso mais rápido do que a memória global.

Logo acima das representações das *threads* na Figura 9 estão os registradores e as memórias compartilhadas. As variáveis que residem nessas memórias podem ser acessadas de uma maneira muito rápida. Os registradores são alocados separadamente para cada *thread* e cada uma delas pode acessar somente os seus próprios registradores. Um *kernel* geralmente usa os registradores para armazenar as variáveis da *thread* que são privativas a cada *thread*. Já as memórias compartilhadas são alocadas para cada bloco de *threads*. Todas as *threads* em um bloco podem acessar variáveis alocadas na memória compartilhada daquele bloco; sendo assim, utilizar a memória compartilhada é uma ótima

maneira das *threads* compartilharem os resultados de seus trabalhos individuais (NVIDIA, 2006).

3.3.2 Por quê usar GPGPUs e CUDA?

As GPUs atuais, com vários núcleos que dispoêm de memórias com grande largura de banda, oferecem enormes quantidades de recursos computacionais, que podem ser utilizadas tanto para processamento gráfico quanto para não gráfico. O fato de estarem disponibilizadas centenas de unidades de processamento tornam esta arquitetura altamente atraente para o desenvolvimento de aplicações paralelas. A razão custo x benefício é um outro diferencial desta arquitetura, que possui um preço bem inferior ao de um conjunto de computadores com capacidade computacional equivalente.

No projeto de desenvolvimento das GPUs optou-se por uma reduzida região de controle e cache, de modo a possibilitar que uma maior quantidade de unidades funcionais pudessem ser implementadas(Figura 10). Esse é um dos motivos que permitem o grande poder de processamento das GPUs, além da organização de memória citada anteriormente.

```
// M(row, col) = *(M.elements + row * M.stride + col)
typedef struct {
    int width;
    int height;
    int stride;
    float* elements;
} Matrix;

// Get a matrix element
__device__ float GetElement(const Matrix A, int row, int col)
{
    return A.elements[row * A.stride + col];
}
// Set a matrix element
__device__ void SetElement(Matrix A, int row, int col,float value)
{
    A.elements[row * A.stride + col] = value;
}
__device__ Matrix GetSubMatrix(Matrix A, int row, int col)
{
    Matrix Asub;
    Asub.width    = BLOCK_SIZE;
    Asub.height   = BLOCK_SIZE;
    Asub.stride   = A.stride;
    Asub.elements = &A.elements[A.stride * BLOCK_SIZE * row + BLOCK_SIZE * col]
    return Asub;
}
// Thread block size
#define BLOCK_SIZE 16
// Forward declaration of the matrix multiplication kernel
__global__ void MatMulKernel(const Matrix, const Matrix, Matrix);

void MatMul(const Matrix A, const Matrix B, Matrix C)
{
    // Load A and B to device memory
    Matrix d_A;
    d_A.width = d_A.stride = A.width; d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    cudaMalloc((void**)&d_A.elements, size);
    cudaMemcpy(d_A.elements, A.elements, size,cudaMemcpyHostToDevice);
    Matrix d_B;
    d_B.width = d_B.stride = B.width; d_B.height = B.height;
    size = B.width * B.height * sizeof(float);
    cudaMalloc((void**)&d_B.elements, size);
    cudaMemcpy(d_B.elements, B.elements, size,cudaMemcpyHostToDevice);
    Matrix d_C;
    d_C.width = d_C.stride = C.width; d_C.height = C.height;
    size = C.width * C.height * sizeof(float);
    cudaMalloc((void**)&d_C.elements, size);

    // Invoke kernel
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
    MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

    // Read C from device memory
    cudaMemcpy(C.elements, d_C.elements, size,cudaMemcpyDeviceToHost);
    cudaFree(d_A.elements);
    cudaFree(d_B.elements);
    cudaFree(d_C.elements);
}
// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    int blockRow = blockIdx.y;
    int blockCol = blockIdx.x;

    // Each thread block computes one sub-matrix Csub of C
    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);

    // Each thread computes one element of Csub
    // by accumulating results into Cvalue
    float Cvalue = 0;
    // Thread row and column within Csub
    int row = threadIdx.y;
    int col = threadIdx.x;

    // Loop over all the sub-matrices of A and B that are
    // required to compute Csub
    // Multiply each pair of sub-matrices together
    // and accumulate the results
    for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
       Matrix Asub = GetSubMatrix(A, blockRow, m);
       Matrix Bsub = GetSubMatrix(B, m, blockCol);

       // Shared memory used to store Asub and Bsub respectively
       __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
       __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

       // Load Asub and Bsub from device memory to shared memory
       // Each thread loads one element of each sub-matrix
       As[row][col] = GetElement(Asub, row, col);
       Bs[row][col] = GetElement(Bsub, row, col);
       __syncthreads();

       // Multiply Asub and Bsub together
       for (int e = 0; e < BLOCK_SIZE; ++e)
           Cvalue += As[row][e] * Bs[e][col];
        __syncthreads();
    }
    SetElement(Csub, row, col, Cvalue);
}
```

Figura 8: Exemplo de Código de Multiplicação de Matrizes.

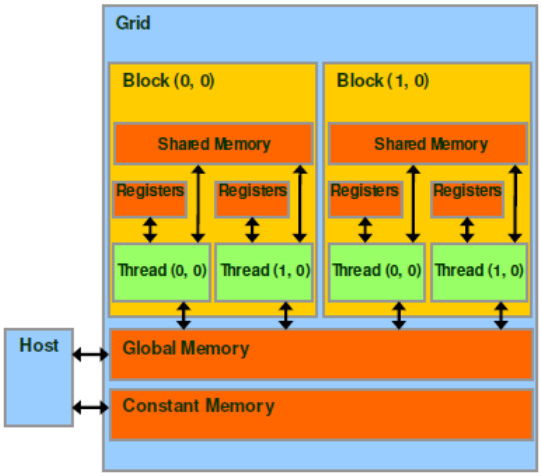

Figura 9: Organização das Memórias.

Figura 10: Comparação entre componentes de uma CPU e de uma GPU.

4 Implementação

Com a simulação computacional é possível estudar um sistema através de um modelo matemático que deve reproduzir as características do sistema original de uma maneira mais fiel possível. Manipulando o modelo e analisando os resultados, pode-se extrair várias informações relevantes ao sistema, sendo possível avaliar suas diversas propriedades. Além disso, com a simulação computacional há a possibilidade de se estudar com eficiência o comportamento de sistemas em condições que podem ser impossíveis de serem simuladas em laboratórios, tal como altas temperaturas e grande quantidade de partículas envolvidas.

Nesse capítulo serão apresentados detalhes sobre a implementação do modelo.

4.1 Configuração do Ambiente de Simulação

Toda a simulação foi realizada em computadores com sistema operacional Linux, com a distribuição Ubuntu 10.4.

Nesta etapa foram instalados o MPI e o CUDA em todas as máquinas disponíveis para o processamento. Inicialmente foi instalado o MPI. Para isso foram seguidos os seguintes passos:

- Primeiro foi necessário configurar o *SSH(Security Shell)* para que as máquinas pudessem se conectar às demais sem a necessidade de usar senha em cada conexão. A Figura 11 mostra os passos necessários para essa configuração.
- O próximo passo foi instalar o *MPI* em todas as máquinas. No Ubuntu basta usar o comando *sudo apt-get install mpich2*, que instalará todas as ferramentas necessárias para seu uso. Feito isso é necessário criar o arquivo .mpd.conf na pasta */home/usuário* em todas as máquinas(mestre e escravos), com pelo menos a linha *"secretword=xxxx"*. A seguir deve-se criar o arquivo *mpd.hosts* em cada máquina.

O arquivo deve conter o nome de todas as demais máquinas. Esses arquivos devem ter permissão total somente para o dono.

- Por fim é necessário iniciar um *daemon MPD* em cada um dos nós. Ele vai ser usado para gerenciar todos os processos MPI presentes. O mestre é o primeiro a ser iniciado. Ele serve como ponto de conexão para todos os demais nós. Para iniciar o *MPD* no mestre, basta executar o comando *mpd —daemon —ncpus=1* (o número de máquinas foi colocado igual a 1 para que cada processo execute numa máquina diferente). Nas demais máquinas basta executar o comando *mpd —daemon —host=< primeirohost > —port=< portaencontradacommpdtrace > —ncpus=1*. A partir desse momento estará configurado o "anel MPI", que é o ambiente através do qual as máquinas irão se comunicar.

```
a@A:~> ssh-keygen -t rsa
```

Agora use o `ssh` para criar um diretório `~/.ssh` como usuário b em B.

```
a@A:~> ssh b@B mkdir -p .ssh
b@B's password:
```

Finalmente copie a nova chave pública de A para `b@B:.ssh/authorized_keys` e entre com a senha de B pela última vez:

```
a@A:~> cat .ssh/id_rsa.pub | ssh b@B 'cat >> .ssh/authorized_keys'
b@B's password:
```

Figura 11: Passos para configurar SSH.

Para instalar o CUDA utilizamos os arquivos fornecidos pela NVIDIA, sendo eles: *CUDA Driver*, *CUDA Toolkit* e *CUDA SDK*. Tendo instalado todos os arquivos, basta compilar o conjunto de programas presentes no SDK.

4.2 Simulação do Modelo

O ínicio da execução do simulador se dá na chamada ao programa gerado pelo compilador, nesse momento será definido o número de processos envolvidos no anel e a cada um deles será atribuída uma das máquinas descritas no arquivo de configuração do MPI citado na seção anterior.

No processo mestre serão definidos os parâmetros de simulação, sendo eles: o tamanho e o tipo do sistema a ser simulado (cubo sólido, cilindro vazado, etc) e os valores das

constantes A, J e D que correspondem às interações dipolares, ferromagnéticas e campo externo, respectivamente.

Tendo então os parâmetros definidos, o processo mestre inicia a distribuição destes entre os demais processos pertencentes ao anel. Primeiramente envia os parâmetros A, J e D e em seguida a matriz que representa o espaço total a ser simulado. Para realizar essa distribuição, o processo mestre usa operações de comunicação bloqueantes, ou seja, ele pára a execução e espera que todos os demais tenham condições de utilizar os dados enviados para só então voltar ao processamento.

A partir do momento que um processo recebe a matriz é iniciada a divisão do espaço de simulação utilizando o método octree. Nesse trabalho optou-se por construir a octree em cada uma das máquinas disponíveis para evitar o problema de endereçamento entre as elas, visto que o espaço de endereçamento em uma máquina não faz sentido quando levado para outra. Isto se tornou um problema porque cada processo após ter a octree construída irá percorrê-la para formar listas encadeadas de spins, onde cada estrutura que representa uma partícula aponta para a seguinte na lista, assim tais ponteiros só fazem sentido na máquina onde foram criados; ao serem levados para o espaço de memória de outra máquina perderíamos o controle sobre os mesmos. Ao final essas listas conterão uma parte diferente do total de spins do sistema e posteriormente cada um dos processos irá pegar uma delas para processar. Construindo a octree em cada processo evitamos também o reenvio de dados, já que inicialmente a matriz contendo todo o espaço deve ser enviada a todos os processos.

4.2.1 Detalhes da Implementação com MPI

Uma característica interessante da implementação foi a necessidade de se definir um novo tipo de dados MPI para enviar a matriz de simulação do mestre para os escravos. Como vimos no capítulo anterior, o MPI predefine alguns tipos de dados. Nenhum desses tipos pode ser usado para transferir a matriz, já que ela é representada no código como uma estrutura que contém a posição e a direção do *spin* no sistema, além de outros valores que são utilizados na construção da *octree*. Além disso, as operações MPI_Send e MPI_Recv só podem ser usadas para enviar dados que estão em posições contíguas de memória.

Definir um novo tipo de dado MPI consiste basicamente em agrupar os dados que precisam ser enviados e a partir disso definir um novo tipo de dados derivado, onde esse novo tipo é uma sequência de pares $\{(t_0, d_0), (t_1, d_1), ..., (t_{n-1}, d_{n-1})\}$ sendo t_i um tipo de dado

MPI e d_i é uma distância em bytes. Por exemplo, $\{(MPI\_INT, 0), (MPI\_FLOAT, 4), (MPI\_CHAR, 16)\}$. Essa sequência é chamada de **mapa de tipos** e a sequência $\{(t_0), (t_1,), ..., (t_{n-1})$ }chamada de **assinatura de tipos**. O mapa de tipos junto a um endereço base, chamado *buf*, especifica um *buffer* de comunicação que consiste de n entradas, onde a i-ésima entrada está no endereço *buf* + d_i e tem o tipo t_i. Uma mensagem montada em tal *buffer* de comunicação vai consistir de n valores dos tipos definidos na assinatura de tipos. Tendo definidas essas sequências, é necessário que todos os processos reconheçam esse novo tipo de dados. Para isso, cada processo deve construir tais sequências e, em seguida, chamar a função MPI_TYPE_commit com o novo tipo. Desse modo todos os processos reconhecerão o novo tipo.

Na versão centralizada (não distribuída) do simulador, o critério de parada no cálculo das energias é o número de interações. Nesta versão do código, geralmente com 5 mil interações o sistema já atinge um estado estável. Na versão distribuída o número de interações também servirá como critério de parada. Nessa versão caberá ao processo mestre a responsabilidade de incrementar o número de interações e enviar uma *flag* para os demais processos, que servirá para sinalizar a eles se a execução deve ou não continuar. Atingindo o número máximo de interações definidas, o processo mestre envia a mensagem de parada a todos os demais.

4.2.2 Detalhes da Implementação com CUDA

Antes de iniciar a implementação da versão distribuída do simulador, foi desenvolvida uma versão centralizada em CUDA(CAMPOS, 2010). A partir dessa versão foram feitas adaptações no código para que o mesmo pudesse ser utilizado na versão distribuída. Devido a isso, essa subseção apresentará os detalhes da versão não distribuída e, em seguida, as adaptações necessárias no código para transformá-la na versão distribuída.

4.2.2.1 Versão CUDA Centralizada

Nessa versão, o CUDA foi utilizado apenas para paralelizar o cálculo da energia de interação potencial entre os *spins*.

Após ajustadas as configurações iniciais do sistema, o algoritmo inicia sua execução através da função *cudaStart*. Nesse ponto, com o intuito de otimizar as requisições de memória, a matriz que representa o espaço de simulação total foi separada em três vetores, um contendo as posições, outro com as direções e outro com as energias de cada partícula.

Desse modo podemos guardá-los separadamente na memória da GPU e, sendo assim, acessá-los de maneira mais eficiente, otimizando as transfêrencias e evitando conflitos.

Os passos do algoritmo de Metrópolis são executados pelas funções *flipRandomParticle*, *integrateSpinsSystem* e *cudaIterate*. Na função *flipRandomParticle* é escolhido um valor aleatório para cada um dos eixos das coordenadas. Estes valores definem o índice da partícula cujo valor de *spins* será modificado. Escolhido o *spin*, os valores de sua orientação serão modificados e armazenados no lugar dos anteriores. O valor antigo do *spin* é armazenado em outra variável, para o caso da nova energia não ser aceita (passo 6 do algoritmo definido no Capítulo 2). Nesse caso o valor anterior é restaurado.

Mudada a orientação da partícula, o passo seguinte é calcular a nova energia do sistema. A energia potêncial de interação entre as partículas (Eq. 2.1) é calculada pelas funções *integrateSpins2xN*, *dipoledipoleInteraction*, que calcula a energia de interação dipolo entre as partículas, e *computeMagnecticFactor*, que calcula a energia de interação ferromagnética. Na função *integrateSpins2xN* essas duas energias são somadas à energia de interação com o campo magnético externo para obter a energia total do sistema.

A energia do *spin* calculada por cada *thread* é guardada no vetor de energias, na posição identificada pelo índice único calculado em cada *thread*. Ao final, os valores são somados através de uma operação de redução paralela feita na GPU. Essa operação está descrita e implementada no algoritmo nomeado *Parallel Reduction*, disponível no conjunto de exemplos de código em C do *NVIDIA CUDA SDK*.

Uma característica importante da implementação em CUDA é o modo como as memórias da GPU são utilizadas. Inicialmente os dados eram mantidos completamente na memória global da GPU, que, embora seja grande o suficiente, é mais lenta que as outras memórias disponíveis no dispositivo, como por exemplo a memória compartilhada. Para tentar aproveitar a maior velocidade de acesso da memória compartilhada, o código foi modificado para que algumas variáveis fossem alocadas em tal memória. Porém, como a memória compartilhada possui menor capacidade de armazenamento, foi necessário dividir a matriz NxN em regiões menores, de maneira que essas regiões coubessem na memória compartilhada. Durante a execução, cada uma dessas regiões é levada para a memória compartilhada, e então é calculada a energia referente aquela região. Sua contribuição na energia total do sistema é então gravada na memória global. Isso é feito até que a energia referente a todas as regiões sejam calculadas. Ao final, a energia total estará armazenada na memória global. Essa técnica de particionamento de dados é conhecida como *tiling* (NYLAND; HARRIS; PRINS, 2004).

Uma outra importante modificação foi o modo como o cálculo da energia é efetuado. Inicialmente, a cada mudança de orientação do *spin* a energia total do sistema era calculada, ou seja, as energias dipolo-dipolo, ferromagnética e de interação com o campo magnético externo eram calculadas para cada uma das partículas. Em seguida era calculada a variação de energia, subtraindo-se a nova energia da energia anterior. Essa variação é necessária para realização dos passos (4) e (6) do algoritmo de Metropolis. Porém, como é necessário obter somente a variação da energia em cada passo de Monte Carlo, podemos fazer apenas um passo que calcule a energia inicial do sistema e depois calcular apenas a variação da energia em cada passo. Para isso, o cálculo na GPU foi modificado de modo a calcular somente a energia de interação dipolo-dipolo entre o *spin* que teve sua orientação modificada e os demais spins (Eq. 4.3). Em seguida, subtrai-se o valor da energia de interação dipolo-dipolo entre esse mesmo *spin* antes de ter sua orientação modificada com os demais spins (Eq. 4.2), como mostra a equação 4.1 a seguir:

$$\Delta E_{dipolo} = E_{D_{depois}} - E_{D_{antes}} \tag{4.1}$$

$$E_{D_{antes}} = \sum_{\substack{j=1 \\ e \neq j}}^{n} \frac{\vec{S}_e \cdot \vec{S}_j}{\left|\vec{r}_{ej}\right|^3} - 3\frac{\left(\vec{S}_e \cdot \vec{r}_{ej}\right)\left(\vec{S}_j \cdot \vec{r}_{ej}\right)}{\left|\vec{r}_{ej}\right|^5} \tag{4.2}$$

Sendo $\vec{S}_e$ o *spin* escolhido antes de ter sua orientação modificada.

$$E_{D_{depois}} = \sum_{\substack{j=1 \\ e \neq j}}^{n} \frac{\vec{S}_f \cdot \vec{S}_j}{\left|\vec{r}_{fj}\right|^3} - 3\frac{\left(\vec{S}_f \cdot \vec{r}_{fj}\right)\left(\vec{S}_j \cdot \vec{r}_{fj}\right)}{\left|\vec{r}_{fj}\right|^5} \tag{4.3}$$

sendo $\vec{S}_f$ o *spin* escolhido após ter sua orientação modificada.

Em seguida, calcula-se a variação das energias de interação ferromagnética e de interação com o campo magnético externo apenas para o *spin* que teve sua direção modificada e soma-se esse valor à ΔE_{dipolo} para obter a variação total da energia ΔE.

Essa abordagem permitiu a realização do cálculo da variação da energia com menos trabalho computacional, melhorando assim o desempenho do simulador. Para diferenciar essa abordagem da anterior, a nova abordagem foi nomeada de 2xN.

4.2.2.2 Versão CUDA Distribuída

Para implementaçao da versão CUDA distribuída foi utilizado o código CUDA da versão centralizada, que será executado por cada uma das máquinas pertencentes ao anel. Apenas algumas modificações foram necessárias:

- Cada processo, ao criar seus vetores de orientação, posição e energia, vai ordená-los de acordo com os *spins* presentes na lista escolhida por ele para o processamento, ou seja, os *spins* que estiverem na lista daquele processo ficarão no início dos vetores. Essa abordagem nos permite parar o cálculo da energia facilmente quando esta tiver sido calculada para todos os *spins* da lista.

- Como nessa versão haverá mais de um processo em execução, o código que realiza a flipagem de um *spin* será realizado apenas no mestre, que enviará o índice do *spin* flipado juntamente com a sua nova orientação para os demais processos. Estes irão então atualizar a GPU com essa nova orientação.

- Cada processo somente irá calcular a energia referente às partículas presentes em sua lista, que ao final será enviada ao processo mestre, onde será somada às energias provenientes dos demais processos para a obtenção da variação total da energia.

4.3 Problemas Encontrados e Soluções

Algumas problemas foram encontrados durante o desenvolvimento do código do simulador. Listamos alguns deles a seguir, juntamente com as soluções encontradas para resolvê-los.

- **Problema:** Tempo gasto para verificar os *spins* pertencentes a lista de cada processo. Para uma matriz de tamanho 64x64x64, o código levava cerca de 20 minutos para realizar a verificação.

- **Solução:** Foi criado um vetor de ponteiros para a estrutura que representa uma partícula e uma função de *hash* para mapear cada partícula numa posição do vetor. Isso permitiu que o tempo para verificação das listas caísse para menos de 1 segundo.

- **Problema:** Limitação de memória para simulação de sistemas maiores que 216x216x216. A representação em memória de tais sistemas ocupa cerca de 484MB da memória, alocados na placa de vídeo.

- **Solução:** Como a simulação foi realizada em placas NVIDIA 9800 GTX, que possuem no máximo de 512MB de memória, não foi possível simular sistemas maiores.

- **Problema:** Nos computadores utilizados, as placas de vídeo eram também usadas pelo sistema operacional. Isso fazia com que todas as operações de transferência de dados entre CPU e GPU tivessem um tempo limite para serem realizadas. Quando o tamanho da matriz a ser simulada passava de 70x70x70, o tempo de transferência de dados ultrapassava esse tempo limite, fazendo com que a execução parasse.

- **Solução:** Após tentar, sem sucesso, executar o simulador com o *GDM* do Ubuntu desligado, a solução foi **desconectar os monitores** da placa de vídeo utilizada para simulação.

4.4 Testes e Resultados

Nessa seção são apresentados os resultados iniciais obtidos com a versão distribuída, assim como uma comparação com os resultados obtidos com a versão centralizada. Todas as simulações foram feitas utilizando duas máquinas, cada uma com um processo. Essas máquinas estão descritas na Figura 12.

	Máquina 1	**Máquina 2**
Processador	Intel Core 2 Quad Q6600 2.4 GHz	Intel Core 2 Quad Q8300 2.5 GHz
Memória	4GB	4GB
Placa de Vídeo	NVIDIA GeForce 9800 GTX	NVIDIA GeForce 9800 GTX
Sistema Operacional	Linux – Ubuntu 10.4 64 bits	Linux – Ubuntu 10.4 64 bits

Figura 12: Configuração das Máquinas Utilizadas nesse Trabalho.

A Figura 13 mostra uma comparação entre os tempos obtidos com a versão distribuída e a versão centralizada, ambas utilizando o código 2xN. A versão distribuída foi executada em duas máquinas com placas de vídeo NVIDIA GeForce 9800 GTX, que possui menor quantidade de memória e menor capacidade de processamento que a placa NVIDIA GeForce 295 GTX, onde foi executada a versão centralizada.

A Figura 13 mostra que o desempenho da versão distribuída ainda é inferior que o desempenho obtido com a versão centralizada. As principais causas dessa perda de desempenho são o fato de que o processo mestre precisa enviar a todos os demais processos a nova orientação do *spin*, que foi obtida no processo de flipagem, em seguida deve esperar

Tamanho do Sistema	Código Distribuído 2xN (s)	Código Centralizado 2xN (s)
37x37x37	3.84	1.55
100x100x100	32.75	13.51
216x216x216	345.66	117.71

Figura 13: Comparativo Entre os Tempos do Código Distribuído Com Código Local.

que todos os processos enviem o resultado do cálculo da energia parcial. Por último, o mestre deve enviar aos demais processos uma *flag* dizendo se a nova orientação do *spin* deve ser aceita ou não.

Na Figura 14 são apresentados os tempos de simulação de cada uma das configurações testadas nesse trabalho. Foram calculados: a) o tempo necessário para a transferência da matriz de simulação para os processos, b) o tempo decorrido para o cálculo da primeira energia e c) o tempo necessário para a realização dos passos de Monte Carlo. O tempo total, além dos tempos citados, leva em conta também o tempo de construção da *octree*, que não foi apresentado na figura.

Tamanho do Sistema	Tempo Transferência da Matiz	Tempo Obter Primeira Energia	Tempo Simulação Monte Carlo	Tempo Total
70x70x70	4.24 s	13.43 s	14.16 s	38.02 s
80x80x80	4.25 s	27.11 s	17.72 s	51.76 s
90x90x90	4.25 s	64.60 s	24.51 s	96.28 s
100x100x100	4.25 s	105.58 s	32.75 s	148.12 s
112x112x112	4.26 s	203.05 s	41.81 s	259.71 s
128x128x128	4.26 s	451.03 s	57.14 s	534.87 s
136x136x136	33.18 s	650.81 s	68.00 s	792.40 s
148x148x148	33.19 s	1509.27 s	104.14 s	1705.46 s
156x156x156	33.19 s	1571.48 s	110.84 s	1792.73 s
168x168x168	33.80 s	2306.61 s	122.13 s	2582,45 s
216x216x216	34.20 s	10423.45 s	345.66 s	10813.45 s

Figura 14: Tempos Obtidos Com a Simulação Distribuída.

5 Conclusão

Neste trabalho foi apresentada uma abordagem de paralelização e distribuição da execução de um simulador que realiza o cálculo da energia potencial de interação entre partículas com base em um modelo físico e computacional.

Esse método de paralelização se mostra ainda com algumas limitações. O comparativo apresentado neste trabalho mostra que a distribuição da execução do simulador ainda não permite ganhos de desempenho. Deve ser salientado que a comparação não foi justa, já que uma GPU mais nova foi utilizada para gerar os tempos da versão centralizada, enquanto uma GPU mais antiga foi empregada para gerar os tempos da versão distribuída. Também não foi possível simular sistemas maiores devido às limitações de memória da GPU.

Um outro fator que fez com que o código distribuído levasse um tempo maior de execução foi a penalidade de comunicação característica desse tipo de ambiente. O modo como é feita a comunicação precisa ser melhorado.

Apesar dessas limitações, algumas melhorias podem ser feitas no código para que se obtenha um desempenho melhor:

- Reduzir o tamanho da estrutura que representa a matriz original, enviada para a construção da *octree*. Isso é possível já que os métodos que realizam a divisão do espaço precisam somente da posição da partícula no espaço e de uma *flag* que diz se tal posição faz parte do objeto simulado. Essa melhoria permitirá que sistemas maiores sejam divididos na CPU, já que com o código atual só é possível realizar a divisão do espaço com a *octree* para matrizes com dimensões menores ou iguais a 256x256x256 (ainda que esta configuração não seja suportada pelas GPUs utilizadas nesse trabalho, devido à restrições de memória disponível).

- Modificar o modo como a distribuição dos parâmetros de simulação e demais mensagens de comunicação são enviados do mestre para os demais processos do ambiente pode ajudar na redução do tempo de comunicação entre os processos. As diretivas MPI_Send e MPI_Recv podem causar uma grande perda de tempo quando o anel

MPI tiver vários computadores, dado que as passagens de mensagens com essas diretivas são ponto-a-ponto, ou seja, o processo mestre precisa enviar uma mensagem de cada vez para cada um dos processos que integram o grupo. Isso pode fazer com que os últimos processos fiquem por muito tempo ociosos, esperando receber a mensagem com os dados para iniciar a simulação. Para resolver esse problema, a solução é usar alguma diretiva de comunicação coletiva como, por exemplo, o MPI_BCast, que permite ao processo mestre enviar os dados a todos os demais processos de uma só vez.

Referências

ALDER, B. J.; WAINWRITE, T. E. Studies in molecular dynamics. i. general method. *The Journal of Chemical Physics*, v. 31, n. 2, p. 459 – 466, Agosto 1959.

AYALA, D. et al. Object representation by means of nonminimal division quadtree and octrees. *ACM Transactions on Graphics (TOG)*, v. 4, n. 1, Jan 1985.

CAMPOS, A. M. *Implementação Paralela de Um Simulador de Spins em Uma Unidade de Processamento Gráfico*. 42 f. Monografia (Graduação) — Faculdade de Ciência da Computação, Universidade Federal de Juiz de Fora, Juiz de Fora, 2008.

CAMPOS, A. M. Correspondência privada. 2010.

FERNANDES, F. M. S. S.; RAMALHO, J. P. P. Simulação computacional. i. fundamentos do método de monte carlo. *Revista Ciência*, v. 5, n. 3, p. 7 – 9, 1989.

FERNANDES, F. M. S. S.; RAMALHO, J. P. P. Simulação computacional. o método da dinâmica molecular. *Revista Ciência*, v. 5, n. 3, p. 7 – 9, 1989.

FORUM, M. P. I. *MPI: A Message-Passing Interface Standard - Versão 2.2*. [S.l.], Setembro 2009.

GRZYBOWSKI, A.; GWOZDZ, E.; BRóDKA, 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, Março 2000.

KARNIADAKIS, G. E.; KIRBY, R. *Parallel Scientific Computing in C++ and MPI*. [S.l.], 2003.

KIRK, D.; WHU, W. mei. Cuda programming model. In: *ECE 498AL*. [S.l.]: University of Illinois, 2009. cap. 2.

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

NVIDIA. *NVIDIA CUDA Programming Guide*. [S.l.], Março 2010.

NYLAND, L.; HARRIS, M.; PRINS, J. Fast n-body simulation with cuda. In: *GPU Gems*. [S.l.]: NVIDIA Corporation, 2004. cap. 31, p. 677–695.

PALMER, D. W.; PRINS, J. F.; WESTFOLD, S. Work-efficient nested data-parallelism. *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Processing (Frontiers 95). IEEE*, 1995.

PEÇANHA, J. P. et al. Simulação computacional da interação de spins em sistemas magnéticos usando múltiplos fluxos de execução. *XII Encontro de Modelagem Computacional*, 2009.

RIBEIRO, G. A. P. As propriedades magnéticas da matéria: Um primeiro contato. *Revista Brasileira de Ensino da Física*, v. 22, n. 3, p. 299 – 305, 2000.

SAMET, H.; WEBBER, R. E. Hierarchical data structures and algorithms for computer graphics. i. fundamentals. *Computer Graphics and Applications, IEEE*, v. 8, n. 3, p. 48 – 68, Maio 1988.

ZHOU, K. et al. Real-time kd-tree construction on graphics hardware. *ACM Transactions on Graphics (TOG)*, v. 27, n. 5, Dezembro 2008.