Thales Luis Rodrigues Sabino

Simulação Computacional de Nano-estruturas em Unidades Gráficas

Juiz de Fora

Thales Luis Rodrigues Sabino

Simulação Computacional de Nano-estruturas em Unidades Gráficas

Orientador:
Marcelo Bernardes Vieira

Co-orientadores:
Marcelo Lobosco
Sócrates de Oliveira Dantas

UNIVERSIDADE FEDERAL DE JUIZ DE FORA
INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Juiz de Fora

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 Bernardes Vieira, D. Sc.

Prof. Marcelo Lobosco

Prof. Sócrates de Oliveira Dantas

Sumário

Lista de Figuras

Resumo

1 Introdução p. 9

1.1 Definição do Problema . . . . . p. 9

1.2 Objetivos . . . . . p. 10

2 Dinâmica Molecular p. 11

2.1 Sistemas Físicos . . . . . p. 11

2.2 Potenciais de Interação . . . . . p. 12

2.2.1 Potencial de Lennard-Jones . . . . . p. 13

2.3 Condições Periódicas de Contorno . . . . . p. 15

2.3.1 Efeito da Periodicidade . . . . . p. 15

2.3.2 Critério da Imagem mais Próxima . . . . . p. 16

2.4 Mecânica Estatística . . . . . p. 16

2.4.1 Energia Potencial . . . . . p. 17

2.4.2 Energia Cinética . . . . . p. 17

2.4.3 Energia Total . . . . . p. 17

2.4.4 Temperatura . . . . . p. 18

2.4.5 Deslocamento Quadrático Médio . . . . . p. 18

2.4.6 Pressão . . . . . p. 19

2.4.7 Calor Específico . . . . . p. 19

2.5 Métodos de Integração Temporal . . . . . p. 19
2.5.1 Método *Leapfrog* . . . . . p. 20
2.5.2 Algoritmo de *Verlet* . . . . . p. 20
2.5.3 Método Preditor-Corretor . . . . . p. 22
2.5.4 Comparação dos métodos . . . . . p. 23

3 Simulação de Dinâmica Molecular p. 24
3.1 Unidades Adimensionais . . . . . p. 24
3.2 Computação de Interações . . . . . p. 25
3.2.1 Listas de Vizinhos . . . . . p. 26
3.2.2 Divisão por Células . . . . . p. 27
3.2.3 Comparação das Técnicas . . . . . p. 27
3.3 Inicialização do Sistema . . . . . p. 28
3.3.1 Inicialização das Coordenadas . . . . . p. 28
3.3.2 Inicialização das Velocidades . . . . . p. 28
3.4 Passo de Simulação . . . . . p. 29
3.5 Medidas de Propriedades Termodinâmicas . . . . . p. 31

4 Simulação de Dinâmica Molecular em Unidades de Processamento Gráfico de Propósito Geral p. 33
4.1 *GPU* : Um Processador com Muitos Núcleos, Multithread e Altamente Paralelo p. 33
4.1.1 *CUDA*: Um Modelo de Programação Paralela Escalável . . . . . p. 35
4.2 Modelo de Programção . . . . . p. 36
4.2.1 Hierarquia de *Threads* . . . . . p. 37
4.2.2 Hierarquia de Memória . . . . . p. 39
4.2.3 *Hospedeiro* e *Dispositivo* . . . . . p. 39
4.3 Cálculo das Interações Interatômicas em *CUDA* . . . . . p. 39
4.3.1 Conceito de Ladrilho . . . . . p. 41

4.3.2 Definindo uma Grade de Blocos de *threads* . . . . . p. 41
4.3.3 Detalhamento da Implementação . . . . . p. 42
4.4 Redutor Paralelo em *CUDA* . . . . . p. 45
4.4.1 Redutor Paralelo . . . . . p. 45
4.4.2 Implementação em *CUDA* . . . . . p. 46
4.4.3 Detalhamento da Implementação . . . . . p. 47

5 Resultados p. 50
5.1 Interface . . . . . p. 50
5.2 Medidas de Propriedades . . . . . p. 51
5.3 Testes de desempenho . . . . . p. 54
5.3.1 Tempo de Execução . . . . . p. 56
5.3.2 Iterações por Segundo . . . . . p. 57
5.3.3 Razão Entre os Tempos de Execução . . . . . p. 57
5.4 Discussão dos Resultados . . . . . p. 59

6 Conclusão p. 61

Referências Bibliográficas p. 62

Lista de Figuras

2.1 Potencial de LJ para o dímero de Argônio: $\varepsilon = 997 Joule/mol$ e $\sigma = 3.4 \times 10^{-10} m$. . . . . p. 14

3.1 Diagrama do passo de simulação. . . . . p. 30

3.2 Variação da energia cinética e potencial em função do tempo. $N = 1000$, densidade= $1.2 g/cm^3$ e temperatura= $30k$. . . . . p. 32

4.1 Comparação do número de operações em ponto flutuante por segundo entre CPU e *GPU* (NVIDIA, 2009). . . . . p. 35

4.2 Comparação da largura de banda entre entre processador e memória principal para CPU e *GPU* (NVIDIA, 2009). . . . . p. 35

4.3 Comparação das estruturas internas entre *GPU* e CPU: A *GPU* dedica mais transistors ao processamento de dados (NVIDIA, 2009). . . . . p. 36

4.4 Diagrama de um grid de blocos de threads (NVIDIA, 2009). . . . . p. 37

4.5 Hierarquia de memória (NVIDIA, 2009). . . . . p. 40

4.6 A grade de blocos de *threads* que calcula as N^2 interações. Aqui são definidos quatro blocos de quatro *threads* cada (NYLAND; HARRIS; PRINS, 2006). . p. 41

4.7 Esquema de um *tile* computacional. Linhas são calculadas em paralelo, mas cada linha é avaliada sequencialmente por uma *thread CUDA* (NYLAND; HARRIS; PRINS, 2006). . . . . p. 42

4.8 Abordagem em árvore para implementação de um redutor paralelo. (HARRIS, 2006). . . . . p. 46

4.9 Problema da redução resolvido com múltiplas chamadas de *kernels* (HARRIS, 2006). . . . . p. 47

4.10 Esquema de um bloco de *threads* fazendo a redução de um vetor (HARRIS, 2006). . . . . p. 48

4.11 Redutor paralelo para redução de vetores tridimensionais. . . . . p. 48

5.1 Interface de controle do simulador. . . . . p. 51

5.2 Interface de exibição em tempo real da simulação. $N = 1000$ átomos. . . . . p. 52

5.3 Interface de exibição em tempo real da simulação com processamento em *GPU* . $N = 65536$ átomos. . . . . p. 52

5.4 Aumento da energia cinética com o aumento da temperatura. . . . . p. 53

5.5 Aumento da energia potencial com o aumento da temperatura. . . . . p. 54

5.6 Aumento da pressão com o aumento da temperatura. . . . . p. 54

5.7 Tempo de simulação usando o Computador 1. . . . . p. 56

5.8 Tempo de simulação usando o Computador 2. . . . . p. 57

5.9 Iterações por segundo usando o Computador 1. . . . . p. 58

5.10 Iterações por segundo usando o Computador 2. . . . . p. 58

5.11 Razão entre tempos de execução da implementação *CUDA* e as demais usando o Computador 1. . . . . p. 59

5.12 Razão entre tempos de execução da implementação *CUDA* e as demais usando o Computador 2. . . . . p. 60

Resumo

Este trabalho trata sobre o tema Simulação Computacional de Dinâmica Molecular. Apresenta uma descrição detalhada sobre os procedimentos necessários para a confecção de um simulador, tanto sequencial, quando paralelo usando o modelo de programação *NVIDIA CUDA*. São apresentados os fundamentos necessários para a compreensão do problema de Dinâmica Molecular com foco nos detalhes de implementação. Para finalizar, este trabalho apresenta os resultados obtidos com a implementação de um simulador de Dinâmica Molecular para sistemas constituídos de átomos de argônio interagindo via potencial de Lennard-Jones.

1 Introdução

Métodos de Dinâmica Molecular (DM) são, agora, meios ortodoxos para simular modelos da matéria em escala molecular. A essência da DM é simples: resolver numericamente o problema dos N-Corpos da mecânica clássica (HAILE, 1992). A DM permite determinar propriedades macroscópicas de sistemas moleculares a partir do estudo em escala microscópica.

Simulações de DM são a realização moderna de uma ideia antiga da ciência. O comportamento de um sistema pode ser determinado se tem-se um conjunto de condições iniciais, além do conhecimento do potencial de interação entre as partículas que o compõe. Desde os tempos de Newton, essa interpretação determinística da mecânica da natureza dominou a ciência (COHEN, 1985). Em 1814, quase um século depois de Newton, Laplace escreveu (LAPLACE, 1914):

> Dada uma inteligência que por um instante compreendesse todas as forças pelas quais a natureza é regida assim como o estado dos seres que a compõem - uma inteligência suficientemente vasta para submeter esses dados a uma análise - que pudesse incorporar na mesma fórmula os movimentos dos maiores corpos do universo e dos mais leves átomos, para ela, nada seria incerto e o futuro, assim como o passado, estariam presentes diante seus olhos.

Desde que os primeiros métodos de DM surgiram, computadores são usados como ferramenta para estudo dos mesmos. O advento de tecnologias de processamento paralelo continua contribuindo significativamente para que resultados de simulações dos métodos de DM sejam obtidos de forma mais rápida e para sistemas cada vez maiores. O estudo de tais tecnologias é de fundamental importância para o avanço nos estudos da DM.

1.1 Definição do Problema

Esta monografia abordará o problema da modelagem e simulação em unidades gráficas (*GPU*) de sistemas envolvendo átomos de argônio.

Na modelagem, o problema é a simplificação dos cálculos para evitar uso de números muito

próximos do limite da representação do hardware e evitar riscos de *underflow* e *overflow*. As unidades gráficas são, hoje, poderosos multiprocessadores massivamente paralelos (NVIDIA, 2009). Um estudo da arquitetura das (*GPU*) será apresentado de forma a tirar proveito do poder computacional proporcionado por elas. Na simulação, o desafio é mapear o problema de forma eficiente para uso das *GPUs* .

O problema de visualização do sistema também será abordado apresentando um estudo sobre visualização de sistemas em escala nanométrica.

1.2 Objetivos

O objetivo primário desta monografia é pesquisar e formalizar os métodos de simulação de DM. Como objetivos secundários, advindos do objetivo primário, temos:

- Apresentar matemáticamente os conceitos dos métodos de DM;
- Apresentar um estudo sobre arquiteturas paralelas com foco nas *GPUs* ;
- Aplicar os conceitos estudados na implementação de um simulador de sistemas contendo átomos de argônio com o uso da tecnologia *CUDA*.

2 *Dinâmica Molecular*

O conceito de DM foi proposto por (ALDER; WAINWRIGHT, 1959) para simular o comportamento mecânico de sistemas moleculares. A base teórica da DM engloba vários dos importantes resultados produzidos por grandes nomes da Mecânica - Euler, Hamilton, Lagrange e Newton. Alguns destes resultados contêm informações fundamentais sobre o aparente funcionamento da natureza, outros, são elegantes resultados que serviram de base para desenvolvimento teórico futuro (RAPAPORT, 1996).

As origens da DM estão no atomismo da antiguidade. No presente momento, sua importância vem da tentativa de relacionar a dinâmica coletiva com a dinâmica de partículas individuais. A DM também é motivada pela promessa de que o comportamento de sistemas em larga escala pode ser explicado pelo exame do movimento de partículas individuais. A descoberta do átomo levantou novas hipóteses para explicar comportamentos produzidos pela matéria em escala macroscópica. Perguntas, antes sem respostas, começaram a ser respondidas. Por exemplo, como o fluxo de fluído em torno de um objeto produz uma trilha turbulenta? Como os átomos de uma molécula de proteína se movem juntos de forma a dar suporte à vida? Como um turbilhão em um fluído produz um vórtice duradouro, como a grande mancha vermelha em Júpiter? Como uma perturbação local de algumas moléculas (por exemplo, por um pulso *laser*) se propaga através de um sistema? Como moléculas individuais se combinam para formar novas moléculas? Tais questões sugerem que a DM pode esclarecer dúvidas de diversas áreas de pesquisa (HAILE, 1992).

2.1 Sistemas Físicos

A porção do mundo físico que se tem interesse é chamada de *sistema*, que é um subconjunto do universo. O sistema é composto por um número arbitrário de partes similares ou dissimilares e o estado dessas partes denotam o *estado* dos mesmos. Para analisar e descrever o comportamento dos sistemas precisa-se associar valores numéricos aos estados do sistema. Tais valores são chamados *observáveis*. Como exemplo, seja um sistema com 10^{20} moléculas de gás. Seu

estado é dado pela posição e momento de cada molécula, além do mais, esse estado nos fornece observáveis como temperatura e pressão.

O estado de um sistema pode ser manipulado e controlado por *interações*. Vários tipos de interações são possíveis, porém o estudo apresentado será focado em sistemas *isolados*. Normalmente o estudo de um sistema não é feito pela observação direta do seu estado, ao invés, é feita uma sondagem do estado pela medida de observáveis.

2.2 Potenciais de Interação

O resultado de qualquer reação química é determinado por propriedades físicas como massa e temperatura e uma propriedade física denominada *potencial de interação*. O potencial determina as forças que agem em vários átomos e decide por sua vez, quando é possível com uma dada energia, contida no sistema quebrar uma ligação ou formar uma nova, isto é, quando uma reação química pode ser realizada (BILLING; MIKKELSEN, 1996).

O modelo microscópico mais rudimentar para uma substância capaz de existir nos três estados da matéria mais familiares - sólido, líquido e gasoso - é baseado em partículas esféricas que interagem umas com as outras. Por questões de simplicidade, as partículas serão referidas como átomos. As interações no seu nível mais simples ocorrem entres pares de átomos e são responsáveis por determinar as duas principais características de uma força interatômica. A primeira é a resistência a compressão, portanto a interação é de repulsão a curtas distâncias. A segunda é a ligação de átomos nos estados de sólido ou liquido, assim, o potencial deve ser de atração a distâncias maiores a fim de manter a coesão do material (RAPAPORT, 1996).

Neste trabalho consideramos somente moléculas esféricas simétricas (átomos). Para N átomos a função de interação intermolecular é representada por $\mathscr{U}(\mathbf{r}^N)$. A notação $\mathbf{r}^N$ representa o conjunto de vetores que localizam os centros de massa dos átomos.

$$\mathbf{r}^N = \{\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3, \ldots, \mathbf{r}_N\} \tag{2.1}$$

A *configuração* do sistema é definida quando são associados valores ao conjunto $\mathbf{r}^N$. Propriedades macroscópicas que estão relacionadas somente com o conjunto $\mathbf{r}^N$ são denominadas propriedades *configuracionais* (HAILE, 1992).

Em muitas simulações a energia de interação intermolecular é dada pela soma par-a-par das interações isoladas, isto é:

$$\mathscr{U} = \sum_{\substack{j=1 \\ j \neq i}}^{N} u(r_{ij}) \tag{2.2}$$

onde $u(r_{ij})$ é a função que calcula a energia potencial do par *i*-*j* e r_{ij} é a distância escalar entre os átomos i e j.

Como não existem forças dissipativas atuando sobre os átomos, as forças de interação intermoleculares são conservativas, portanto a força total atuante sobre um átomo i está relacionada com o gradiente do potencial da seguinte forma:

$$\mathbf{F}_i = -\nabla_i \mathscr{U}(\mathbf{r}^N) = -\frac{\partial \mathscr{U}(\mathbf{r}^N)}{\partial \mathbf{r}_i} \tag{2.3}$$

Em física, a dinâmica é um ramo da mecânica que estuda as relações entre as forças e os movimentos que são produzidos por estas. A DM é uma metodologia que estuda como as forças de interação intermoleculares se relacionam com o movimento dos átomos que compõe o sistema sendo estudado. As posições $\mathbf{r}^N$ são obtidas a partir da resolução das equações do movimento de Newton:

$$\mathbf{F}_i(t) = m\ddot{\mathbf{r}}_i(t) = -\frac{\partial \mathscr{U}(\mathbf{r}^N)}{\partial \mathbf{r}_i} \tag{2.4}$$

onde $\mathbf{F}_i$ é a força causada pelos outros $N-1$ átomos sobre o átomo i, $\ddot{\mathbf{r}}_i$ sua aceleração e m sua massa molecular.

2.2.1 Potencial de Lennard-Jones

O melhor modelo matemático que descreve a interação intermolecular mencionada, para sistemas simples, é o potencial de de Lennard-Jones (LJ). Esse modelo foi proposto por John Lennard-Jones (LENNARD-JONES, 1924), originalmente para modelar argônio líquido. O potencial de LJ é comumente usado para simular o comportamento de gases nobres a baixas densidades. A Figura 2.1 mostra o gráfico do potencial de LJ em função da distância.

Para um par de átomos i e j localizados em $\mathbf{r}_i$ e $\mathbf{r}_j$, o potencial de LJ é definido como:

$$u_{lj}(r_{ij}) = 4\varepsilon \left[\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right], r_{ij} \leq r_c = 2^{1/6}\sigma \tag{2.5}$$

onde $\mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j$ e $r_{ij} \equiv \left|\mathbf{r}_{ij}\right|$, σ é aproximadamente o diâmetro molecular e ε é a profundidade do poço de potencial. Este potencial possui uma propriedade atrativa quando r_{ij} é grande e

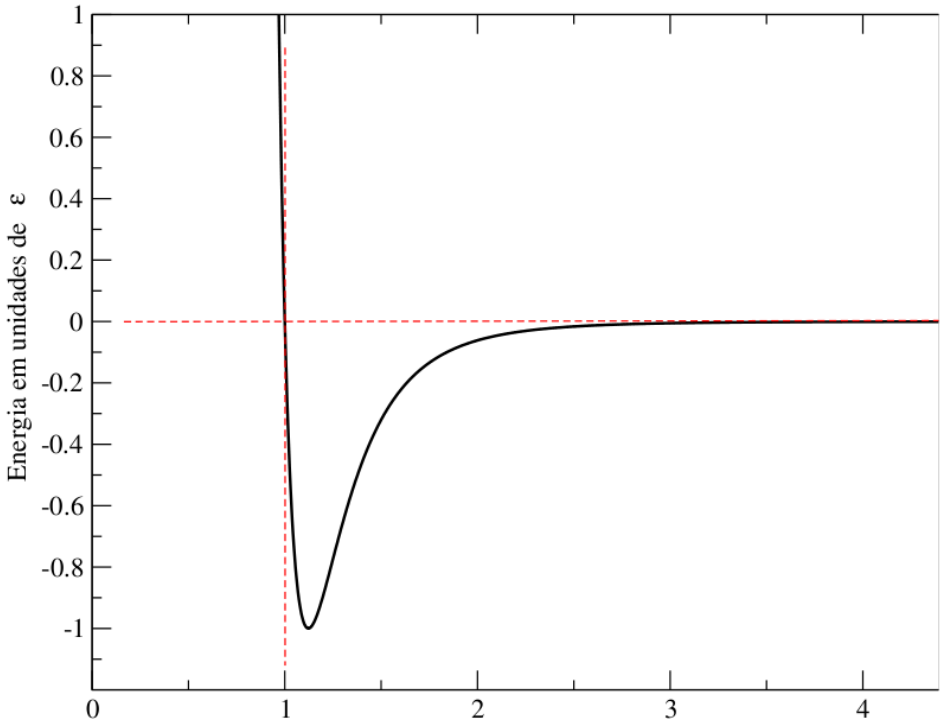

Figura 2.1: Potencial de LJ para o dímero de Argônio: $\varepsilon = 997 Joule/mol$ e $\sigma = 3.4 \times 10^{-10} m$.

atinge um mínimo quando $r_{ij} = 1.122\sigma$, sendo fortemente repulsivo a curtas distâncias, passando por 0 quando $r = \sigma$ e aumentando rapidamente à medida que r_{ij} decresce.

Sabendo que a força correspondente a $u(r_{ij})$ é dada pela Equação 2.3, tem-se que a força que um átomo j exerce em um átomo i é:

$$\mathbf{f}_{ij} = \left(\frac{48\varepsilon}{\sigma^2}\right)\left[\left(\frac{\sigma}{r_{ij}}\right)^{14} - \frac{1}{2}\left(\frac{\sigma}{r_{ij}}\right)^{8}\right]\mathbf{r}_{ij} \tag{2.6}$$

Seguindo da Equação 2.4 temos que a equação que rege o movimento de átomo i, dada a força de interação intermolecular (Eq. 2.6) é:

$$m\ddot{\mathbf{r}}_i = \mathbf{F}_i = \sum_{\substack{j=1 \\ j\neq i}}^{N} \mathbf{f}_{ij} \tag{2.7}$$

onde a soma é feita sobre todos os N átomos, exceto o próprio i.

2.3 Condições Periódicas de Contorno

A DM é normalmente aplicada a sistemas contendo de centenas a milhares de átomos. Tais sistemas são considerados pequenos e são dominados por interações de superfície, isto é, interações dos átomos com as paredes do recipiente envoltório. A simulação é feita em um recipiente de paredes rígidas sobre as quais os átomos irão colidir na tentativa de escapar da região de simulação. Em sistemas de escala macroscópica, somente uma pequena fração está próxima o suficiente das paredes de forma a interagir com as mesmas. Uma simulação desse caráter deve ser capaz de tratar como os átomos interagem com as paredes além de como eles interagem entre si. Por exemplo, seja um sistema tridimensional com $N = 10^{21}$ com uma densidade de líquido. Tem-se que o número de átomos próximos das paredes é da ordem de $N^{2/3}$, o que resulta em 10^{14} átomos - somente um em 10^7. Mas se $N = 1000$, um valor mais típico em DM, 500 átomos estão imediatamente adjacente às paredes, deixando somente alguns átomos no seu interior. Se as duas primeiras camadas forem excluídas, restam somente 216 átomos. Portanto a simulação irá falhar em capturar o estado típico dos átomos do interior e as medidas irão refletir esse fato. A menos que seja de interesse estudar o comportamento dos átomos próximos às paredes, elimina-se sua existência através das *condições periódicas de contorno* (CPC) (RAPAPORT, 1996; HAILE, 1992).

Para usar CPC em uma simulação onde N átomos estão confinados em um volume V, considera-se que esse volume é somente uma parte do material. O volume V é chamado de *célula primária* (V deve ser suficientemente grande para representar esse material). V é rodeado por réplicas exatamente iguais chamadas *células imagem*. Cada célula imagem possui o mesmo tamanho de V e possui N átomos igualmente posicionados. Diz-se então que a célula primária replicada periodicamente em todas as direções cartesianas do sistema forma uma amostra macroscópica da substância de interesse. Essa periodicidade se estende a posição e momento dos átomos nas células imagem (HAILE, 1992).

2.3.1 Efeito da Periodicidade

Existem duas consequências dessa periodicidade. A primeira é que um átomo que deixa a região de simulação por uma fronteira particular, imediatamente reentra na região de simulação através da fronteira oposta. A segunda é que átomos que estão dentro da distância de corte r_c em uma fronteira, interagem com os átomos na célula imagem adjacente, ou equivalentemente, com os átomos da próximos da fronteira oposta.

Os efeitos causados pelas CPC devem ser levados em conta tanto na integração das equações

de movimento quanto na avaliação da equação de interação. Após cada passo de integração, as coordenadas de cada um dos N átomos deve ser avaliada e, se um átomo foi movido para fora da região de simulação, suas coordenadas devem ser ajustadas de forma a trazê-lo de volta, mas pela fronteira oposta. Se, por exemplo, a coordenada x de cada átomo é definida como $x \in [-L_x/2, L_x/2]$, onde L_x é o tamanho da região na direção de x, os testes a serem feitos são os seguintes: se $r_{xi} \geq L_x/2$ então substitua r_{xi} por $r_{xi} - L_x$ e se $r_{xi} < -L_x/2$ então substitua r_{xi} por $r_{xi} + L_x$. Os testes devem ser repetidos para as componentes y e z da posição de cada átomo.

2.3.2 Critério da Imagem mais Próxima

Supondo a utilização de um potencial com alcance finito, isto é, quando duas partículas i e j estão separadas por uma distância $r_{ij} \geq r_c$, não ocorre interação entre elas. Supondo uma região cúbica de lado $L > 2r_c$. Quando essas condições são satisfeitas é óbvio que ao menos um entre todos os pares formados pela partícula i na célula primária e o conjunto de todas as imagens periódicas de j irão interagir.

Como demonstração, suponha um átomo i que interage com duas imagens j_1 e j_2 de j. Duas imagens devem estar separadas por um vetor translação ligando a célula primária em uma célula secundária, e cujo comprimento é no mínimo $2r_c$ por hipótese. No intuito de interagir com ambas j_1 e j_2, i deve estar a uma distância $r < r_c$ de cada uma delas, o que é impossível, uma vez que elas estão separadas por $r \geq r_c$.

Uma vez que estás condições são satisfeitas pode-se seguramente utilizar o *critério da imagem mais próxima*: dentre todas as possíveis imagens do átomo j, selecionar a mais próxima e descarte todas as outras. Isto pode ser feito pois somente a imagem mais próxima é a candidata a interagir, todas as outras, certamente, não irão contribuir para a força aplicada sobre i (ALLEN; TILDESLEY, 1987).

2.4 Mecânica Estatística

Simulações computacionais produzem informações no nível microscópico (posições atômicas e/ou moleculares, velocidades e acelerações de átomos individuais). A conversão dessas informações muito detalhadas sobre um sistema para termos macroscópicos (pressão, energia interna etc.) é a área de estudo da mecânica estatística (ALLEN; TILDESLEY, 1987).

Medir quantidades de interesse em DM usualmente significa efetuar médias temporais de propriedades físicas sobre a trajetória do sistema. Propriedades físicas são, usualmente, medidas

em função das coordenadas e velocidades atômicas. Por exemplo, seja $\mathscr{A}$ o valor de uma propriedade física genérica em um instante de tempo t, podemos definir $\mathscr{A}$ como:

$$\mathscr{A} = f(\mathbf{r}^N(t), \mathbf{v}^N(t)) \tag{2.8}$$

e então obter sua média:

$$\langle \mathscr{A} \rangle = \frac{1}{N_T} \sum_{t=1}^{N_T} \mathscr{A}(t) \tag{2.9}$$

onde t é um índice que varia de 1 até o número total de medidas efetuadas N_T.

A maneira mais viável de realizar tais medidas na prática é calcular $\mathscr{A}(t)$ em cada passo de simulação (ou a cada certo número de passos) e usar um acumulador $\mathscr{A}_{acum}$ também atualizado convenientemente. No final de N_T medidas, divide-se $\mathscr{A}_{acum}$ por N_T obtendo assim $\langle \mathscr{A} \rangle$.

A seguir são apresentadas as propriedades mais comuns que podem ser medidas.

2.4.1 Energia Potencial

A energia potencial média $\langle \mathscr{E}_\mathscr{U} \rangle$ é obtida pela média dos seus valores instantâneos. Para o caso de um potencial de interação $u(r_{ij})$, $\langle \mathscr{E}_\mathscr{U} \rangle$ é obtido como:

$$\langle \mathscr{E}_\mathscr{U} \rangle = \frac{1}{N_T} \sum_{i=1}^{N} \sum_{j>i}^{N} u(r_{ij}) \tag{2.10}$$

2.4.2 Energia Cinética

A energia cinética média $\langle \mathscr{E}_\mathscr{K} \rangle$ é calculada como:

$$\langle \mathscr{E}_\mathscr{K} \rangle = \frac{1}{2N_T} \sum_{i=1}^{N} m_i \mathbf{v}_i(t)^2 \tag{2.11}$$

2.4.3 Energia Total

A energia total média $\langle \mathscr{E} \rangle$ é uma propriedade conservativa na dinâmica Newtoniana. Verificações sobre a variação da energia total são comuns em simulações de DM. Durante uma simulação, tanto $\mathscr{E}_\mathscr{U}$ quanto $\mathscr{E}_\mathscr{K}$ flutuam, porém sua soma deve permanecer constante. Métodos de verificação e correção em uma possível variação de $\langle \mathscr{E} \rangle$ serão apresentados no Capítulo 3.

$$\langle \mathscr{E} \rangle = \langle \mathscr{E}_{\mathscr{K}} \rangle + \langle \mathscr{E}_{\mathscr{U}} \rangle \tag{2.12}$$

2.4.4 Temperatura

A temperatura T está diretamente relacionada com a energia cinética $\mathscr{K}$, via Teoria cinética dos gases. Para cada grau de liberdade do sistema ($d = 1$, sistema unidimensional, $d = 2$, sistema bidimensional, $d = 3$, sistema tridimensional) é associada uma média da energia cinética $k_B T/2$. Portanto tem-se que a temperatura instantânea é dada por:

$$\mathscr{E}_{\mathscr{K}} = \frac{3}{2} N k_B T \tag{2.13}$$

onde k_B é a constante de *Boltzmann*. Reescrevendo a Equação 2.13 de modo a evidenciar T obtêm-se:

$$T = \frac{2}{3} \frac{\mathscr{E}_{\mathscr{K}}}{N k_B} \tag{2.14}$$

2.4.5 Deslocamento Quadrático Médio

O deslocamento quadrático médio (*DQM*) é uma propriedade que diz o quão longe um átomo se deslocou de sua posição inicial, isto permite determinar a difusividade atômica. Por definição tem-se:

$$DQM = \left\langle |\mathbf{r}(t) - \mathbf{r}(0)|^2 \right\rangle \tag{2.15}$$

Se o sistema está frio, *DQM* satura para um valor finito enquanto que, se o sistema está em estado líquido e a altas temperaturas, o *DQM* cresce linearmente com o tempo. É possível medir o coeficiente de difusividade $\mathscr{D}$ através do coeficiente de inclinação da curva resultante do *DQM*:

$$\mathscr{D} = \lim_{t \to \infty} \frac{1}{2d \cdot t} \left\langle |\mathbf{r}(t) - \mathbf{r}(0)|^2 \right\rangle \tag{2.16}$$

onde d é a dimensão do sistema e t é o intervalo de tempo onde as medidas foram efetuadas.

2.4.6 Pressão

A pressão $\mathscr{P}$ é obtida a partir da função do *Virial* $\mathscr{W}$ (HANSEN; EVANS, 1994):

$$\mathscr{W}(\mathbf{r}^N) = \sum_{i=1}^{N} \mathbf{r}_i \cdot \mathbf{F}_i \tag{2.17}$$

onde $\mathbf{F}_i$ é a força total atuando em um átomo i.

A pressão é então definida como:

$$\mathscr{P}\mathscr{V} = Nk_BT + \frac{1}{d}\langle \mathscr{W} \rangle \tag{2.18}$$

onde $\mathscr{V}$ é o volume do sistema.

2.4.7 Calor Específico

Propriedades termodinâmicas baseadas em flutuações possuem uma forma diferente de serem estimadas quando se deseja obter propriedades macroscópica através do movimento em escala microscópica (RAPAPORT, 1996). A propriedade mais comum para esse tipo de abordagem é o *calor específico* $\mathscr{C}_\mathscr{V} = \partial\mathscr{E}/\partial T$ a um volume constante. $\mathscr{C}_\mathscr{V}$ é usualmente definido em termos da flutuação de energia, ou seja:

$$\mathscr{C}_\mathscr{V} = \frac{N}{k_BT^2}\left\langle \delta\mathscr{E}^2 \right\rangle \tag{2.19}$$

onde $\left\langle \delta\mathscr{E}^2 \right\rangle = \left\langle \mathscr{E}^2 \right\rangle - \langle \mathscr{E} \rangle^2$. Contudo, em DM, $\left\langle \mathscr{E}^2 \right\rangle = 0$. Ao invés de $\langle \mathscr{E} \rangle$, as flutuações mais relevantes a serem consideradas são as de $\mathscr{E}_\mathscr{U}$ e $\mathscr{E}_\mathscr{K}$, que são idênticas. Assim o $\mathscr{C}_\mathscr{V}$ é obtido como:

$$\mathscr{C}_\mathscr{V} = \frac{3k_B}{2}\left(1 - \frac{2N\left\langle \delta\mathscr{E}_\mathscr{K}^2 \right\rangle}{3(k_BT)^2}\right)^{-1} \tag{2.20}$$

2.5 Métodos de Integração Temporal

Muitos métodos de integração numérica estão disponíveis para resolução das equações de movimento. Vários são rapidamente dispensáveis devido ao alto poder computacional exigido. O método de integração não pode ser mais custoso do que a avaliação da força de interação intermolecular. Esse custo computacional extra de alguns métodos eleva o tempo de simulação

para níveis indesejáveis.

A seguir são apresentados três métodos de integração numérica muito usados em simulações de DM. A Seção 2.5.4 mostra uma breve comparação dos métodos apresentados.

2.5.1 Método *Leapfrog*

O método mais simples de integração numérica usado em simulações de DM é o método *Leapfrog*. Proposto por (BEEMAN, 1976), este método apresenta excelentes propriedades de conservação de energia e os requerimentos de informações sobre estados passados do sistema é mínimo.

O método *Leapfrog* possui dois passos. Primeiro calcula-se as velocidades no instante de tempo $t + \Delta t/2$. As posições, então, são calculadas no instante de tempo $t + \Delta t$, a partir das velocidades calculadas:

$$\begin{aligned} v_{xi}(t + \Delta t/2) &= v_{xi}(t - \Delta t/2) + \Delta t \cdot a_{xi}(t) \\ r_{xi}(t + \Delta t/2) &= r_{xi}(t) + \Delta t \cdot v_{xi}(t + \Delta t/2) \end{aligned} \tag{2.21}$$

Caso seja necessário saber a velocidade no tempo em que as coordenadas são calculadas, então pode-se usar:

$$v_{xi}(t) = v_{xi}(t - \Delta t/2) + (\Delta t/2) \cdot a_{xi}(t) \tag{2.22}$$

O nome *Leapfrog* vem do fato de que coordenadas e velocidades não são calculadas simultaneamente, o que caracteriza uma desvantagem desse método.

2.5.2 Algoritmo de *Verlet*

Em DM, o algoritmo de integração mais utilizado é o algoritmo de *Verlet* (VERLET, 1967). A ideia básica é escrever duas expressões em série de Taylor até a terceira ordem para as posições $\mathbf{r}(t)$, uma para um instante de tempo posterior, outra para um instante de tempo anterior. Seja as velocidades das partículas do sistema **v** e suas acelerações **a** e a terceira derivada de **r** em relação a sendo **b** tem-se:

$$\begin{aligned}\mathbf{r}(t+\Delta t) &= \mathbf{r}(t)+\mathbf{v}(t)\Delta t+\frac{1}{2}\mathbf{a}(t)(\Delta t)^2+\frac{1}{6}\mathbf{b}(t)(\Delta t)^3+O[(\Delta t)^4] \\ \mathbf{r}(t-\Delta t) &= \mathbf{r}(t)-\mathbf{v}(t)\Delta t+\frac{1}{2}\mathbf{a}(t)(\Delta t)^2-\frac{1}{6}\mathbf{b}(t)(\Delta t)^3+O[(\Delta t)^4]\end{aligned} \tag{2.23}$$

Adicionando as duas expressões obtêm-se:

$$\mathbf{r}(t+\Delta t) = 2\mathbf{r}(t)-\mathbf{r}(t-\Delta t)+\mathbf{a}(t)(\Delta t)^2+O[(\Delta t)^4] \tag{2.24}$$

Esta é a forma básica do método de *Verlet*. Como é imediato ver que o erro de truncamento desse método com o sistema evoluindo em Δt é da ordem de $(\Delta t)^4$, mesmo a derivada terceira não aparecendo explicitamente.

Um problema com esta versão do método de *Verlet* é que as velocidades não são geradas diretamente. Mesmo elas não sendo necessárias para a evolução temporal, saber as velocidades no instante de tempo atual pode ser necessário, por exemplo, para calcular a energia cinética total K e avaliar se a simulação está sendo feita corretamente. Pode-se pensar em calcular as velocidades da seguinte maneira:

$$\mathbf{v}(t) = \frac{\mathbf{r}(t+\Delta t)-\mathbf{r}(t-\Delta t)}{2\Delta t} \tag{2.25}$$

No entanto, o erro associado a essa expressão é da ordem de Δt^3 ao invés de Δt^4.

Uma variante ainda melhor do método básico de *Verlet* é conhecido como *esquema velocidade de Verlet*, onde posições, velocidades e acelerações são obtidos a partir das mesmas quantidades no instante t:

$$\begin{aligned}\mathbf{r}(t+\Delta t) &= \mathbf{r}(t)+\mathbf{v}(t)\Delta t+\frac{1}{2}\mathbf{a}(t)(\Delta t)^2 \\ \mathbf{v}(t+\Delta t/2) &= \mathbf{v}(t)+\frac{1}{2}\mathbf{a}(t)\Delta t \\ \mathbf{a}(t+\Delta t) &= -\frac{1}{m}\nabla U(\mathbf{r}(t+\Delta t)) \\ \mathbf{v}(t+\Delta t) &= \mathbf{v}(t+\Delta t/2)+\frac{1}{2}\mathbf{a}(t+\Delta t)\Delta t\end{aligned} \tag{2.26}$$

É importante observar que o método *Leapfrog* é uma variante do método de *Verlet*.

2.5.3 Método Preditor-Corretor

Métodos do tipo Preditor-Corretor (PC) constituem outra classe comum de métodos de integração das equações de movimento (Eq. 2.4). Os métodos baseados no PC consistem de três passos, a saber (ALLEN; TILDESLEY, 1987):

- **Preditor:** A partir das posições ($\mathbf{r}^N$) e suas derivadas (por exemplo, **v** e **a**) até uma certa ordem q, todas conhecidas no instante de tempo t, é feita uma *predição* das novas posições e derivadas no instante $t+\Delta t$ através de uma expansão em série de Taylor.
- **Avaliação das forças:** As forças são calculadas tomando o negativo do gradiente nas posições previstas (Eq. 2.3). A aceleração resultante será, em geral, diferente da aceleração prevista. A diferença entre as duas constitui o *erro*.
- **Corretor:** O erro calculado é utilizado para *corrigir* as posições e suas derivadas. Todas as correções são proporcionais ao erro que é o coeficiente de proporcionalidade usado para determinar a estabilidade do algoritmo.

Sendo $P(x)$ e $C(x)$ os passos preditor e corretor, respectivamente, a formulação do método PC para a equação $\dot{x} = f(x,t)$ é dada por (RAPAPORT, 1996):

$$P(x) : x(t+\Delta t) = x(t) + \Delta t \sum_{i=1}^{k} \alpha_i f(t + [1-i]\Delta t) \tag{2.27}$$

$$C(x) : x(t+\Delta t) = x(t) + \Delta t \sum_{i=1}^{k} \beta_i f(t + [2-i]\Delta t) \tag{2.28}$$

Com os coeficientes que satisfazem as seguintes expressões:

$$\sum_{i=1}^{k} (1-i)^q \alpha_i = \frac{1}{q+1} \tag{2.29}$$

$$\sum_{i=1}^{k} (2-i)^q \beta_i = \frac{1}{q+1} \tag{2.30}$$

Os coeficientes, para $k = 4$, estão tabulados em (RAPAPORT, 1996). Como o presente trabalho não trata da implementação deste método em particular, será usado apenas para efeitos de ilustração com os demais métodos de integração das equações de movimento.

2.5.4 Comparação dos métodos

Devido a sua grande flexibilidade e uma potencial alta precisão local, fazem do método PC mais apropriado para problemas mais complexos, como corpos rígidos ou dinâmica com restrições, onde uma grande precisão em cada passo de tempo é desejável, ou onde as equações de movimento incluem forças dependentes de velocidades. A abordagem do método *Leapfrog* é mais simples e requer um armazenamento mínimo, mas possui a desvantagem de precisar de adaptações para diferentes problemas. O método *Leapfrog* fornece excelentes propriedades de conservação de energia mesmo com potenciais altamente repulsivos a pequenas distância, como o potencial de LJ, mesmo com um Δt grande. Outra vantagem do método *Leapfrog* é o seu baixo requerimento de armazenamento de informações. Para simulações onde armazenamento se seja um problema importante, este pode ser o método mais adequado (RAPAPORT, 1996).

3 *Simulação de Dinâmica Molecular*

Neste capítulo serão apresentados os conceitos e técnicas necessários para implementação de um simulador de DM.

Ao observar o modelo físico das interações interatômicas percebe-se que a resolução do problema dos N-Corpos (interação de um para todos) é $O(N^2)$ onde N é o número de corpos do sistema. Esse comportamento quadrático leva a tempos proibitivos à medida que N cresce. As técnicas de computação buscam reduzir o número de cálculos necessários para computar a força resultante em cada átomo. Divisão por células e listas de vizinhos são duas técnicas fortemente utilizadas para reduzir a complexidade do problema dos N-Corpos.

Na simulação de sistemas atômicos se torna evidente o problema do uso de unidades associadas medidas microscópicas. Normalmente, as distâncias atômicas são medidas em *Angstroms* ($1\mathring{A} = 1.0 \times 10^{-10}m$). A dificuldade de se trabalhar com números tão pequenos está limitada pela capacidade de representação numérica da máquina escolhida. Para contornar esse problema utilizam-se *unidades adimensionais*. Todas as quantidades físicas são expressas em termos de tais unidades a fim de trabalhar com números próximos da unidade.

3.1 Unidades Adimensionais

Unidades adimensionais (UA), ou unidades reduzidas são as unidades escolhidas para representar todas as quantidades físicas necessárias. Por exemplo, pode-se escolher como unidade de distância $10\mathring{A}$, isso implica que cada unidade de distância do modelo representa $10\mathring{A}$ em unidades reais.

Há várias razões para se fazer uso de UA. Uma das razões mais simples é tirar o máximo de proveito da facilidade de se trabalhar com números próximos da unidade ao invés de números extremamente pequenos associados com a escala atômica. Outro benefício do uso de UA é a simplificação das equações de movimento que regem o sistema devido ao fato de seus parâmetros, na maior parte dos casos, serem absorvidos pelas UA. Do ponto de vista prático, o

uso de UA elimina o risco de surgirem valores que não podem ser representados pelo *hardware* de um computador (RAPAPORT, 1996).

As UA mais adequadas para estudos de DM baseados no potencial de LJ (Eq. 2.5) são aquelas baseadas na escolha de σ, ε e m como unidades de comprimento, energia e massa, respectivamente. A escolha de tais unidades, em termos práticos, implica que para as equações de movimento (Eq. 2.4) e 2.6), $\sigma = \varepsilon = m = 1$. As seguintes substituições se tornam necessárias para representação do modelo em UA: $r \rightarrow r\sigma$ para comprimento, $e \rightarrow e\varepsilon$ para energia e $t \rightarrow t\sqrt{m\sigma^2/\varepsilon}$ para o tempo. A forma final das equações de movimento, energia cinética e energia potencial ficam, respectivamente (RAPAPORT, 1996):

$$\ddot{\mathbf{r}}_i = 48 \sum_{\substack{j=1 \\ j \neq i}}^{N} \left(r_{ij}^{-14} - \frac{1}{2} r_{ij}^{-8} \right) \mathbf{r}_{ij} \tag{3.1}$$

$$\mathscr{E}_{\mathscr{K}} = \frac{1}{2N} \sum_{i=1}^{N} \mathbf{v}_i^2 \tag{3.2}$$

$$\mathscr{E}_{\mathscr{U}} = \frac{4}{N} \sum_{1 \leq i < j \leq N} \left(r_{ij}^{-12} - r_{ij}^{-6} \right) \tag{3.3}$$

Para simular um sistema de argônio líquido, as relações entre as unidades adimensionais e as unidades físicas reais são as seguintes (RAPAPORT, 1996): distâncias são medidas em função de $\sigma = 3.4\mathring{A}$; a unidade de energia é definida por $\varepsilon/K_b = 120K$, sabendo que $K_b = 1$, as unidades de energia ficam em função de $\varepsilon = 8.314J/mole$; sabendo que a massa atômica do argônio é $m = 39.95 \times 1.6747 \times 10^{-27}kg$, então a unidade de tempo adimensional corresponde a $2.161 \times 10^{-12}s$. Tipicamente, um $\Delta t = 0.005$ (em unidades adimensionais) é utilizado como passo de tempo (da ordem de 10 femtosegundos). A densidade do argônio líquido é $0.942g/cm^3$. Para uma região cúbica de lado L, contendo N átomos, implica que $L = 4.142N^{1/3}\mathring{A}$, o que em UA fica $L = 1.128N^{1/3}$.

3.2 Computação de Interações

Ao olhar para a Equação 2.7 que define como os átomos interagem entre si percebe-se que são necessárias $O(N^2)$ computações para calcular a força resultante sobre cada um dos N de um sistema. Todos os pares de átomos devem ser examinados pois não se conhece a priori quais átomos interagem com quais. A contínua mudança da configuração (posições relativas) dos átomos é uma característica do estado líquido da matéria.

Duas técnicas muito usadas em simulações de DM são apresentadas. O objetivo de tais técnicas é reduzir o número de computações necessárias para avaliar as forças resultantes atuantes sobre os átomos. Ambas as técnicas tiram proveito da anulação do potencial de interação a partir de certo ponto (Fig. 2.1) denominada raio de corte r_c, portanto, átomos que estão separados por um distância maior do que o raio de corte podem ser desconsiderados no cálculo da força.

Para o potencial de LJ a distância de corte r_c típica é definida em função de σ como (RAPAPORT, 1996):

$$r_c = 2^{1/6}\sigma \tag{3.4}$$

3.2.1 Listas de Vizinhos

A técnica da lista de vizinhos consiste em construir uma lista de átomos para cada átomo do sistema que esteja a uma distância menor ou igual do que a distância de corte r_c. Seja $\mathscr{N}_i = \{j_1, j_2, \ldots j_m\}, m < N$ a lista de vizinhos do átomo i, tem-se que um átomo j qualquer pertence a $\mathscr{N}_i$ se, e somente se, $r_{ij} \leq r_c$.

O cálculo da força total atuante sobre um átomo i exige uma modificação da Equação 2.7 para que o somatório seja feito somente sobre os vizinhos do átomo i:

$$\ddot{\mathbf{r}}_i = \sum_{j \in \mathscr{N}_i}^{|\mathscr{N}_i|} \left(r_{ij}^{-14} - \frac{1}{2} r_{ij}^{-8} \right) \mathbf{r}_{ij} \tag{3.5}$$

É importante notar que o cálculo da lista de vizinhos que considera os átomos que estão a uma distância $r_{ij} \leq r_c$ é ineficiente no sentido de que as listas geradas devem ser reavaliadas a cada iteração do sistema. A redução da complexidade da avaliação das forças resultantes para $O(N)$ não apresenta um ganho significativo quando comparada a complexidade para cálculo da lista de vizinhos que é $O(N^2)$. Para contornar esse problema define-se uma nova distância de corte r_n como:

$$r_n = r_c + \Delta r \tag{3.6}$$

O valor de Δr é inversamente proporcional a taxa com que a lista de vizinhos deve ser reconstruída. A decisão de reconstruir a lista de vizinhos é baseada no monitoramento da velocidade máxima dos átomos a cada passo de tempo até que a condição (Eq. 3.7) seja satisfeita

(RAPAPORT, 1996).

$$\sum_{i=0}^{N} \left(\max_i |\mathbf{v}_i| \right) > \frac{\Delta r}{2\Delta t} \tag{3.7}$$

onde Δt é o intervalo de tempo escolhido para integração das equações de movimento (Eq. 2.4).

O sucesso do uso dessa abordagem está na mudança lenta do estado do sistema em nível microscópico o que implica que a lista de vizinhos permanece válida por certo número de iterações - tipicamente entre 10 e 20 mesmo com um Δr pequeno.

3.2.2 Divisão por Células

A divisão por células fornece um meio de organizar a informação sobre a posição dos átomos em uma forma que evita a maior parte do trabalho necessário e reduz o esforço computacional a $O(N)$. A técnica consiste na divisão da região de simulação em uma grade de pequenas células. Cada célula possui aresta maior que r_c em comprimento. Essa exigência não é obrigatória, mas torna o método eficiente e fácil de ser implementado.

Seja $\mathscr{C} = \{c_1, c_2, \cdots, c_m\}$ o conjunto de células que compõe a região de simulação. Seja o par de átomos i e j. Se $i \in c_k$ onde c_k é uma célula qualquer, com a divisão por células é claro que i interage com j somente se j estiver localizado nas células imediatamente adjacentes ou no próprio c_k. Se nenhuma das condições anteriores for satisfeita, isto é, se j está a mais de uma célula de distância de c_k, então é claro que $r_{ij} > r_c$, o que implica que i e j não interagem.

Devido a simetria proporcionada por essa divisão, somente metade das células vizinhas precisam ser consideradas (um total 14 para três dimensões e 5 para duas dimensões, incluindo a célula a qual o átomo pertence).

3.2.3 Comparação das Técnicas

A escolha de uma ou outra técnica depende das facilidades oferecidas pela linguagem de programação escolhida. A técnica da lista de vizinhos é eficiente quando a linguagem oferece suporte a ponteiros. O uso de ponteiros permite a criação de uma lista de vizinhos dinâmica, que pode crescer ou se retrair ao longo do tempo, poupando recursos de armazenamento. A técnica de divisão por células é mais eficiente em termos de varredura espacial. Enquanto a técnica da lista de vizinhos ainda precisa varrer todo o espaço para verificar quais átomos estão dentro do raio de corte r_c, o que ainda demanda custo de tempo de $O(N^2)$. A técnica da divisão por células necessita de $O(N)$, o que o torna mais eficiente.

Somente uma pequena fração dos átomos examinados pelo método de divisão por células estão no alcance de interação de um átomo *i* qualquer. Uma abordagem híbrida pode ser empregada de forma a tirar proveito das vantagens de ambas. A abordagem híbrida consiste em criar uma lista de vizinhos, como mostrado na técnica de lista de vizinhos (Sec. 3.2.1) mas somente verificando os átomos que estão nas células imediatamente adjacentes (Sec. 3.2.2).

3.3 Inicialização do Sistema

A preparação do estado inicial do sistema normalmente é feita em dois passos, um para inicialização das coordenadas e outro para inicialização das velocidades de cada átomo do sistema. As coordenadas podem ser inicializadas de duas formas diferentes: aleatoriamente ou em uma grade regular. As velocidades são inicializadas de modo aleatório e um fator de escalamento é usado para atingir a temperatura desejada.

3.3.1 Inicialização das Coordenadas

Inicialmente o espaço é discretizado em células de igual tamanho onde cada célula pode conter um ou mais átomos. No caso em que existe somente um átomo por célula, tal átomo é posicionado no centro dessa célula. O algoritmo que inicializa as coordenadas recebe como parâmetro valores inteiros que dizem quantos átomos existem por linha e coluna, no caso bidimensional, e profundidade, no caso tridimensional.

3.3.2 Inicialização das Velocidades

A inicialização da velocidade é feita em três passos. O primeiro consiste no cálculo do módulo da velocidade v_{mag} de cada átomo do sistema que depende diretamente da temperatura do seguinte modo:

$$v_{mag} = \sqrt{\frac{dk_bT}{mN}} \tag{3.8}$$

onde N é o número de átomos do sistema. Considera-se que o sistema é composto por N átomos de massa m e d é a dimensão física do sistema.

O segundo passo consiste na atribuição de um vetor unitário aleatório a cada átomo que determinará a direção da velocidade de cada molécula. Para gerar tal vetor são necessários dois números aleatórios ξ_1 e ξ_2 tais que:

$$\omega_{\mathbf{d}}(\theta,\phi) = (\pi\xi_1, 2\pi\xi_2) \tag{3.9}$$

Calculados θ e ϕ, para cada vetor velocidade v_i associado ao átomo i define-se o vetor unitário aleatório como:

$$\begin{aligned} v_{i_x} &= \sin\theta\cos\phi \\ v_{i_y} &= \sin\theta\sin\phi \\ v_{i_z} &= \cos\theta \end{aligned} \tag{3.10}$$

Para o caso bidimensional basta eliminar a componente v_{i_z} do vetor $\mathbf{v}_i$ e fazer $\theta = \pi/2$. Isto garante um vetor unitário aleatório com distribuição uniforme sobre uma esfera. O terceiro e último passo consiste no escalamento do vetor obtido $\mathbf{v}_i$ pelo módulo da velocidade de cada átomo v_{mag}:

$$\mathbf{v}_i = v_{mag}\mathbf{v}_i \tag{3.11}$$

3.4 Passo de Simulação

O passo de simulação é a rotina executada a cada iteração que define a evolução do sistema. A Figura 3.1 mostra o diagrama de um passo de simulação típico de um sistema de DM. Primeiramente o sistema é devidamente inicializado como descrito na Seção 3.3. O primeiro passo da rotina que determina a evolução temporal do sistema é o cálculo do momento linear total $\mathbf{P}$, que nada mais é do que o vetor resultante da soma dos vetores velocidades dos N átomos:

$$\mathbf{P} = \sum_{i=1}^{N} m_i\mathbf{v}_i \tag{3.12}$$

O próximo passo consiste no cálculo do centro de massa do sistema. Na mecânica clássica o centro de massa de um corpo é o ponto onde pode ser pensado que toda a massa do corpo está concentrada para o cálculo de vários efeitos. O centro de massa $\mathbf{C}_m$ de um sistema de N átomos é calculado como:

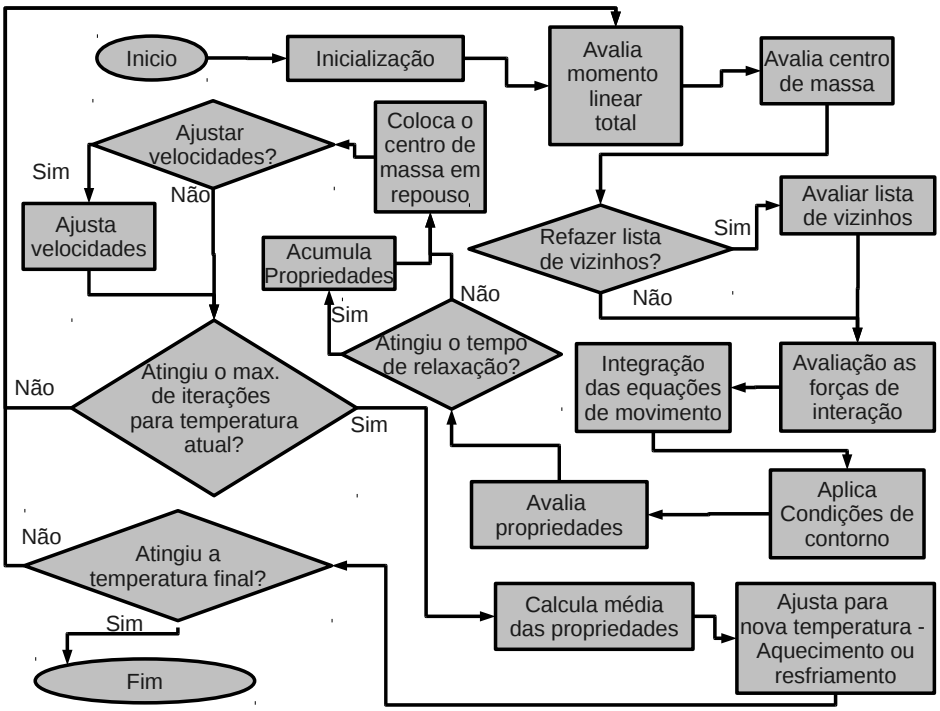

Figura 3.1: Diagrama do passo de simulação.

$$\mathbf{C}_m = \frac{1}{M}\sum_{i=1}^{N} m_i \mathbf{r}_i, \mathbf{r}_i \in \mathbf{r}^N \tag{3.13}$$

onde $\mathbf{r}_i$ é a posição de cada átomo de massa m_i. O centro de massa pode ser usado para uma avaliação visual do comportamento do sistema. O centro de massa deve permanecer em repouso no centróide da região de simulação ao longo do tempo. Caso seja observado algum tipo de anomalia quando ao seu posicionamento, sabe-se que algo está errado.

Como descrito na Seção 3.2.1, não existe a necessidade de reavaliar a lista de vizinhos a cada iteração. Um teste é feito para verificar se a lista deve ou não ser refeita. O intervalo de iterações entre duas construções da lista de vizinhos é tomado como parâmetro da simulação. Tipicamente esse valor é um entre 10 e 20 iterações.

A avaliação das forças de interação entre os átomos é então realizada como descrito na Seção 3.2. O passo seguinte consiste na integração numérica das equações do movimento que regem o sistema (Sec. 2.5). Em seguida as propriedades do sistema são avaliadas (Sec. 2.4) e um teste se o sistema atingiu o tempo de relaxação é feito. Se esse tempo foi atingido, implica que o sistema atingiu seu estado de equilíbrio, então as propriedades avaliadas anteriormente podem ser acumuladas para extração da média ao final da simulação.

Uma rotina que coloca o centro de massa em repouso é inserida no passo de simulação para

garantir que não haja deslocamento de massa do sistema. Isto é feito por um ajuste na velocidade de cada átomo do sistema. Um fator de ajuste $\mathscr{V}$ é obtido apartir da soma dos momentos individuais dos átomos:Uma rotina que coloca o centro de massa em repouso é inserida no passo de simulação para garantir que não haja deslocamento de massa do sistema. Isto é feito por um ajuste na velocidade de cada átomo do sistema. Um fator de ajuste $\mathscr{V}$ é obtido a partir da soma dos momentos individuais dos átomos:

$$\mathscr{V} = \frac{1}{M}\sum_{i=1}^{N} m_i \mathbf{v}_i \tag{3.14}$$

as velocidades são então ajustadas pela subtração de cada $\mathbf{v}_i$ pelo fator $\mathscr{V}$, ou seja, para cada átomo do sistema i temos $\mathbf{v}_i = \mathbf{v}_i - \mathscr{V}$.

O ajuste de velocidades é necessários para trazer o sistema para a temperatura desejada T. Este ajuste exige um re-escalamento das velocidades por um fator v_{fac} calculado em função da temperatura instantânea T_i:

$$v_{fac} = \sqrt{\frac{T}{T_i}} \tag{3.15}$$

a taxa com que esse ajuste deve ocorrer é, na maioria dos casos, determinada empiricamente.

As simulações de DM ocorrem para uma faixa de temperatura. O valor estabelecido como sendo o número máximo de iterações na verdade é para cada temperatura. O penúltimo teste a ser feito é se o número de iterações até o momento atingiu o número máximo para uma temperatura. Se isto ocorreu, é calculada a média das propriedades medidas até o momento, estes valores de médias são salvos em um arquivo para análise posterior. O próximo passo consiste no ajuste da temperatura para um novo valor, o que depende da variação de temperatura estabelecida e se o sistema está esfriando ou aquecendo. Após o ajuste, é verificado se a temperatura atingiu o seu valor máximo ou mínimo, em caso positivo, a simulação terminou e os dados gerados podem ser submetidos para análise, caso contrário a velocidade total é reavaliada e o ciclo recomeça.

3.5 Medidas de Propriedades Termodinâmicas

Um sistema de DM é inicializado com uma temperatura nominal T. As coordenadas dos N átomos do sistema são inicializadas assim como suas velocidades. Porém, a configuração inicial do sistema como definida na Seção 3.3, não necessariamente é a configuração de equilíbrio do sistema. Caracterizar o estado de equilíbrio não é uma tarefa fácil, especialmente em sistemas

pequenos onde as propriedades flutuam consideravelmente (RAPAPORT, 1996) [1]. A Figura 3.2 apresenta a variação das energias cinética e potencial ao longo do tempo para 250 iterações de um sistema com $N = 1000$ átomos. É possível ver que o sistema atingiu o estado de equilíbrio em aproximadamente 50 iterações.

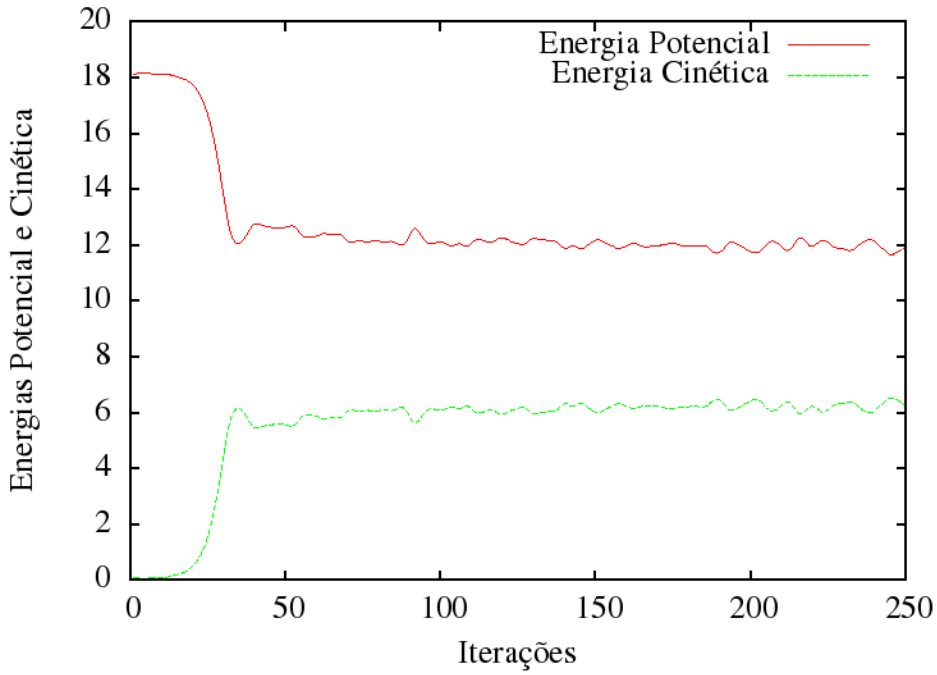

Figura 3.2: Variação da energia cinética e potencial em função do tempo. $N = 1000$, densidade$= 1.2g/cm^3$ e temperatura$= 30k$.

Medir propriedades termodinâmicas (Sec. 2.4) sobre uma série de passos ao longo do tempo e depois obter sua média reduz a influência da flutuação dessa propriedade. Ao tempo necessário para o sistema atingir o estado de equilíbrio é dado o nome de *tempo de relaxação*. Na maior parte dos sistemas o tempo de relaxação é atingido rapidamente. A partir do momento em que o tempo de relaxação foi atingido, dá-se inicio à *fase de produção*, onde as propriedades termodinâmicas são realmente medidas. Ao tempo de espera entre uma medição e outra é dado o nome de *tempo para medição*. Na fase de produção as propriedades de interesse são avaliadas considerando o estado do sistema (posições e velocidades) e em seguida são acumulados até que o número máximo de passos de tempo estabelecido para a simulação é atingido. O quadrado desses valores são também acumulados para que o cálculo da variância de cada propriedade medida seja avaliada.

[1] Métodos para determinar quando um sistema mais complexo atinge o equilíbrio podem ser encontrados em (RAPAPORT, 1996)

4 *Simulação de Dinâmica Molecular em Unidades de Processamento Gráfico de Propósito Geral*

Este capítulo tem por objetivo descrever a implementação da simulação de DM em unidades gráficas de propósito geral. A Seção 4.1 descreve o modelo de programação *CUDA* e a Seção 4.3 descreve a implementação dos cálculos de interações interatômicas usando o modelo de programação *CUDA*.

4.1 *GPU* : Um Processador com Muitos Núcleos, Multithread e Altamente Paralelo

Unidades ou placas gráficas se tornaram poderosos processadores massivamente paralelos que, usados como co-processadores, são usadas para resolver problemas complexos de maneira eficiente. Um bom motivo para o uso de placas gráficas é encontrado em (NVIDIA, 2009):

> Guiado por um mercado insaciável por demanda de aplicações gráficas de alta definição e em tempo real, as unidades gráficas programáveis evoluíram em processadores com tremendo poder computacional com muitos núcleos altamente paralelos e alta banda de memória.

Em novembro de 2006 a *NVIDIA* apresentou o *CUDA (Compute Unified Device Archtecture)*, um novo modelo de programação paralela que permite acesso ao poder computacional de suas das placas gráficas.

Quando os computadores surgiram, eram máquinas de grande porte que ocupavam salas inteiras e necessitavam de mão-de-obra especializada para realizar sua operação. A saída de programas que eram executados em tais computadores era puramente textual, sem nenhum tipo de elemento gráfico. Quando, nos anos 60, começou-se a usar uma tela de raios catódicos para exibir a saída gerada pelos computadores, tornou-se necessário algum tipo de processamento gráfico a fim de gerar os elementos necessários para exibição em tais telas (CAMPOS, 2008).

Com o surgimento dos novos sistemas operacionais que faziam uso de interface gráfica nos anos 90, ficou evidente que as interfaces de vídeo desenvolvidas até então, não suportavam todo o processamento que as era delegado. Até então as placas gráficas somente exibiam na tela o que estava em sua memória. Todo o processamento de transformação das informações em pixels ficava a cargo das CPU's (unidades centrais de processamento) e FPU's (unidades de ponto flutuante). A solução para este problema surgiu em 1994 quando surgiram as primeiras placas aceleradoras gráficas 2D. Elas foram os primeiros dispositivos a possuir uma unidade dedicada ao processamento gráfico: A *GPU* , ou Unidade de Processamento Gráfico, que é um processador dedicado à resolução de operações em ponto flutuante que são a base para a construção de imagens.

Com a exigência cada vez maior de aplicações gráficas de tempo real, tanto CPU's quando *GPUs* evoluíram para processadores multinucleados com tremendo poder computacional, o que significa que ambos são, hoje, sistemas paralelos. Além do mais, esse paralelismo continua crescendo com a lei de Moore. O desafio atual é desenvolver aplicações que usem esse paralelismo de forma transparente de modo a tirar proveito do crescente aumento do número de núcleos das *GPUs* .

A evolução do poder computacional das *GPUs* da NVIDIA em comparação com as CPU's é visto na Figura 4.1. Como pode ser visto, as novas arquiteturas tem capacidade de processamento de quase 1 TFlop/s, usando somente uma placa gráfica de custo relativamente baixo.

Para atingir tal capacidade, a largura de banda entre as *GPUs* e sua memória sofreu um aumento significativo, como visto na Figura 4.2, ultrapassando a casa dos 100 GB/s.

A razão para essa discrepância na capacidade de operações em ponto flutuante entre CPU e *GPU* está no fato de que a *GPU* é especializada em computação altamente paralela e intensiva, além do mais, a *GPU* é desenhada de forma a dedicar mais transistores para processamento de dados ao invés de cache e controle de fluxo, como ilustrado pela Figura 4.3.

Mais especificamente, a *GPU* é especialmente adequada para problemas que podem ser expressos em computação paralela de dados, isto é, o mesmo programa executado em porções de dados diferentes. Isto elimina a necessidade de um controle de fluxo sofisticado, como o existente nas CPU's. A largura de banda existente elimina a necessidade de uma cache muito grande.

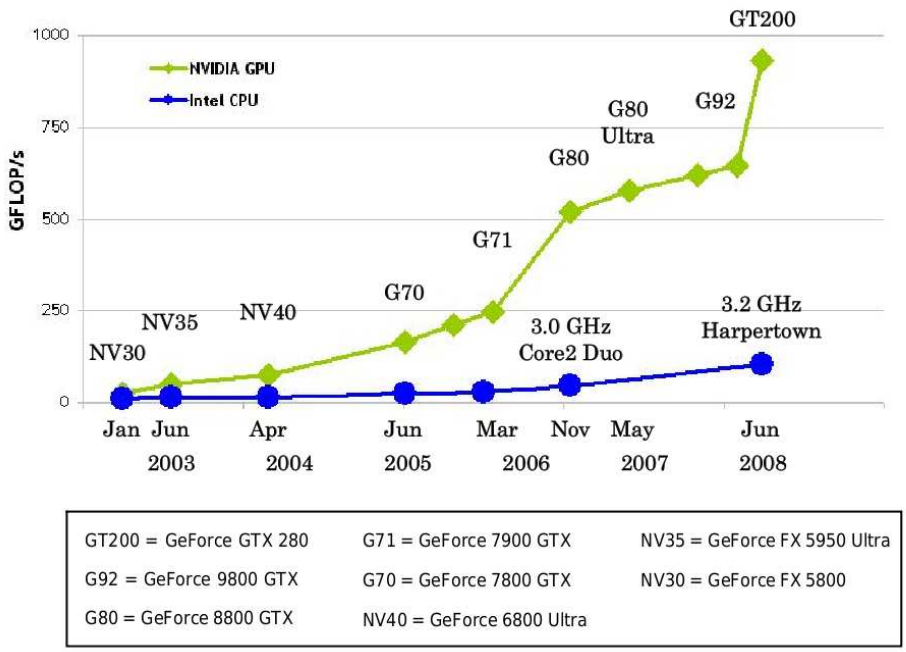

Figura 4.1: Comparação do número de operações em ponto flutuante por segundo entre CPU e *GPU* (NVIDIA, 2009).

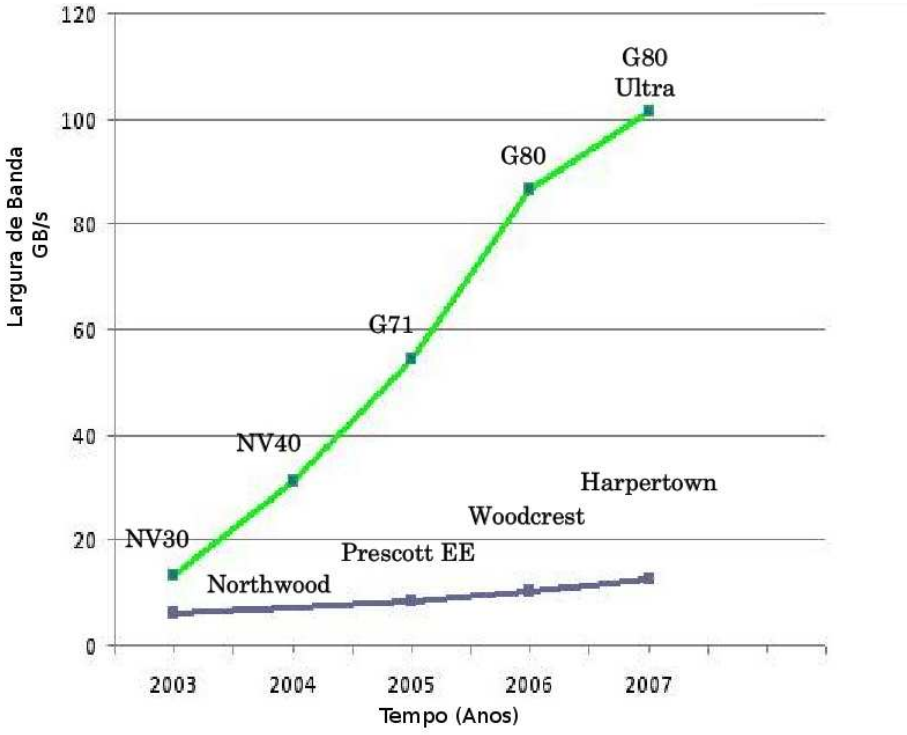

Figura 4.2: Comparação da largura de banda entre entre processador e memória principal para CPU e *GPU* (NVIDIA, 2009).

4.1.1 *CUDA*: Um Modelo de Programação Paralela Escalável

Apresentado em 2006 pela *NVIDIA*, o *CUDA* é um modelo de programação paralela e um ambiente de *software* planejado para auxiliar desenvolvedores a enfrentar o desafio de escrever

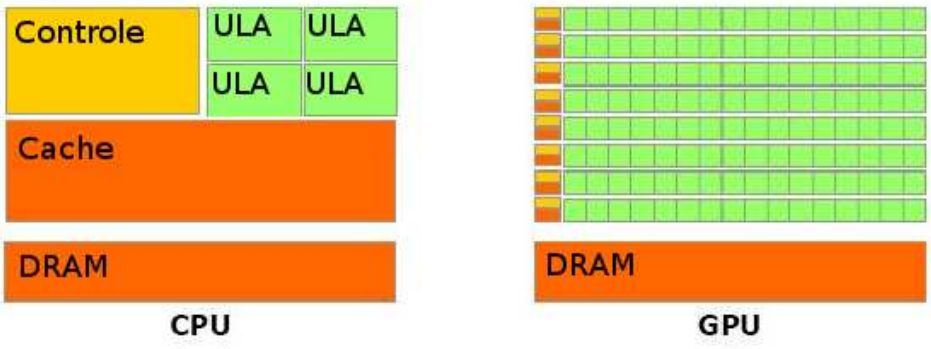

Figura 4.3: Comparação das estruturas internas entre *GPU* e CPU: A *GPU* dedica mais transistors ao processamento de dados (NVIDIA, 2009).

softwares paralelos escaláveis enquanto mantém uma curva de aprendizado baixa para os que estão familiarizados com linguagens de programação padrão, como o *C*.

Existem três abstrações chaves como núcleo - uma hierarquia de grupos de *threads*, memória compartilhadas e uma barreira de sincronização - que são expostas de forma transparente ao desenvolvedor como simples extensões da linguagem *C*. Estas abstrações fornecem um paralelismo com granularidade alta tanto de dados quanto de *threads*. Essa granularidade guia o desenvolvedor a particionar o problema de interesse em pequenos problemas que podem ser resolvidos independentemente em paralelo (NVIDIA, 2009).

Um programa *CUDA* compilado pode, além do mais, ser executado em qualquer quantidade de núcleos, e somente o sistema em tempo de execução precisa saber o número físico de processadores.

4.2 Modelo de Programção

O *CUDA* extende a linguagem de programação *C* permitindo que o programador possa definir funções em *C* chamadas de *kernels*, que, quando invocadas, são executadas *N* vezes em paralelo por *N* diferentes *threads CUDA* ao invés de somente uma vez como funções *C* regulares (NVIDIA, 2009).

Um *kernel* é definido usando a palavra-chave **__global__** na assinatura de uma função. O número de *threads CUDA* para cada chamada é definido usando a sintaxe <<< ... >>> como ilustrado a seguir:

```
// Definição de um kernel
__global__ void vecAdd( float *A, float *B, float *C ) {
```

```
}

int main( int argc, char *argv[] ) {
    // Chamada ao kernel
    vecAdd<<<1, N>>>( A, B, C );
}
```

No exemplo anterior um *kernel* chamado *vecAdd* é definido e na função *main* é chamado para ser executado *N* vezes em paralelo por *N threads CUDA*.

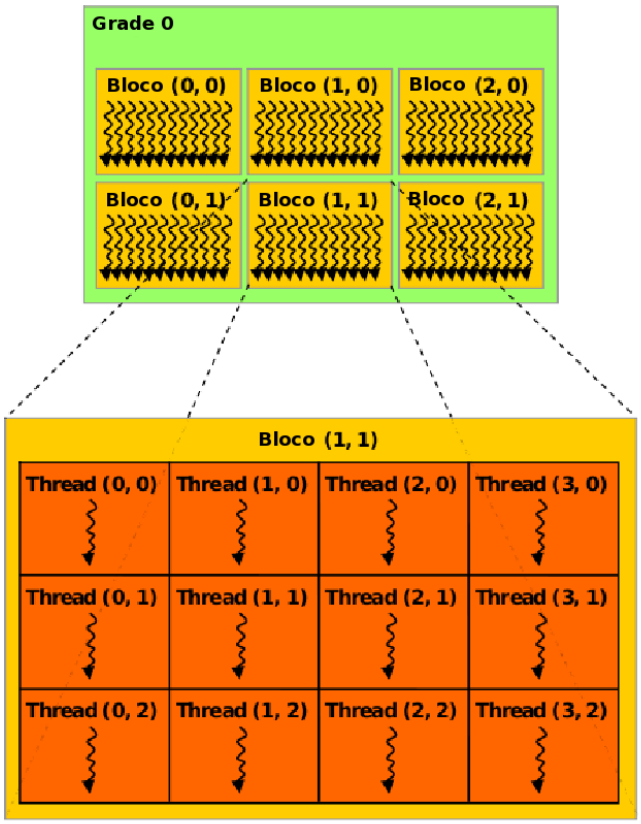

Figura 4.4: Diagrama de um grid de blocos de threads (NVIDIA, 2009).

4.2.1 Hierarquia de *Threads*

O *CUDA* trabalha com uma hierarquia de *threads* permitindo ao programador estabelecer como deve ser a divisão do trabalho de um programa de forma a utilizar de forma eficiente o *hardware* gráfico. A Figura 4.4 mostra como é estruturada a hierarquia de *threads*. A cada

kernel executado é atribuída uma **grade** de **blocos** de *threads*. A **grade** é composta por vários **blocos** e cada **bloco** é composto por várias *threads*.

Cada *thread* criada pelo *CUDA* recebe um identificador que pode ser usado para acesso à memória ou mesmo como um fator em alguma operação aritmética. Cada *thread* tem acesso a um conjunto de variáveis definidas pelo sistema em tempo de execução, são elas:

- **threadIdx**: Índice da *thread* dentro do bloco;
- **blockDim**: Dimensão do bloco ao qual a *thread* pertence;
- **blockIdx**: Índice do bloco dentro da grade de blocos;
- **gridDim**: Dimensão da grade à qual o bloco pertence.

Por conveniência, a variável **threadIdx** é um vetor tridimensional. Isto fornece um meio natural de navegar através dos elementos de um domínio, como vetores, matrizes ou campos. O código a seguir ilustra a utilização dos índices da variável **threadIdx** para realizar uma soma de duas matrizes A e B, ambas de tamanho $N \times N$, e armazena o resultado em uma matriz C:

```
__global__ void matAdd( float *A, float *B, float *C ) {
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main( int argc, char *argv[] ) {
    // Define a dimensão do bloco de threads
    dim3 dimBlock(N, N);

    // Chamada do kernel, 1 bloco de N x N threads
    matAdd<<<1, dimBlock>>>(A, B, C);
}
```

O índice de uma *thread* com seu identificador estão diretamente relacionados um com o outro: Para um bloco unidimensional eles são o mesmo, para um bloco bidimensional de tamanho (D_x, D_y), o identificador da *thread* de *índice* (x, y) é $(x + yD_x)$, para um bloco tridimensional de tamanho (D_x, D_y, D_z), o identificador de uma *thread* de índice (x, y, z) é $(x + yD_x + zD_xD_y)$.

As *threads* que residem dentro de um bloco podem cooperar entre si através de uma *memória compartilhada* e podem também sincronizar sua execução para coordenar acessos a memória. Pode-se especificar um ponto de sincronização dentro de um *kernel* através de uma chamada a função **__syncthreads()**. A função **__syncthreads()** atua como uma barreira onde todas as *threads* dentro de um bloco devem esperar até que todas as outras tenham atingido essa barreira para continuar sua execução.

4.2.2 Hierarquia de Memória

As *threads CUDA* podem acessar dados de múltiplos espaços de memória durante sua execução, como ilustrado na Figura 4.5. Cada *thread* possui um espaço privado de memória, cada bloco possui uma memória em que todas as *threads* têm acesso e que posssui o tempo de vida daquele bloco. Finalmente, todas as *threads* tem acesso a mesma memória global.

4.2.3 *Hospedeiro* e *Dispositivo*

O *CUDA* assume que as *threads* são executadas em um *dispositivo* fisicamente separado, que opera basicamente como um co-processador do *hospedeiro* executando o programa *C*. Este é o caso, por exemplo, quando *kernels* executam em uma *GPU* e o restante do programa *C* executa em uma *CPU*.

O *CUDA* também assume que ambos o hospedeiro e o dispositivo possuem sua própria DRAM, referenciadas como *memória do hospedeiro* e *memória do dispositivo*, respectivamente. Um programa pode gerenciar os espaços de memória visíveis aos *kernels* através de chamadas ao *CUDA runtime*. Isto inclui alocação e dealocação de memória do dispositivo assim como transferência entre as memórias do dispositivo e hospedeiro.

4.3 Cálculo das Interações Interatômicas em *CUDA*

O cálculo das interações interatômicas pode ser encarado como uma interação de todos para todos, o que faz com que o problema possa ser tratado como o clássico problema dos N-Corpos. A primeira formulação completa deste problema foi proposta por (NEWTON, 1686), originalmente pensada para o estudo de interações gravitacionais de corpos celestes.

O problema dos N-Corpos é relativamente simples de resolver, mas não é utilizado como descrito em sistemas de larga escala devido a sua complexidade computacional $O(N^2)$. A seguir

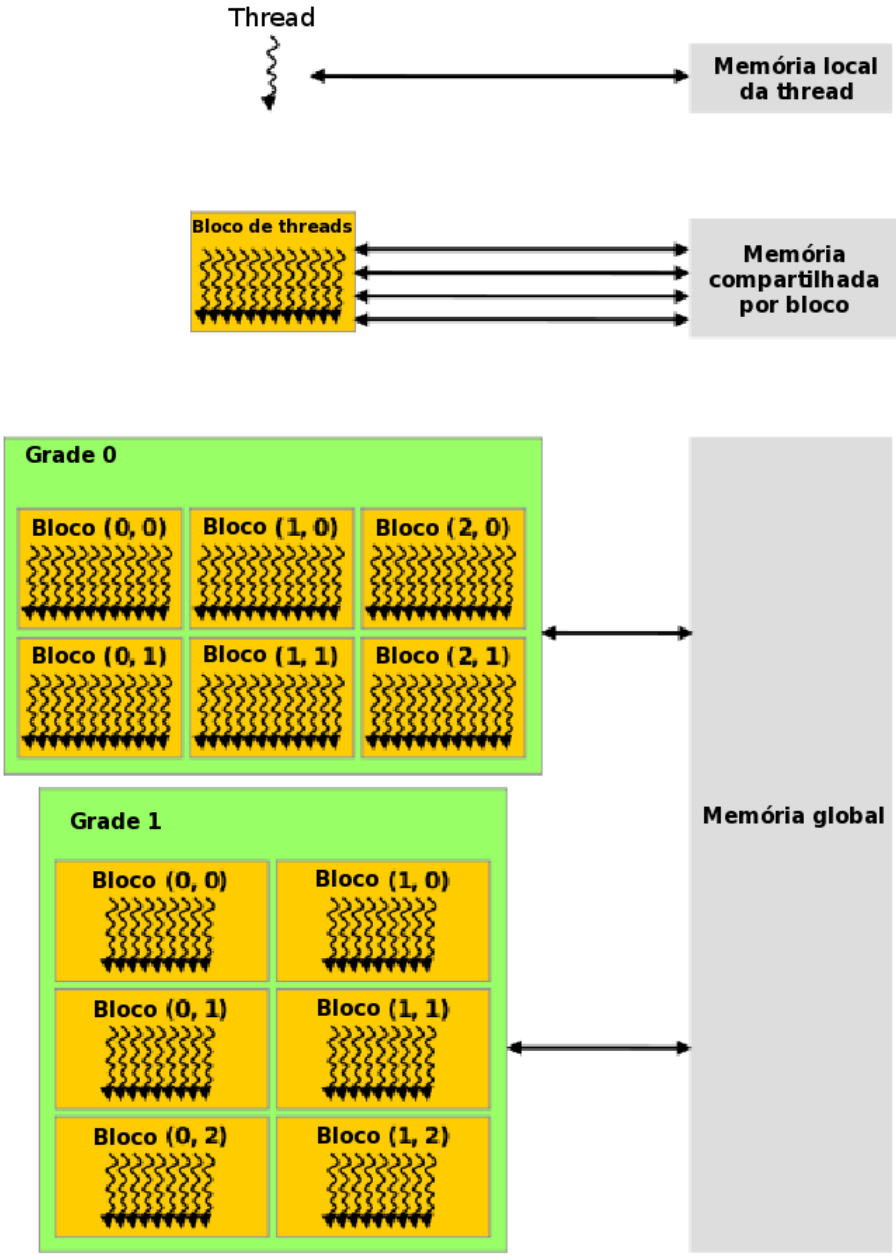

Figura 4.5: Hierarquia de memória (NVIDIA, 2009).

é descrita uma implementação usando o modelo de programação *CUDA* a fim de calcular todas as interações de forma rápida e eficiente.

Pode-se pensar na estrutura de dados do algoritmo que calcula as interações tipo par como uma matriz $M_{N\times N}$. A força total atuante sobre um átomo i, $\mathbf{F}_i$ é, então, obtida pela soma de todas as entradas $\mathbf{f}_{ij}$ da linha i da matriz M. Contudo, essa abordagem requer memória da ordem de $O(N^2)$ e é limitada pela largura de banda disponível. Uma solução é serializar a computação de uma linha i da matriz, de forma a atingir o desempenho máximo da unidade gráfica, mantendo as unidades aritméticas ocupadas, e diminuir a largura de banda necessária

para essas computações (NYLAND; HARRIS; PRINS, 2006).

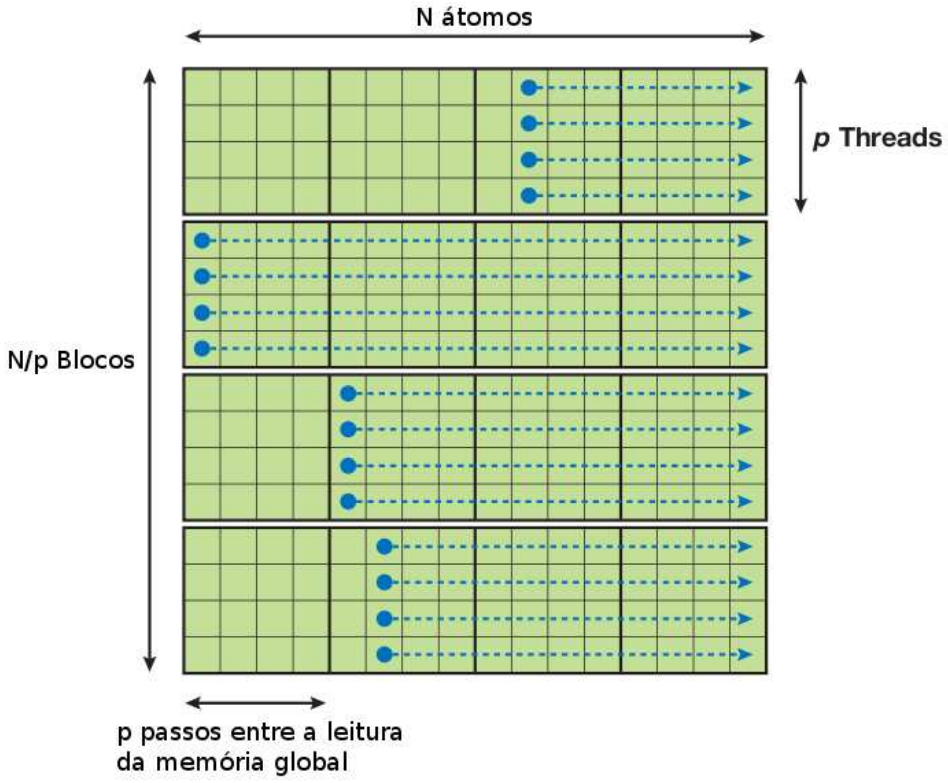

Figura 4.6: A grade de blocos de *threads* que calcula as N^2 interações. Aqui são definidos quatro blocos de quatro *threads* cada (NYLAND; HARRIS; PRINS, 2006).

4.3.1 Conceito de Ladrilho

Consequentemente é introduzido a noção de um *ladrilho* (em inglês *tile* computacional), uma região da matriz M consistindo de p linhas e p colunas. Somente $2p$ descrições são necessárias para avaliar todas as p^2 interações dentro de um *tile*. Essas descrições podem ser armazenadas na memória compartilhada, aumentando ainda mais o desempenho das operações. O efeito total das interações dentro de um *tile* de p átomos é visto como uma atualização a p vetores força. A Figura 4.7 descreve o esquema de um *tile*.

4.3.2 Definindo uma Grade de Blocos de *threads*

As interações são calculadas por um *kernel* onde cada *thread* é responsável pela avaliação de uma das linhas da matriz M. Esse *kernel* é chamado como uma grade de blocos de *threads* para calcular as interações dos N átomos do sistema. A divisão de *threads* é feita baseada na

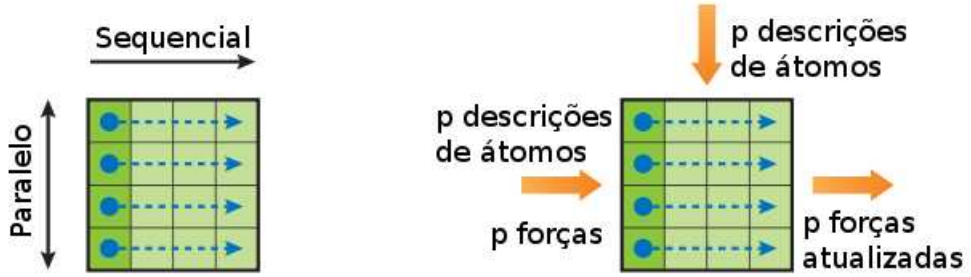

Figura 4.7: Esquema de um *tile* computacional. Linhas são calculadas em paralelo, mas cada linha é avaliada sequencialmente por uma *thread CUDA* (NYLAND; HARRIS; PRINS, 2006).

escolha do tamanho do *tile*. A grade do *kernel* deve possuir *p threads* por bloco e uma *thread* por átomo. Assim são necessários N/p blocos, assim define-se uma grade unidimensional de tamanho N/p.

4.3.3 Detalhamento da Implementação

Esta seção tem como objetivo descrever os detalhes da implementação em *CUDA* com detalhes de codificação que podem passar despercebidos. Aqui é introduzido o modificador de função **__device__**. Esse modificador aplicado em uma função especifica que esta é compilada para execução no dispositivo *CUDA* disponível.

A primeira função a ser definida é aquela que calcula a interação entre um par de átomos *i* e *j* qualquer. A função ***__device__ float3*** *lennard_jones_interaction(* ***float3*** *bi,* ***float3*** *bj)* recebe as posições dos átomos *i* e *j*, bi e bj respectivamente e retorna a força que *j* exerce em *i* de acordo com a Equação 2.6. Esta função será usada por cada uma das *threads* no momento de calcular a interação entre dois átomos

A segunda função é a responsável por calcular todas as interações dentro de um *tile* computacional. A função ***__device__ float3*** *lennard_jones_potential(* ***float3*** *pos,* ***float3*** *acc)* recebe com parâmetros a posição de um corpo qualquer e sua aceleração calculada até o momento. Então, é feita uma varredura pela memória compartilhada (o que está disponível para o *tile*) e a aceleração passada como parâmetro é acumulada.

```
__device__ float3 lennard_jones_potencial( float3 pos, float3 acc ) {
    extern __shared__ float3 sh_pos[];

    // Varre o tile calculando as interações
```

```
    for( int i = 0; i < blockDim.x; i++ ) {
        float3 acc_res = lennard_jones_interaction( sh_pos[i], pos );

        acc.x += acc_res.x;
        acc.y += acc_res.y;
        acc.z += acc_res.z;
    }

    return acc;
}
```

A útima função necessária para o cáculo das interações é a função que faz o gerenciamento dos *tiles*. Como mostrado na Figura 4.6, as *threads* de um mesmo bloco sincronizam ao final do calculo das interações e novas posições são carregadas para a memória compartilhada do bloco. A função que faz esse gerenciamento é ***__device__ float3*** *compute_forces(* ***float3*** *pos,* ***float3*** **positions,* ***int*** *n).* O primeiro parâmetro dessa função é a posição de um átomo que uma *thread* esta por conta. O segundo é o vetor de coordenadas que será usado para preenchimento da memória compartilhada dos blocos de *threads* a fim de calcular as interações de um *tile*. O último parâmetro é o tamanho do vetor de coordenadas. A função *compute_forces()* é definida a seguir:

```
/*
 * Macro usada para forçar que os tiles trabalhem
 * em uma porção diferente do vetor de posições.
 * Isso evita que diferentes threads façam leituras
 * na mesma posição do vetor positions e aumenta a
 * eficiência.
 */
#define WRAP(x,m) (((x)<(m))?(x):((x)-(m)))
__device__ float3 compute_forces( float3 pos, float3 *positions, int n ) {
    extern __shared__ float3 sh_pos[];

    float3 acc = {0.0f, 0.0f, 0.0f};
    int p = blockDim.x;
    int tiles = n / p;
```

```
    for( int tile = 0; tile < tiles; tile++ ) {
        sh_pos[ threadIdx.x ] =
            positions[
                WRAP( blockIdx.x + tile, gridDim.x ) * p + threadIdx.x
            ];

        __syncthreads();
        acc = lennard_jones_potencial( pos, acc );
        __syncthreads();
    }

    return acc;
}
```

O objetivo da função *compute_forces()* é gerenciar os *tiles* de maneira que as *threads* possam calcular as interações. A macro *WRAP* é usada para forçar que cada *tile* trabalhe em uma porção diferente dos dados de entrada. Isto evita que as *threads* façam leituras de uma mesma posição de memória. Assim, as leituras são todas realizadas em uma única vez, o que aumenta o desempenho dos cálculos. Após a leitura das posições para a memória compartilhada todoas as *threads* sincronizam e finalmente o cálculo das posições pode começar.

A última função a ser definida é aquela que executa o método de integração escolhido. Para este trabalho o método de integração *Leap-frog* foi escolhido. Esta função é definida como o *kernel* que calcula as interações de todos os átomos.

```
__global__ void kernel_integrate_system(
    float3 *new_pos, float3 *new_vel,
    float3 *old_pos, float3 *old_vel,
    float dt,
    int n
)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    float3 pos = old_pos[i];
    float3 acc = compute_forces( pos, old_pos, n );
    float3 vel = old_vel[i];
```

```
    // Aplica o método de integração
    ...
}
```

Os parâmetros *new_pos* e *new_vel* são ponteiros para escrita das novas posições e velocidades, enquanto que *old_pos* e *old_vel* são os vetores que contém as posições e velocidades para leitura. Essa divisão é feita para evitar escritas em posições ainda não lidas no vetor original. Por exemplo, se uma *thread* se um bloco qualquer termina a computação de um *tile* primeiro, e ele escreve as novas posições e velocidades e no mesmo vetor onde esta fez a leitura, pode acontecer de uma outra *thread* ler o novo valor quando deveria ler o valor antigo. Isso pode levar a erros ao longo da simulação. O paramêtro *dt* é o intervalo de tempo usado no método de integração (Sec. 2.5.1). O parâmetro *n* é o tamanho dos vetores de posições e velocidades.

4.4 Redutor Paralelo em *CUDA*

Esta seção tem como objetivo descrever a implementação de um redutor paralelo usando *CUDA*.

A operação de redução consiste em reduzir um vetor para um número de elementos menor do que seu tamanho original. A operação de soma de todos os elementos de um vetor é um exemplo de redução que reduz o número de elementos para apenas um elemento.

4.4.1 Redutor Paralelo

O primeiro conceito a ser definido é o de redutor paralelo. Enquanto um redutor sequencial é implementado de maneira trivial, ao se trabalhar com vários fluxos de execução deve-se tomar alguns cuidados. Uma abordagem comum para implementação da redução paralela é uma abordagem em árvore, como mostrado na Figura 4.8. Nesta abordagem é feita a redução de partes do vetor em paralelo. No exemplo da Figura 4.8 os valores 4, 7, 5 e 9 da segunda linha são calculados ao mesmo tempo na primeira iteração. Na segunda iteração, os valores 11 e 14 são calculados e finalmente, o valor 25 é tomado como resultado da redução.

Esta é uma primitiva importante para obtenção de propriedades, como a energia cinética e energia potencial (Eq. 2.11 e 2.10). Em suma, quando é necessário aplicar algum operador sobre os elementos de um vetor, um redutor pode ser usado.

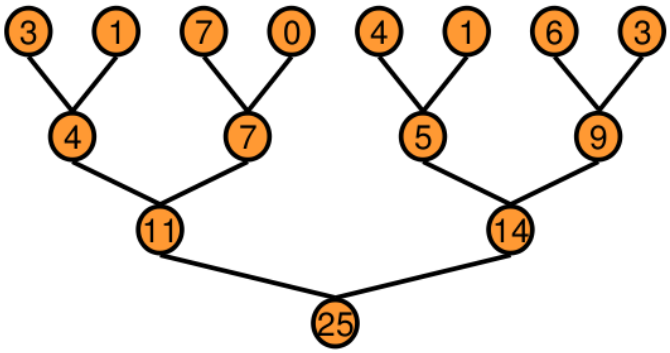

Figura 4.8: Abordagem em árvore para implementação de um redutor paralelo. (HARRIS, 2006).

Um redutor é relativamente fácil de ser implementado em *CUDA* devido a sua natureza. A dificuldade surge ao tentar extrair o máximo de desempenho da *GPU* . Ao mesmo tempo, a implementação de um redutor serve como um excelente exemplo de como um código *CUDA* pode ser otimizado[1].

4.4.2 Implementação em *CUDA*

A abordagem de árvore é usada dentro de cada bloco, ou seja, cada bloco delegara às suas *threads* a tarefa de aplicar o operador desejado sobre os elementos de um vetor de forma paralela. Como um bloco de *threads* possui um tamanho limitado, é requisito do problema que seja possível usar vários blocos de *threads* para processar vetores muito grandes e manter todos os processadores da *GPU* ocupados.

O primeiro problema surge da necessidade de comunicação dos resultados parciais entre blocos. Como visto na Seção 4.2.1, não existe um sincronizador global, somente uma barreira interna de cada bloco. Uma vez que existisse um sincronizador global, as *threads* de todos os blocos seriam obrigadas a esperar os resultados parciais serem produzidos e então a computação continuaria recursivamente. Mas o *CUDA* não possui um sincronizador global de *threads*. Tal primitiva é inviável para implementação em *hardware* com a quantidade de processadores das *GPUs* atuais. Além do mais, forçaria os desenvolvedores a trabalhar com um número de blocos reduzido, pois não haveria um alto número de processadores. A solução surge com a decomposição do problema em múltiplas chamadas ao mesmo *kernel*. Tal chamada atua como um ponto de sincronização global e, de acordo com (NVIDIA, 2009), chamadas a *kernels* possui um custo de *hardware* desprezível e um custo de software muito baixo.

[1]Detalhes sobre otimizações que podem ser realizadas são encontradas em (HARRIS, 2006)

A Figura 4.9 mostra o ponto de sincronização entre as reduções. Em um primeiro momento, chamado de Nível 0 é feita a redução do vetor usando-se 8 blocos de *threads*. Cada bloco faz a redução de uma parte do vetor produz um único elemento. Os elementos produzidos pelos diferentes blocos são passados para o mesmo *kernel*, lançado com somente um bloco para produzir o elemento final.

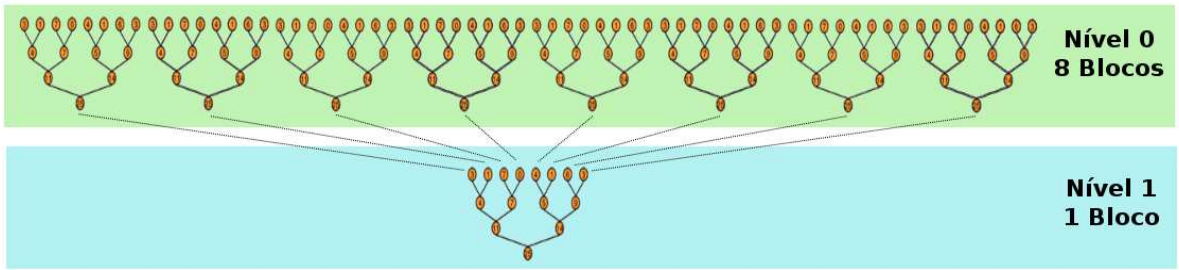

Figura 4.9: Problema da redução resolvido com múltiplas chamadas de *kernels* (HARRIS, 2006).

O esquema da redução paralela em *CUDA* é mostrado na Figura 4.10. Os elementos do vetor são carregados para a memória compartilhada do bloco. Cada *thread* faz a redução de dois elementos e armazena na própria memória compartilhada de forma a ser utilizado no próximo passo até que o bloco produza somente um elemento. Ao final da redução parcial, o valor gerado é carregado para a memória global para que possa ser utilizado para uma possível chamada à um *kernel* subsequente para completar a redução.

Pode-se notar que o redutor, na maior parte dos casos, é usado para reduzir vetores pela operação de soma. Porém, no estudo de DM é desejável uma primitiva que seja capaz de reduzir um arranjo de vetores tridimensionais. Por exemplo, quando se deseja calcular as velocidade total (Eq. 3.12) ou o centro de massa do sistema (Eq. 3.13). Para tal, uma pequena modificação no redutor proposto é feita de modo que cada *thread* faça a redução de dois vetores, como mostrado na Figura 4.11.

Cada redução tem um custo de 3 Flops, pois cada redução feita pelas *thread* opera sobre 3 valores.

4.4.3 Detalhamento da Implementação

Como dito na Seção 4.4.1, a implementação de um redutor em *CUDA* é simples, mas é de difícel extração de desempenho. O código a seguir mostra a implementação do redutor da maneira mais simples possível:

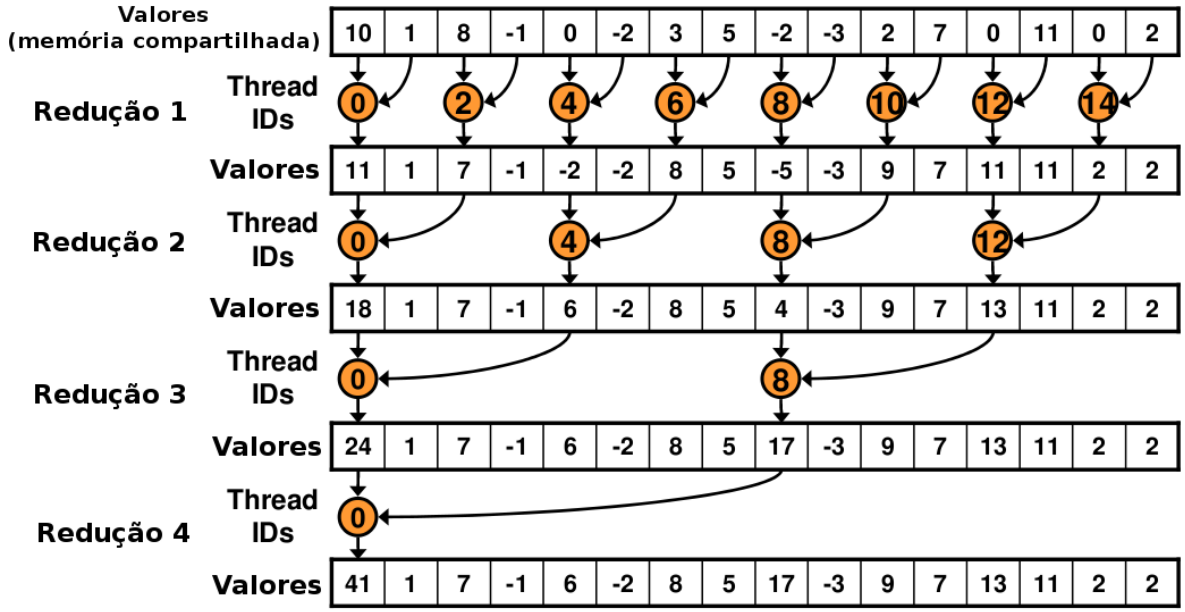

Figura 4.10: Esquema de um bloco de *threads* fazendo a redução de um vetor (HARRIS, 2006).

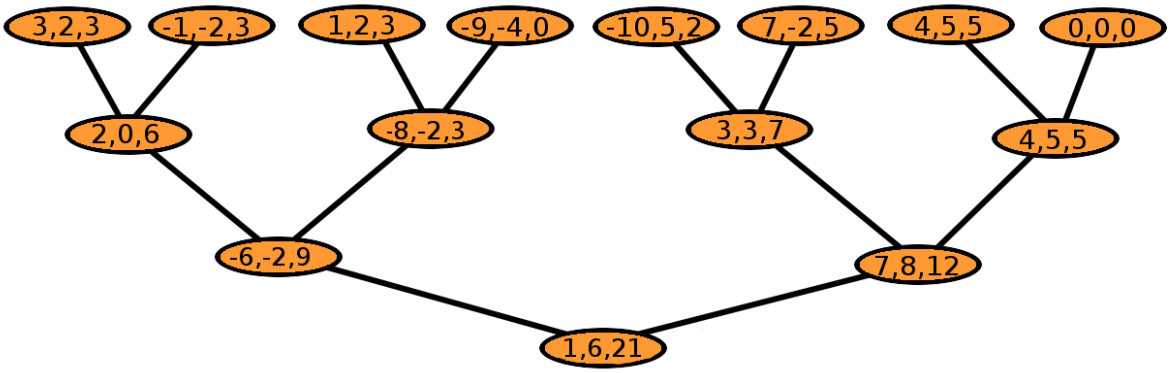

Figura 4.11: Redutor paralelo para redução de vetores tridimensionais.

```
__global__ void kernel_reduce( float *pD_input, float *pD_output, unsigned int n )
    extern __shared__ float sdata[];

    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

    // Carrega dados para a memória compartilhada
    sdata[tid] = ( i < n ) ? pD_input[i] : 0.0f;

    __syncthreads();
```

```
    //Faz a redução na memória compartilhada
    for( unsigned int s = 1; s < blockDim.x; s *= 2 ) {
        if( (tid % (2*s)) == 0 ) {
            sdata[tid] = sdata[tid + s];
        }
        __syncthreads();
    }

    // Escreve o resultado deste bloco na memória global
    if( tid == 0 ) pD_ouput[blockIdx.x] = sdata[0];
}
```

Primeiramente, os valores são carregados para a memória compartilhada e então as *threads* sincronizam para garantir que todas terminaram a tarefa antes de começar a redução. Cada *thread*, então, faz a redução de quantos elementos forem necessários, como mostrado na Figura 4.9. O resultado final da redução de cada bloco estará localizado na primeira posição da memória compartilhada. Esse valor é então escrito na memória global para que possa ser lido pelo hospedeiro.

5 *Resultados*

Este capítulo tem por objetivo mostrar os resultados obtidos com a implementação do simulador, tanto sequencial quanto paralelo usando *CUDA*. A Seção 5.1 visa mostrar a interface visual do simulador que permite interação com o usuário de maneira fácil e intuitiva. A Seção 5.3 se presta a exibição dos resultados de desempenho, como tempo de execução de ambas as versões e *Speed-up*.

5.1 Interface

A interface desenvolvida conta com duas telas. A primeira tela Figura 5.1 consiste na interface de controle do simulador. O usuário pode configurar os paramêtros de simulação através do painel *Simulation Parameters*. Estão disponíveis configurações de tamanho da região de simulação, configurações dos passos de medição, intervalo de relaxação, número máximo de iterações, intervalo entre as construções das listas de vizinhos, e intervalo de ajuste da temperatura para manter o sistema na temperatura desejada. Estão incluídos também opções de configuração de temperatura, que são as temperaturas máxima e mínima assim como a variação de temperatura ao final de cada ciclo de simulação. Densidade e o intervalo de tempo também podem ser ajustados. Foi incluído uma opção que permite ao usuário escolher entre o usar e não usar a lista de vizinhos para cálculo das interações.

A segunda tela Figura 5.2 exibe a simulação em tempo real. É possível visualizar as propriedades extraídas do sistema assim visualizar a posição e velocidade dos átomos. A posição de cada átomo é representado por uma esfera vermelha, e a velocidade é exibida pelo vetor em azul. O tamanho do vetor velocidade corresponde ao seu módulo. Isto permite acompanhar a evolução do sistema como um todo e identificar possíveis erros de implementação. A esfera verde no centro da região representa o centro de massa do sistema e pode ser usado para verificar algum deslocamento de massa indesejado.

Como resultado visual de desempenho é possível perceber a capacidade de processamento

que as *GPUs* modernas podem atingir. A Figura 5.2 mostra o visualizador para $N = 1000$ átomos. A Figura 5.3 mostra o visualizador *CUDA* para $N = 65536$ átomos. O visualizador para o modelo sequencial para esse número de átomos se torna inviável ao decorrer da simulação. O número de interações necessárias para evoluir o sistema é demasiadamente grande para permitir qualquer tipo de interação com o mesmo, ao contrário do visualizador *CUDA*.

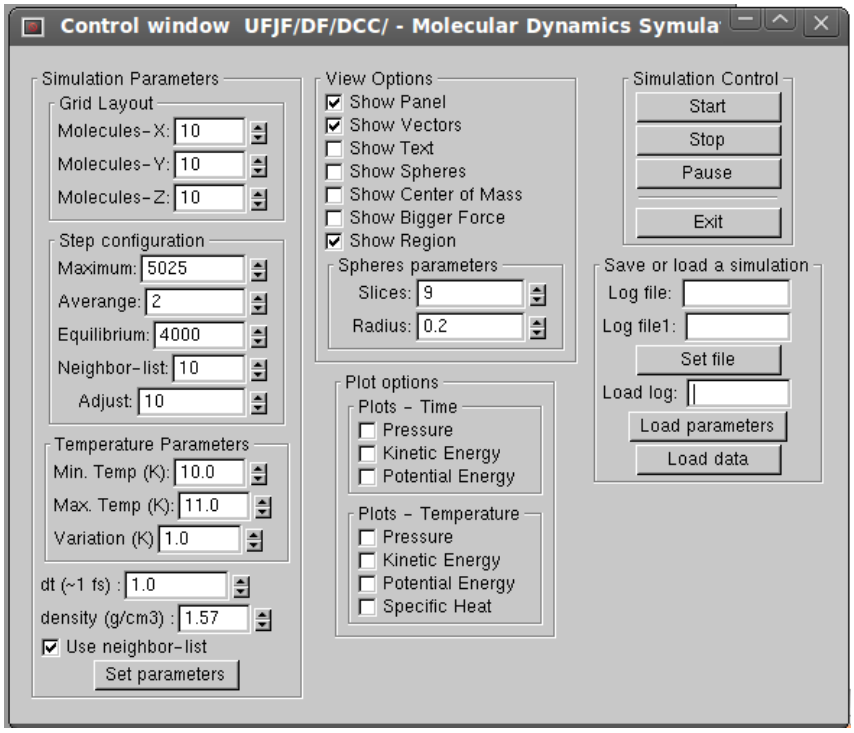

Figura 5.1: Interface de controle do simulador.

5.2 Medidas de Propriedades

Esta seção tem como objetivo mostrar alguma medidas de propriedades extraídas do simulador, mais especificamente, energias cinética e potencial médias e pressão média.

Foi realizada uma simulação com a seguinte configuração:

- Número de átomos: 512;
- Número máximo de iterações: 10000;
- Lista de vizinhos reavaliada a cada 10 iterações;
- Intervalo de medição: 100;

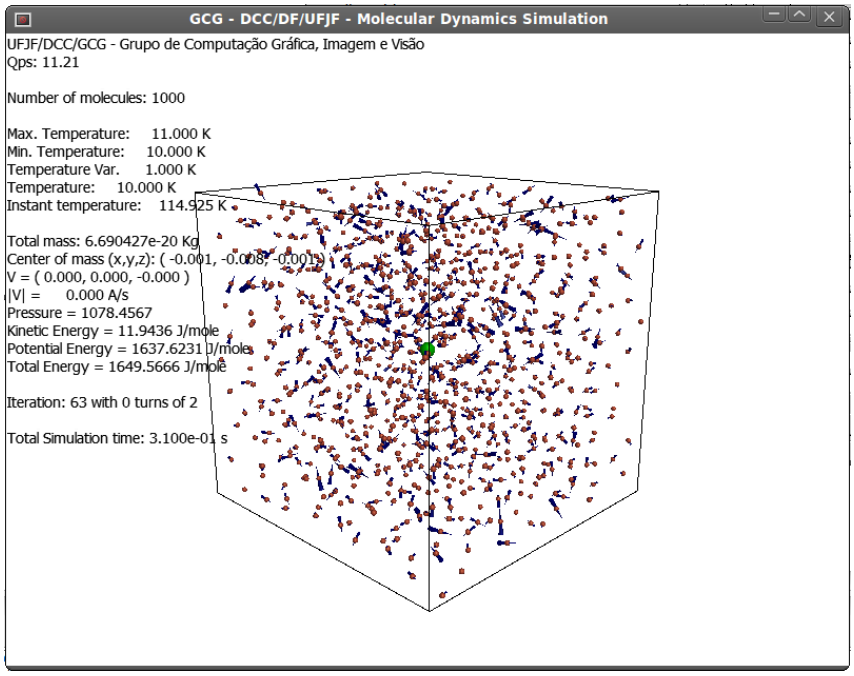

Figura 5.2: Interface de exibição em tempo real da simulação. $N = 1000$ átomos.

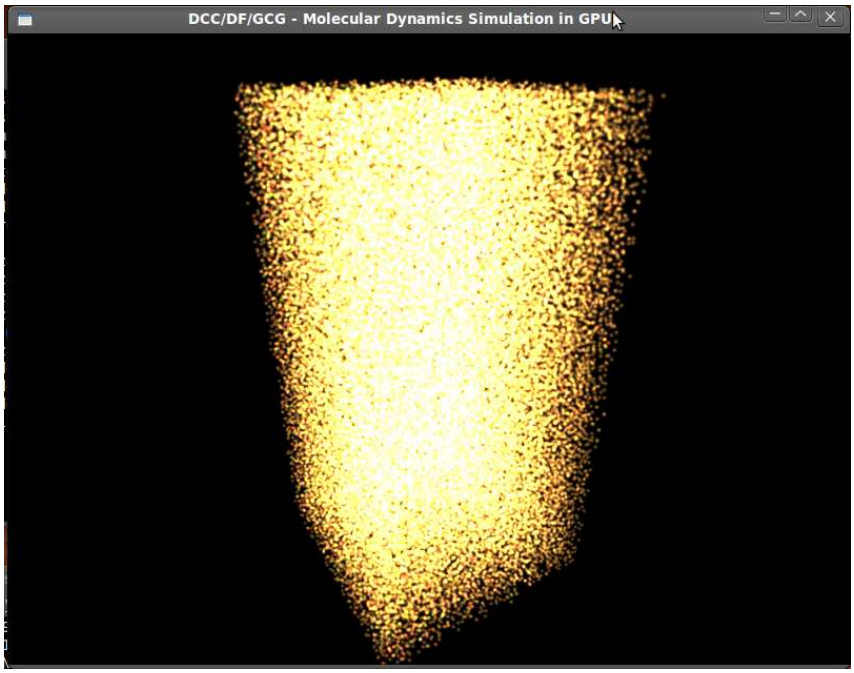

Figura 5.3: Interface de exibição em tempo real da simulação com processamento em *GPU* . $N = 65536$ átomos.

- Temperatura variando de 20K a 360K;
- Variação da temperatura: 1K;

- Densidade: 1 g/cm^3;
- Intervalo de tempo (Δt): 1 fs.

O gráfico da Figura 5.4 mostra a energia cinética média ao longo da simulação. Como era de se esperar, a energia cinética cresce linearmente a medida que a simulação avança no tempo, pois a simulação é configurada para uma rampa de aquecimento. Quanto mais alta a temperatura, maior é a velocidade individual dos átomos do sistema, e como a energia cinética é medida a partir dessas velocidades, maior o seu valor.

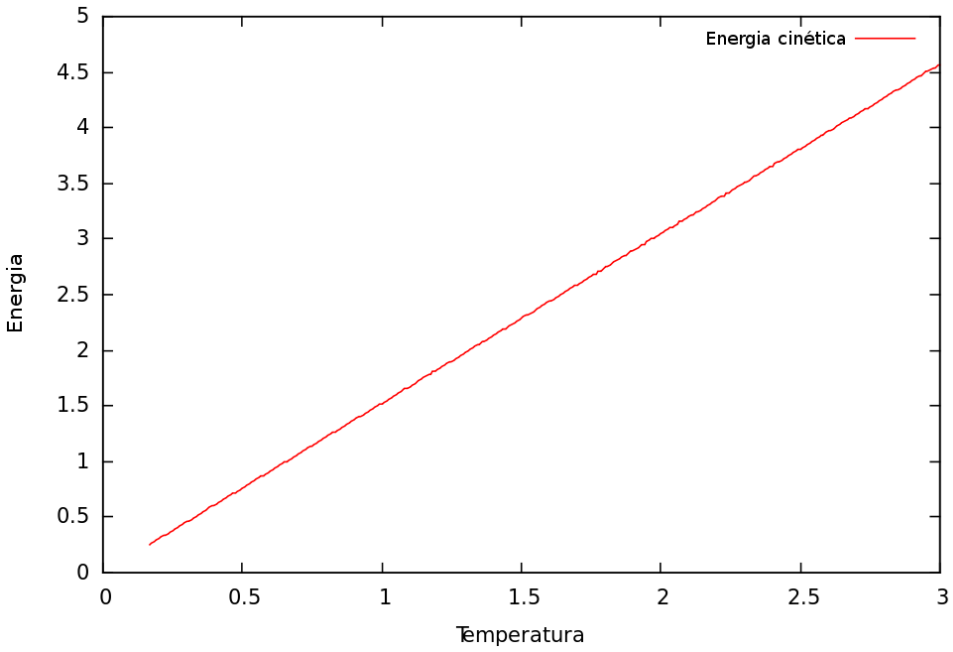

Figura 5.4: Aumento da energia cinética com o aumento da temperatura.

O gráfico da Figura 5.5 mostra a energia potencial ao longo da simulação. Existe uma pequena pertubação para temperaturas mais baixas. Isso é explicado pois a energia potencial adimensional depende exclusivamente da distância entre os átomos. Como o sistema é forçado a ficar na temperatura estabelecida, a baixas temperaturas a distância entre os átomos não é regular, o que leva a essa variação na energia potencial.

O gráfico da Figura 5.6 mostra a variação da pressão ao longo da simulação. A pertubação que aparece a baixas temperaturas é explicada pelo termo do Virial (Eq. 2.18). O termo do Virial é calculado em função da distância entre os átomos.

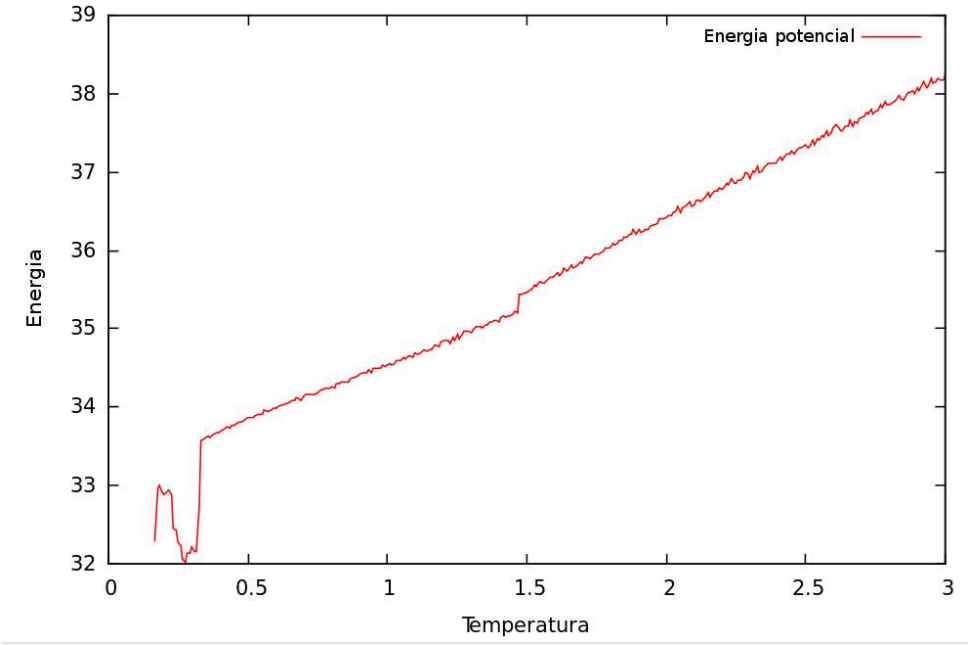

Figura 5.5: Aumento da energia potencial com o aumento da temperatura.

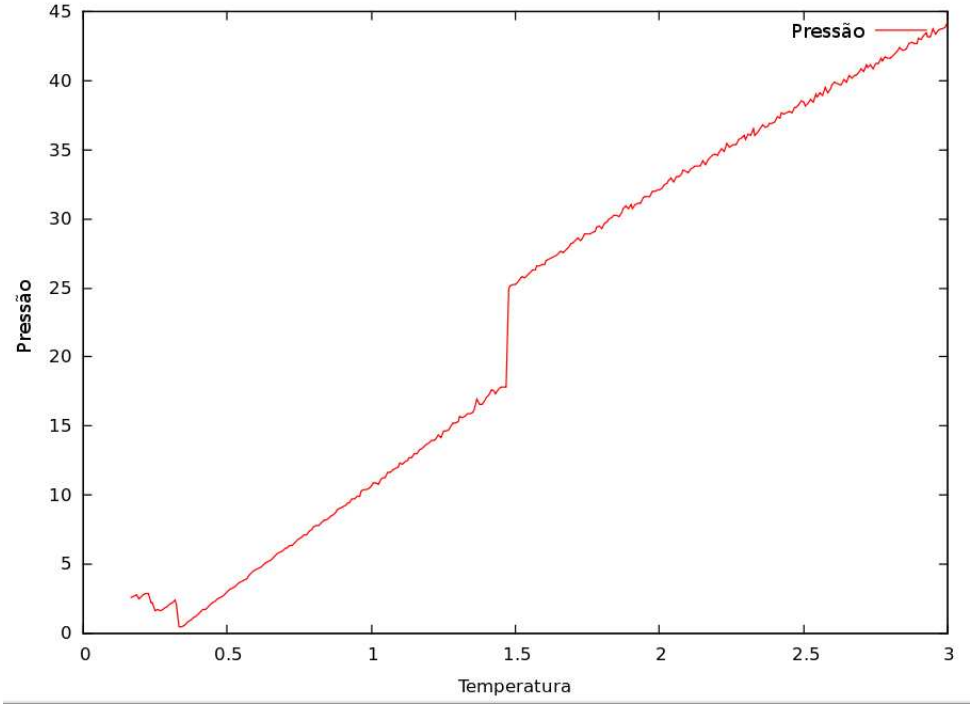

Figura 5.6: Aumento da pressão com o aumento da temperatura.

5.3 Testes de desempenho

Os testes de desempenho foram realizados em dois computadores: O primeiro possui processador Intel Core 2 Quad Q8300 de 2.5GHz, 4GB de memória RAM e placa de vídeo NVIDIA

GeForce 9800GTX, o segundo possui processador Intel Core 2 Quad Q9550 de 2.83GHz, 4GB de memória RAM e placa de vídeo NVIDIA GeForce GTX295.

As especificações das unidades gráficas usadas no Computador 1 são as seguintes:

- **GPU**: NVIDIA GeForce 9800GTX;
- **Frequência do Relógio**: 1.67GHz;
- **Número de Processadores**: 128;
- **Número de Multiprocessadores**: 16;
- **Memória**: 512 MB;
- **Interface de memória**: 256 bits.

Para o Computador 2 as especificações são:

- **GPU**: NVIDIA GeForce GTX295[1];
- **Frequência do Relógio**: 1.24GHz;
- **Número de Processadores**: 240;
- **Número de Multiprocessadores**: 30;
- **Memória**: 896MB;
- **Interface de memória**: 448 bits.

Todas as simulações foram executadas como especificado abaixo: [2]

- Número de átomos: 512, 1024, 2048, 4096, 8192, 16384, 32678 e 65536; [3]
- Número máximo de iterações: 10000;
- Intervalo de relaxação: 4000;
- Intervalo de medição: 100;

[1] A unidade gráfica é formada por duas *GPUs* , cada um com a configuração listada.

[2] Cada simulação foi executada 5 vezes e o resultado mostrado apresenta a média entre os valores.

[3] A variação foi escolhida dessa forma pois a implementação *CUDA* atualmente funciona somente com número de átomos em potência de 2

- Em caso de uso da lista de vizinhos, o intervalo para sua reavaliação da lista é 10 iterações;
- Temperatura fixa em $100K$
- Densidade: $0.4g/cm^3$
- Intervalo de tempo (Δt): $1fs$.

5.3.1 Tempo de Execução

Os gráficos a seguir (Fig. 5.7 e 5.8) apresentam os resultados de tempo de execução comparando a implementação com a utilização ou não da lista de vizinhos e comparando também com a implementação em *CUDA*.

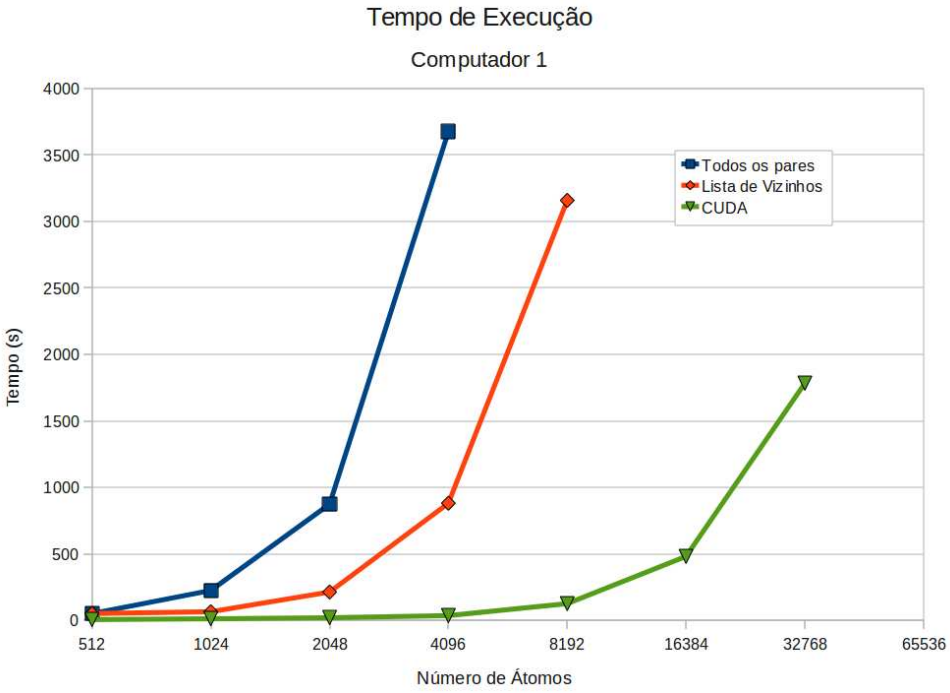

Figura 5.7: Tempo de simulação usando o Computador 1.

É claro que as implementações sequenciais possuem um comportamento $O(N^2)$, mas a implementação que faz uso da lista de vizinhos precisa de menos tempo para calcular todas as interações e por isso suporta um número de átomos maior. A implementação *CUDA* não esconde o comportamento quadrático, pois a avaliação é feita para todos os pares. Porém, o paralelismo proporcionado faz com que seja possível executar simulações com um número grande de átomos. O uso de uma lista de vizinhos na implementação em *CUDA* teria como objetivo reduzir o comportamento da curva de tempo para $O(N)$.

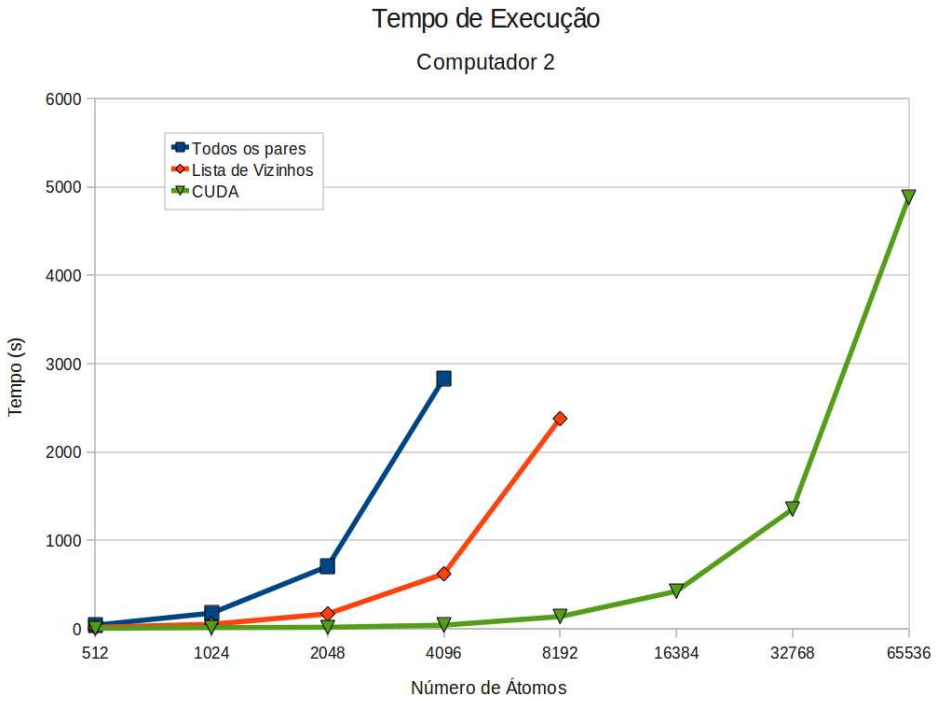

Figura 5.8: Tempo de simulação usando o Computador 2.

5.3.2 Iterações por Segundo

Os gráficos a seguir (Fig. 5.9 e 5.10) mostra quantas iterações por segundo o simulador é capaz de calcular. Quando maior esse número, mais eficientemente é feito o cálculo das interações.

Novamente fica evidente que o uso de uma lista de vizinhos faz com que o número de iterações por segundo aumente, assim como o paralelismo do código *CUDA* faz com que o número de iterações por segundo seja elevado quando $N < 8192$.

5.3.3 Razão Entre os Tempos de Execução

Os gráficos a seguir (Fig. 5.11 e 5.12) apresentam a razão A_N entre os tempo de execução do código *CUDA* em relação com as outras duas implementações. Esta razão é calculada como mostrado a seguir:

$$A_N = \frac{T_{C_N}}{T_{S_N}} \tag{5.1}$$

onde T_{C_N} é o tempo total de execução do código *CUDA* para N átomos e T_{S_N} é o tempo total de execução do código sequêncial, também para N átomos.

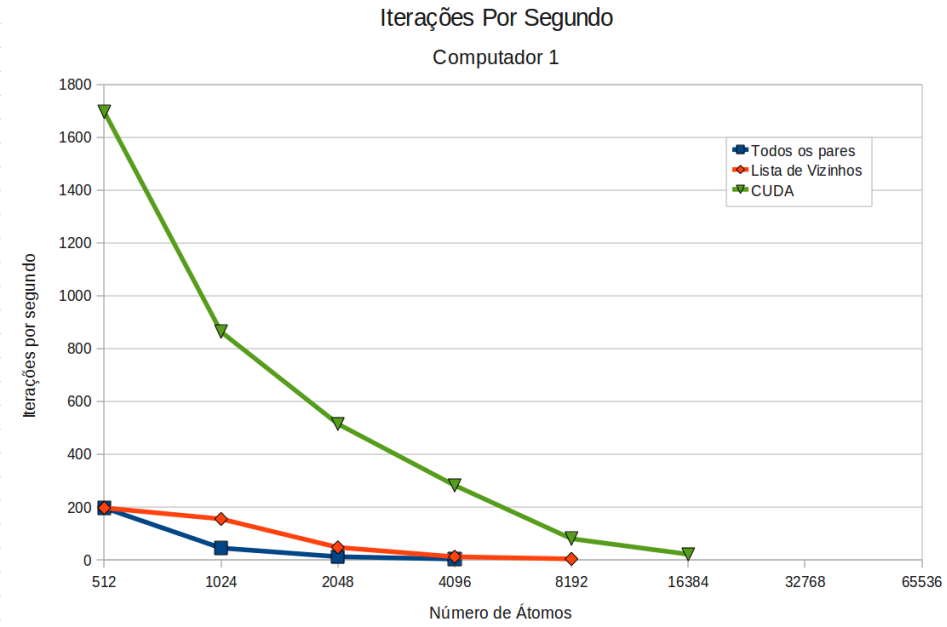

Figura 5.9: Iterações por segundo usando o Computador 1.

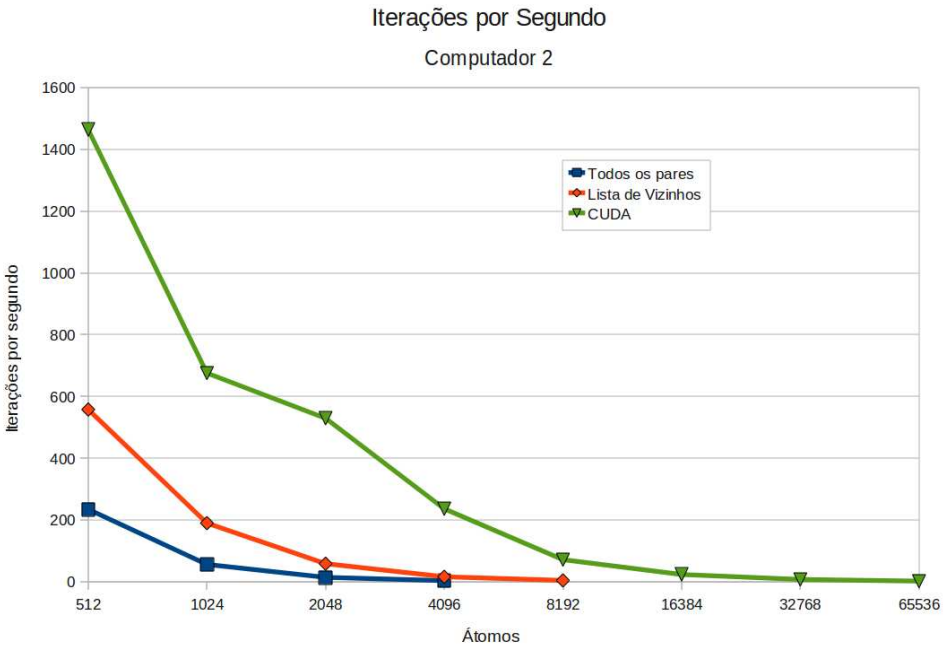

Figura 5.10: Iterações por segundo usando o Computador 2.

A comparação da execução *CUDA* com a execução sequencial que calcula a interação de todos para todos possui um ganho nítido. Como era de se esperar, quanto maior o número de átomos, maior é o ganho em relação à implementação sequencial. Na comparação com o código que faz uso da técnica da lista de vizinhos o ganho não é tão expressivo pois a lista de vizinhos reduz o número de cálculos para $O(N)$ enquanto a execução *CUDA* ainda possui custo $O(N^2)$.

O ganho da execução do código *CUDA* em relação ao código que faz uso da técnica da lista de vizinhos acontece pois existe a necessidade de reavaliar a lista de vizinhos, e esta avaliação ainda é $O(N^2)$.

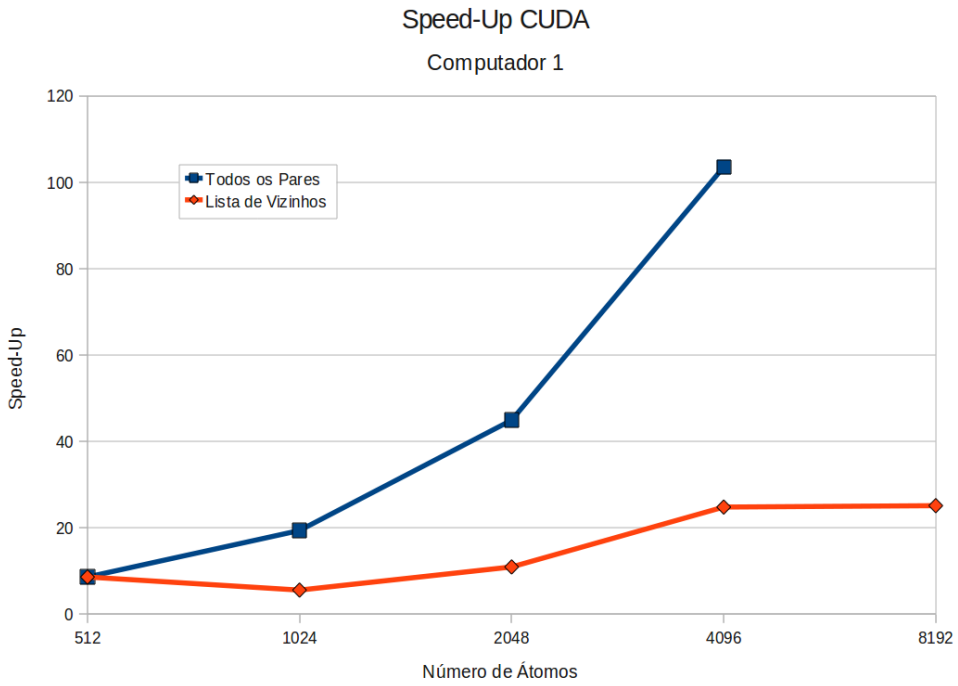

Figura 5.11: Razão entre tempos de execução da implementação *CUDA* e as demais usando o Computador 1.

5.4 Discussão dos Resultados

Como é possível perceber nos gráficos mostrados na Seção 5.3, o resultado para a implementação *CUDA*, mesmo tendo custo computacional de $O(N^2)$, supera por uma diferença significativa as implementações sequenciais[4].

Nos gráficos de tempo de execução (Sec. 5.3.1) é interessante notar que o tempo de execução do Computador 1 supera o tempo de execução do Computador 2 para $N > 4096$. Isto acontece pois relógio da *GPU* do Computador 1 possui uma frequência de operação um pouco maior do que a unidade gráfica da *GPU* do Computador 2. Isto quer dizer que, para o mesmo número de *threads* a *GPU* Computador 1 é mais rápida. Contudo, o alto número de processadores da *GPU* do Computador 2 é visivelmente percebido para $N > 8192$. Isto é justificado pelo fato de que quanto mais alto o número de processadores, mais *threads* em paralelo podem ser executadas sem necessidade de um escalonamento.

[4]Valores da variância ficaram todos abaixo de 0.02 %, portanto não foram colocados nos gráficos.

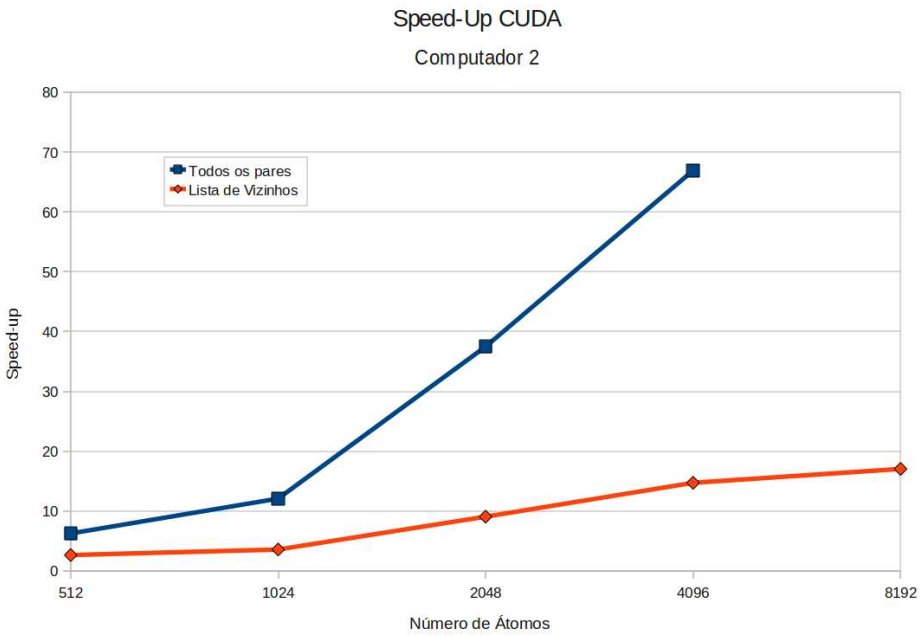

Figura 5.12: Razão entre tempos de execução da implementação *CUDA* e as demais usando o Computador 2.

6 Conclusão

Neste trabalho foi estudado o problema da DM do ponto de vista físico e foram descritos os detalhes necessários para a implementação de um simulador de DM para sistemas compostos por átomos de argônio. Foram descritos os detalhes para uma implementação sequencial e uma implementação paralela utilizando a tecnologia *CUDA*.

O modelo *CUDA* implementado necessita de melhorias para suportar um números arbitrários de átomos, não somente potências de 2.

A implementação da técnica da lista de vizinhos melhorou bastante o tempo de execução para um número maior de átomos no modelo sequencial. Porém, está mesma técnica ainda precisa ser implementada em *CUDA* a fim de extrair o máximo do poder computacional proporcionado pelas unidades gráficas assim como a implementação descrita em (ANDERSON; LORENZ; TRAVESSET, 2008).

Os resultados de tempo de execução foram satisfatórios, mostrando que a implementação em *CUDA* é capaz de atingir um número de átomos grande e manter o número de iterações por segundo em um valor relativamente alto, mesmo calculado todas as N^2 interações.

Como trabalhos futuros podem ser implementados integradores mais complexos em *CUDA* de modo a tratar outros tipos de sistemas, com diferentes tipos de átomos. Outra proposta é a integração de analisadores de propriedades na aplicação desenvolvida de forma a evitar a necessidade de uso de outras ferramentas para essa tarefa.

Referências Bibliográficas

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>.

ALLEN, M. P.; TILDESLEY, D. J. *Computer Simulation of Liquids*. Oxford: Oxford University Press, 1987.

ANDERSON, J. A.; LORENZ, C. D.; TRAVESSET, A. General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of Computational Physics*, p. 17, 2008. ISSN 5342-5359.

BEEMAN, D. Some multistep methods for use in molecular dynamics calculations. *Journal of Computational Physics*, v. 20, p. 130+, February 1976.

BILLING, G. D.; MIKKELSEN, K. V. *Introduction to Molecular Dynamics and Chemical Kinetics*. 1. ed. New York, NY, USA: Wiley-Interscience, 1996. ISBN 0-471-12739-6.

CAMPOS, A. M. *Implementação Paralela de um Simulador de Spins em uma Unidade de Processamento Gráfico*. Dez 2008.

COHEN, I. B. *Revolution in Science*. Cambridge, MA: Havard University Press, 1985.

HAILE, J. M. *Molecular Dynamics Simulation - Elementary Methods*. 1. ed. Clemson, South California: A Wiley-Interscience Publication, 1992. ISBN 0-471-81966-2.

HANSEN, D. P.; EVANS, D. J. A generalized heat flow algorithm. *Molecular Physics: An International Journal at the Interface Between Chemistry and Physics*, Taylor & Francis, p. 767–779, 1994.

HARRIS, M. *Optimizing Parallel Reduction in CUDA*. 2006. Slides. Disponível em: <http://developer.download.nvidia.com/compute/cuda/11/Website/projects/reduction/doc/reduction.pdf>.

LAPLACE, P. S. *Philosophical Essay on Probabilities*. New York, NY, USA: Roslyn, 1914.

LENNARD-JONES, J. E. Cohesion. *Proceedings of the Physical Society*, A 106, p. 461–482, 1924.

NEWTON, I. S. *Principia Mathematica*. 1. ed. [S.l.]: Imprimatur, 1686. (Philosophiae Naturalis).

NVIDIA. *NVIDIA CUDA Programming Guide*. [S.l.], 2009.

NYLAND, L.; HARRIS, M.; PRINS, J. Fast n-body simulation with cuda. NVIDIA, p. 677–695, 2006. Disponível em: <http://developer.nvidia.com/gpugems3>.

RAPAPORT, D. C. *The Art of Molecular Dynamics Simulation*. New York, NY, USA: Cambridge University Press, 1996. ISBN 0521445612.

VERLET, L. Computer experiments on classical fluids. i. thermodynamical properties of lennard-jones molecules. *Phys. Rev.*, p. 159–257, 1967.