Centro Universitário de Brasília
Faculdade de Tecnologia e Ciências Sociais Aplicadas

Computação de Alto Desempenho Utilizando Unidade de Processamento Gráfico

Alison Barros da Silva

BRASÍLIA
2008

Alison Barros da Silva

Computação de Alto Desempenho Utilizando Unidade de Processamento Gráfico

Trabalho apresentado ao
Centro Universitário de
Brasília (UniCEUB/FATECS)
como pré-requisito para a
obtenção do grau de
Engenheiro Eletricista
com Ênfase em Computação.

Orientador: Flávio Antônio Klein
Co-orientador: Rafael Morgado Silva

BRASÍLIA
2008

Agradecimentos

Em primeiro lugar, agradeço aos meu pais Amaury Amaral da Silva e Clotilde Cruz Barros por toda a atenção e educação oferecida a mim , e pela paciência que tiveram em todos os momentos difíceis, não só durante a realização deste trabalho, mas durante todos os desafios que me propus. Quero que saibam que o apoio fornecido por vocês foi de fundamental importância e sem ele eu não teria conseguido. Amo vocês!

Agradeço meu orientador Me. Flávio Antônio Klein pela compreensão e paciência durante todo o semestre, e se mostrar sempre pronto a ajudar.

Agradeço meu co-orientador Dr. Rafael Morgado por ter acreditado no meu trabalho e ter investido nele, mesmo em horas em que eu achei que não fosse conseguir.

Agradeço também aos meus amigos da confraria: Dr. André Penna e Dr. Mendeli Vainstein, pelas discussões e incentivos, além de me ajudarem com as brincadeiras da simulação do modelo de Ising nas férias.

Um agradecimento especial ao Dr. Fernando Albuquerque de Oliveira, que me deu a oportunidade de trabalhar em um grupo de pesquisa de excelência e por se mostrar sempre disposto a ajudar com seus comentários que, apesar de confusos, funcionam muito bem.

Agradeço aos meus amigos Ulisses Sebastian Uziech, Fernando Barbosa Vito, Luiz Rafael Vasconscelos e Gustavo Garcia Rondina pelos momentos de distração e discussões sobre física e computação mesmo em momentos que estávamos todos alterados.

Quero agradecer também a minha namorada Fran pelo apoio, motivação e pela paciência durante toda a realização deste trabalho, sua presença foi muito importante. Te adoro!

Centro Universitário de Brasília
Faculdade de Tecnologia e Ciências Sociais Aplicadas
Bacharel em Engenharia da Computação

Computação de Alto Desempenho Utilizando Unidade de Processamento Gráfico

ALISON BARROS DA SILVA

MONOGRAFIA APROVADA PELO CORPO DOCENTE DO CENTRO UNIVERSITÁRIO DE BRASÍLIA UNICEUB COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE BACHAREL EM ENGENHARIA ELÉTRICA COM ÊNFASE EM COMPUTAÇÃO

Prof. Me. Flávio Antônio Klein

Prof. Dr. Rafael Morgado Silva

Prof. Dr. Luís Cláudio Lopes de Araújo

Prof. Me. Claudio Penedo de Albuquerque

Brasília
12/2008

Sumário

Lista de Figuras

Lista de Símbolos

Lista de Equações

Resumo

Nos últimos anos o hardware gráfico se desenvolveu além de suas funções fixas. Hoje existem placas gráficas completamente programáveis; novos ambientes de programação estão sendo desenvolvidos através de linguagens de programação de rápida curva de aprendizado, como a linguagem CUDA. Nas placas gráficas mais modernas são encontradas centenas de unidades de processamento (unidades lógico-aritméticas), tornando possível distribuir o poder de processamento de forma a resolver problemas complexos que seriam impossíveis de resolver utilizando um único núcleo de processamento, devido ao tempo de execução. O paralelismo oferecido pela arquitetura SIMD (Single Instruction Multiple Data) e a maior flexibilidade nas linguagens de programação deste tipo de hardware abrem novas possibilidades de aplicação, fato que vem chamando a atenção da comunidade científica. Neste trabalho, será mostrado uma simulação do movimento browniano correlacionado, implementado no hardware Geforce NVIDIA 9800 GTX com a plataforma de desenvolvimento CUDA SDK 2.0. Usando este hardware é possível resolver, em paralelo, um *ensemble* de partículas descrito pela Equação de Langevin Generalizada (ELG), que é uma equação integro-diferencial estocástica. Como a ELG é equivalente a equação do movimento de Heisenberg para um operador quântico, e também a dinâmica Hamiltoniana clássica, ela é útil para modelar fenômenos dinâmicos de interesse em muitas áreas do conhecimento como: difusão anômala, circuitos nanoeletrônicos, escoamento em meios desordenados, condução anômala etc. Os resultados mostram uma aceleração de 27 vezes para o código executando em GPU em comparação com um código otimizado executando em um processador Intel Q6600. Com esse resultado, é possível concluir que usar hardware gráfico para simulação de processos estocásticos é uma excelente opção, permitindo a solução de sistemas maiores, aumento de precisão dos resultados, abrindo novas possibilidades de pesquisa.

Palavras-chave: Unidade de processamento gráfico, Graphics Processing Unit, GPGPU, High Performance Computing, Compute Unified Device Architecture, CUDA, Movimento Browniano, Equação de Langevin Generalizada.

Abstract

In last years, graphics hardware evolved beyond the traditional fixed functions. Today there are graphics boards completely programmable; new programming frameworks are being developed, mainly through languages of quick learning curve, like C language. In a last generation graphics board, we find dozens or even hundreds of processing units (arithmetic logic units), making possible to distribute processing power to solve complex problems that otherwise would be impossible to solve in a one core system, due to prohibitive CPU time consumption. The parallelism offered by SIMD (Single Instruction Multiple Data) architecture and the bigger flexibility in programming this hardware opens many new possibilities of applications, fact that is drawing attention from scientific community. In this monography, we show a correlated Brownian movement simulation, implemented in a Nvidia 9800 GTX graphics board with CUDA SDK 2.0 development platform. By using this hardware, we are able to solve in parallel an ensemble of particles described by the Generalized Langevin Equation (GLE), which is a stochastic integro-differential equation. Since the GLE is equivalent to Heisemberg equation of motion for a quantum operator, and also to classical Hamiltonian dynamics , it is helpful in modelling almost any dynamical physics phenomenon in many research fields, like anomalous difusion, nanoelectronics, fluid flow in disordered media, anomalous conduction, among many others. Our results show an acceleration of 27 times for a code running in the GPU against an optimized serial code running in an Intel Q6600 CPU. From this, we conclude that using the GPU for simulation of stochastic processes is worth, allowing solutions of very large systems, improving the precision of the results and opening new possibilities of research.

1 Introdução

No mundo moderno, em diversas situações nos deparamos com uma grande quantidade de dados que precisam ser processados rapidamente. Principalmente em aplicações científicas precisamos de muito poder computacional, geralmente são executadas em supercomputadores ou clusters de computadores (CPCs), com tempos de execução da ordem de dias ou meses. Assim muito investimento é feito no sentido de obter equipamentos que diminuam o tempo de execução das aplicações, acelerando assim o processo de pesquisa.

Uma alternativa aos supercomputadores e CPCs é o uso de hardware auxiliar. Este hardware auxiliar pode ser de dois tipos: específico, como no caso de chips ASIC (Application Specifc Integrated Circuit) , ou de uso geral, como os FPGAs (Field Programable Gate Arrays) e placas gráficas de última geração. Dentre essas alternativas a mais econômica é o uso do hardware gráfico, pois são produzidos em larga escala devido à grande demanda pelo público de jogos eletrônicos.

Nos últimos anos, o hardware gráfico evoluiu além das funções fixas tradicionais. Hoje existem placas gráficas completamente programáveis. Em uma placa gráfica de última geração, encontramos dezenas ou mesmo centenas de unidades de processamento (núcleos). A Nvidia Geforce 9800GTX, por exemplo, possui 128 unidades de processamento. O paralelismo oferecido pela arquitetura multi-núcleos e a maior flexibilidade na programação deste hardware abrem diversas possibilidades em termos de aplicações, fato que vem chamando a atenção da comunidade científica. O modelo de programação Computed Unified Device Architecture (CUDA) [1], criado pela NVIDIA Corporation, permite desenvolver aplicações não gráficas que

usam o processador gráfico como um coprocessador matemático massivamente paralelo. Em certas aplicações, o ganho chega a duas ordens de magnitude em relação aos processadores mais modernos da Intel ou AMD [2].

Este trabalho tem como objetivo apresentar uma alternativa de baixo custo ao uso de supercomputadores e clusters de computadores em computação científica e mostrar que essa tecnologia diminui significativamente o tempo de processamento de programas massivamente paralelos.

No capítulo 2, será descrito o movimento browniano: primeiramente, será dado um breve histórico das pesquisas iniciais envolvendo este fenômeno, então será mostrada a formulação de Langevin para o movimento browniano, e em seguida essa formulação será generalizada para o movimento browniano correlacionado.

No capítulo 3, será apresentada uma breve motivação ao uso das placas gráficas, depois será feita uma descrição dos processadores de fluxo, assim como exemplificar sua arquitetura base. Após a apresentação dos conceitos básicos de processadores de fluxo será feita uma descrição detalhada da placa de vídeo Geforce 9800 GTX fabricado pela NVIDIA Corporation e montada pela XFX.

No capítulo 4, serão apresentados os conceitos básicos da linguagem de programação CUDA.

No capítulo 5, será mostrado todo o desenvolvimento do trabalho. Detalhes de implementação e métodos de otimização do algoritmo proposto serão apresentados, assim como os problemas enfrentados durante o desenvolvimento do trabalho e o tratamento utilizado na solução de cada um deles. Os programas serão executados em três tecnologias distintas: serial (CPU), paralela (CPU+MPI) e massivamente paralela (GPU). O tempo de processamento será comparado e será feita uma análise dos dados obtidos.

No último capítulo serão apresentadas as conclusões e perspectivas para trabalhos futuros. O apêndice dará uma descrição do algoritmo numérico utilizado para o cálculo da Equação de Langevin Generalizada.

2 O Movimento Browniano

O movimento irregular de partículas diminutas foi pela primeira vez relatado pelo físico holandês Jan Ingenhousz, em 1785. Entretanto, o fenômeno só recebeu notoriedade com o estudo do botânico Robert Brown, que em 1827, estudando grãos de pólen sobre a superfície da água acreditou ter encontrado a essência da matéria viva, porém logo percebeu que o fenômeno também ocorria em qualquer partícula fina, orgânica ou não [3].

Em 1888, o físico francês Louis-Georges Gouy atribuiu o movimento da partícula aos efeitos dos movimentos térmicos das moléculas do fluido e estabeleceu os seguintes pontos [4]

1. O movimento é extremamente irregular e a trajetória aparentemente não possui tangentes.
2. Aparentemente as partículas são completamente independentes.
3. Quanto menor as partículas, mais ativo é o movimento.
4. Quanto menos viscoso o fluido, mais ativo é o movimento.
5. Quanto maior a temperatura, mais ativo é o movimento.
6. A composição e densidade das partículas não têm nenhum efeito sobre o movimento.
7. O movimento nunca cessa.

Em um de seus famosos artigos de 1905, Einstein propõe um modelo para o movimento browniano:

> "Corpos de tamanho visível ao microscópio, e que estão em suspensão em um líquido, devem executar, como conseqüência dos movimentos térmicos moleculares, movimentos de tal magnitude que podem ser facilmente observáveis com a utilização de um microscópio. É possível que os movimentos a serem aqui discutidos sejam idênticos ao assim chamado movimento molecular browniano entretanto, os dados que tenho disponíveis sobre este último são tão imprecisos que eu não poderia formar uma opinião a respeito."[5]

Nesse mesmo artigo Einstein faz toda uma formulação matemática do movimento browniano utilizando a física estatística, mais especificamente, a teoria cinético-molecular do calor. Utilizando essa formulação ele deduz a relação entre difusão e viscosidade.

Einstein propõe nesse trabalho a seguinte relação entre o coeficiente de difusão e o deslocamento quadrático médio de uma partícula que difunde em um meio

$$D = lim_{t\to\infty}\frac{< x^2 >}{2t}, \tag{2.1}$$

onde D é a constante de difusão, expressa por

$$D = \frac{K_b T}{m\gamma} \tag{2.2}$$

Nessa equação K_b é a constante de Boltzmann, T a temperatura absoluta, m a massa da partícula e γ é a fricção.

2.1 A Equação de Langevin

Vamos considerar o movimento unidimensional de uma partícula livre (nenhuma força atua além daquela que é originada pelos choques moleculares) imersa em um fluido qualquer de viscosidade η. A segunda lei de Newton diz que a equação que descreve o movimento de tal partícula é

$$m\frac{dv(t)}{dt} = F(t)$$

Onde F(t) é a força atuante devido ao bombardeio das moléculas do fluido. Langevin sugeriu que a força F(t) poderia ser decomposta em duas componentes: uma força de fricção proporcional à velocidade, devida ao fluido, e uma força aleatória R(t) que tem origem no impacto com as moléculas do fluido. Então a equação se torna:

$$m\frac{dv(t)}{dt} = -m\gamma v(t) + R(t), \tag{2.3}$$

$$< R(t) >= 0$$

$$< R(0)R(t) >= g\delta(t)$$

onde g é uma constante dada por $2\gamma K_b T$.

Essa força representa os eventuais choques entre a partícula browniana e as moléculas do fluido que constituem o meio que envolve a partícula, sendo comum chamar essa força de ruído de fundo. Esse ruído pode ser: delta-correlacionado ou não-correlacionado.

Então, podemos tomar a média do *ensemble*[1] na equação 2.3, resultando na seguinte equação

$$m\frac{dv(t)}{dt} = -m\gamma < v(t) >$$

Essa equação diferencial pode ser resolvida utilizando as funções de Green [6]. Assumindo $v(0) = v_0$ e $x(0) = x_0$ como sendo a velocidade da partícula e o deslocamento da partícula, no instante $t = 0$, respectivamente:

$$v(t) = v_0 \exp((-\gamma/m)t) + \frac{1}{m}\int ds \exp(-(\gamma/m)(t-s))F(s) \tag{2.4}$$

[1] *Idealização que consiste de um grande número de cópias de um sistema, cada cópia representando um possível estado que o sistema real pode ter.*

$$x(t) = x_0 + \frac{m}{\gamma}[1 - \exp((-\gamma/m)t)]v_0 + \frac{1}{\gamma}\int ds[1 - \exp(-(\gamma/m)(t-s)]F(s) \quad (2.5)$$

As variáveis x(t) e v(t) são variáveis estocásticas e possuem as mesmas propriedades da força R(t).

Tomando a média sob o *ensemble* na equação 2.4 e 2.5 temos que as integrais são nulas, pois $< F(s) >= 0$, portanto $< v(t) >= v_0 \exp(-\frac{\gamma}{m}t)$ e $< x(t) - x_0 >= \frac{\gamma}{m}(1 - \exp(-\frac{\gamma}{m})t)v_0$. Vemos então que para a equação de Langevin a velocidade média da partícula sofre um decaimento exponencial com o tempo e que quando o tempo é muito longo temos que o deslocamento médio pode ser escrito como $\frac{m}{\gamma}v_0$.

2.2 A Equação de Langevin Generalizada

Na Equação de Langevin clássica (ELC), vista na seção anterior, não existe nenhum termo que considere o passado distante da partícula, ou ainda, não há um termo de memória associado, logo a equação clássica é usada apenas na representação de processos não correlacionados. Entretanto, na natureza existem fenômenos correlacionados de grande interesse, então uma equação que descreva tais fenômenos é necessária.

Mori (1957) propôs um formalismo baseando-se na ELC que considera ruídos correlacionados, esse formalismo ampliou o campo de atuação da Equação de Langevin para processos não-markovianos com ruídos correlacionados. No problema em questão a função de correlação das forças pode ser escrita como

$$C_F =< R(t)R(t') >= mk_bT\Gamma(t-t')$$

O termo $\Gamma(t-t')$ é a função memória e representa uma componente de fricção não instantânea, ou seja, dependente de acontecimentos anteriores. Introduzindo esse termo na ELC, o termo de dissipação é substituído por uma integral sobre tempos passados, chegando assim à equação

$$m\frac{dv(t)}{dt} = -m\int dt'\Gamma(t-t')v(t') + R(t) \tag{2.6}$$

que é chamada de Equação de Langevin Generalizada (ELG). Apartir da ELG, é possível obter o resultado particular, ou seja, a ELC escolhendo os termos e funções adequadas, nesse caso a ELC pode ser obtida da ELG fazendo $\Gamma(t) = 2\gamma\delta(t)$.

2.3 Função Memória

É natural se perguntar a origem e o significado da função memória. Adotando o teorema da flutuação-dissipação [7], temos que a origem da memória é atribuída a correlação do ruído. Supondo um ruído composto pelas vibrações de vários modos harmônicos, originado das diversas vibrações do meio, podemos escrever a força $R(t)$ como

$$R(t) = \sum_i g(\omega_i)cos(\omega_i t + \theta_k(\omega_i)) \tag{2.7}$$

onde $g(\omega_i)$ e ω_i são coeficientes obtidos a partir do espectro de frequências e $\theta_k(\omega_i)$ é uma distribuição gaussiana de números aleatórios entre $\pm\pi$. Multiplicando 3.4 por $R(0)$ e tomando a média temos que a função de correlação é

$$\begin{aligned} C_F &= < R(0)R(t)_k >= \sum_{i,j} g(\omega_i)g(\omega_j) < cos(\omega_i t + \theta_k(\omega_i))cos(\theta_k(\omega_j)) > \\ &\rightarrow mk_bT\int_0^\infty \rho(\omega)cos(\omega t)d\omega \end{aligned} \tag{2.8}$$

onde foi utilizado o fato de $< cos(\theta(\omega))cos(\theta(\omega')) >= \frac{1}{2}\delta(\omega-\omega')$ e passou-se para o limite contínuo. Definimos agora a densidade de estados como $\rho(\omega) = \frac{g(\omega)}{2}^2$.

Utilizando o desenvolvimento acima pode-se escrever a função memória

como

$$\Gamma(t) = \int_0^\infty \rho(\omega) cos(\omega t) d\omega \tag{2.9}$$

Se todas as frequências tiverem o mesmo peso probabilístico, a função $g(\omega_i)$ será constante e retorna-se ao ruído branco e consequentemente à equação de Langevin Clássica.

2.4 Tipo de difusão

Na natureza existem vários tipo de difusão, e que podem ser definidos utilizando uma generalização da Eq. (2.1), dada por $< x^2 > \sim t^\alpha$.

Observando o expoente α é possível dividir os tipos de difusão da seguinte maneira

$$\alpha \longrightarrow \begin{cases} < 1 \Rightarrow & \text{Subdifusão} \\ = 1 \Rightarrow & \text{Difusão normal} \\ > 1 \Rightarrow & \text{Superdifusão} \end{cases} \tag{2.10}$$

Em termos da constante de difusão é possivel relacionar com os casos acima da seguinte forma

$$D = \lim_{t\to\infty} \frac{< [x(t) - x(0)]^2 >}{2t} = \begin{cases} 0 \Rightarrow & \text{Subdifusão} \\ cte \Rightarrow & \text{Difusão normal} \\ \infty \Rightarrow & \text{Superdifusão} \end{cases} \tag{2.11}$$

Agora será mostrado que é possível obter o tipo de difusão do sistema conhecendo o comportamento da sua função memória [15]. Utilizando a função de correlação de velocidades $C_v(t-t') =< v(t)v(t') >$ é possível relacioná-la diretamente à constante de difusão da Eq. 2.11 através da fórmula de Kubo [7]

$$D = \int dt' C_v(t') \tag{2.12}$$

Fazendo a transformada de Laplace [16] da função de correlação de velocidades temos

$$\widetilde{C_v}(s) = \int dt \exp(-st) C_v(t). \tag{2.13}$$

Utilizando a fórmula de Kubo é possível reescrever a constante de difusão como

$$D = lim_{z\to 0}\widetilde{C_v}(s). \tag{2.14}$$

A transformada de Laplace da função de correlação de velocidades pode ser obtida facilmente multiplicando a ELG por $v(0)$ e tomando a média sobre um *ensemble* de partículas de mesmo tipo, as partículas possuem ruído independentes e não há correlação entre a sequência de ruído e as velocidades iniciais, portanto $< F(t)v(0) >= 0$. Assim o desenvolvimento pode ser assim descrito $m < \frac{dv(t)}{dt}v(0) >= -m\int dt'\Gamma(t-t') < v(t')v(0) > + < F(t)v(0) >$

$$\dot{C_v} = -\int dt'\Gamma(t-t')C_v(t'). \tag{2.15}$$

A transformada de Laplace da equação acima é

$$\widetilde{C_v}(s) = \frac{C_v(0)}{s+\widetilde{\Gamma}(s)}. \tag{2.16}$$

Utilizando o teorema de equipartição da energia [20] pode-se escrever a

constante de difusão da seguinte forma

$$D = \frac{C_v(0)}{\widetilde{\Gamma}(0)} = \frac{k_B T}{\widetilde{\Gamma}(0)}, \tag{2.17}$$

Esta equação mostra que conhecendo o comportamento de $\widetilde{\Gamma}(0)$ é possível conhecer o tipo de difusão do sistema. Fazendo a transformada de Laplace

$$\widetilde{\Gamma}(0) = \int dt \Gamma(t) \tag{2.18}$$

Se $\widetilde{\Gamma}(0)$ for finito, os sistema apresentará difusão normal, se $\widetilde{\Gamma}(0)$ for nulo ou infinito, o sistema apresentará superdifusão e subdifusão, respectivamente.

Utilizando a formulação de banho térmico (ver seção 2.3) é possível relacionar $\widetilde{\Gamma}(0)$ diretamente à densidade de estados de ruído do sistema $\rho(\omega)$, fazendo uso da equação 2.9.

3 *Unidade de Processamento Gráfico*

No início da década de 80, Jim Clark e um grupo de estudantes da universidade de Stanford formaram a Silicon Graphics, onde criaram a primeira estação de trabalho 3D batizada com o nome IRIS 1400. Essa estação de trabalho utilizava aceleradores gráficos que possuíam uma GeometryEngine [1], utilizando tecnologia Very Large Scale Integration[2], e um frame buffer[3]. Este hardware abriu caminho para os primeiros Computer-Aided Design (CAD) interativos e para visualização científica [8]. Desde então o hardware gráfico vem sendo linha de pesquisa em diversas áreas da ciência, primeiro no auxílio em simulações aeronáuticas, depois para aplicação em estações de trabalho e dispositivos médicos para auxiliar na reconstrução de imagens e por último na indústria do entretenimento, onde são utilizadas para dar mais realismo aos jogos eletrônicos.

Esse hardware evoluiu de tal forma que exigiu o desenvolvimento de uma unidade de processamento dedicada para lidar com esta crescente demanda, chamada Graphic Processing Unit (GPU). Nestas, o número de transistores utilizados superou o das Unidades Centrais de Processamento (CPUs) mais modernas. Com esse avanço uma atenção especial foi dada as GPUs visando utilizar esse poder de processamento em aplicações não gráficas [9] .

Neste capítulo será discutida a arquitetura do hardware gráfico e será

[1] *Processador de propósito específico para computação gráfica.*

[2] *Processo de criação de circuito integrado combinando milhões de transistores em um único chip.*

[3] *Dispositivo de vídeo que direciona a saída de acordo com um buffer de memória contendo um frame de dados completo.*

dada atenção somente às questões necessárias ao entendimento deste trabalho, questões relacionadas ao uso do hardware para aplicações gráficas serão omitidas.

3.1 Processadores de fluxo

A complexidade nos processamentos multimídias modernos tais como gráficos 3D, compressão de imagens e processamento de sinais fez com esse tipo de aplicação necessitasse de milhares de cálculos computacionais por segundo. A fim de atingir essa alta taxa de computação foram desenvolvidas arquiteturas de hardware específicos, não programáveis, porém o projeto desse tipo de arquitetura é extremamente penoso. Esse tipo de processamento passou a exigir uma certa flexibilidade, então o uso de um hardware programável se tornou uma opção interessante.

Aplicativos multimídia possuem um alto grau de paralelismo, permitindo a execução de milhões de cálculos computacionais em paralelo com uma taxa comunicação e armazenamento mínima, isso permite que um dado passe de uma unidade lógico-aritmética (ULA) diretamente para a próxima, formando assim o conceito de fluxo de dados.

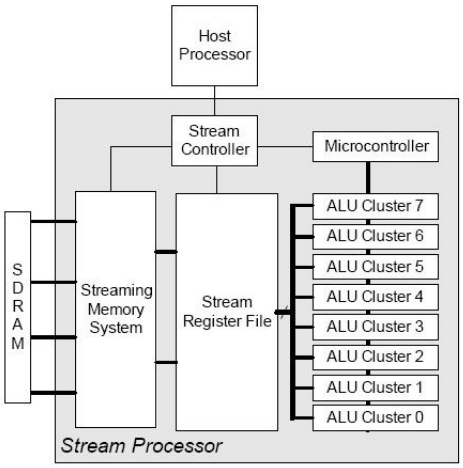

Figura 3.1: Arquitetura do processador de fluxo

Com o processador de fluxo é possível atingir uma alta taxa de operações por segundo, sem prejudicar a programabilidade, pois o processador explora de maneira eficiente a natureza paralela do programa. Nessa arquitetura os dados do programa são organizados em fluxos e os cálculos a serem efetuados são agrupados em kernels. Na figura 3.1, temos uma arquitetura de processador de fluxo que consiste de oito unidades lógico-aritméticas, o Host Processor (retângulo superior) representa a CPU do computador ao qual o processador de fluxo está acoplado. A CPU é responsável por submeter o *kernel* ao controlador de fluxo (*stream controller*), que prontamente envia as operações para o microcontrolador, que então as distribui nas ALUs. O programa fica armazenado na memória off-chip, durante toda a execução dados são lidos e escritos da memória off-chip no *Stream Register* Files e vice-e-versa [10].

3.2 Geforce 9800 GTX

Esta seção apresenta a arquitetura e as configurações detalhadas da placa de vídeo NVIDIA Geforce 9800 GTX que foi utilizada para executar a simulação do movimento browniano correlacionado. A figura abaixo mostra o diagrama da arquitetura da GPU versão G92 da NVIDIA.

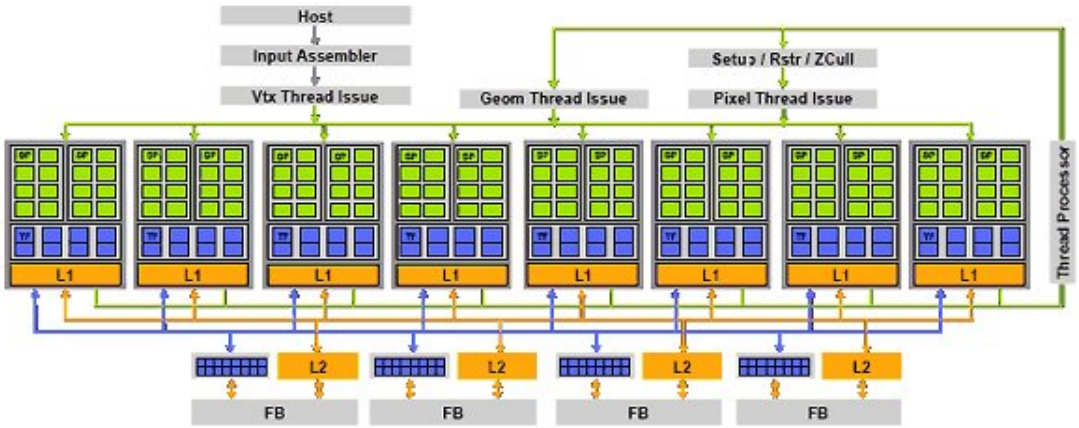

Figura 3.2: Arquitetura da GPU geração G92 da NVIDIA.

A figura 3.2 mostra a arquitetura da GPU da placa utilizada nesse trabalho, a GEFORCE 9800 GTX fabricada pela NVIDIA e montada pela XFX. Esse modelo utiliza a geração G92 de GPU com 1688MHZ de clock, possui 16 multiprocessadores (agrupamento de oito retângulos verdes) com 8 scalar processors cada (retângulos verdes), obtendo um total de 128 processadores de fluxo, memória global de GDDR3 512MB e 1100Mhz de clock atingindo uma largura de banda de até 70.4GB/s.

A geração G92 de GPU da NVIDIA utiliza Unidades Lógico-Aritméticas (ULA) escalares de uma dimensão, ou seja, cada unidade é capaz de tratar uma instrução escalar por ciclo de clock, essa instrução pode ser tanto uma operação em ponto-flutuante quanto uma operação de inteiros. Para operações em ponto-flutuante a arquitetura utiliza o padrão IEEE 754 que versa sobre a implementação de operações binárias em ponto flutuante. Nesse padrão a operação pode ser executada completamente no hardware, completamente no software ou em ambos [11].

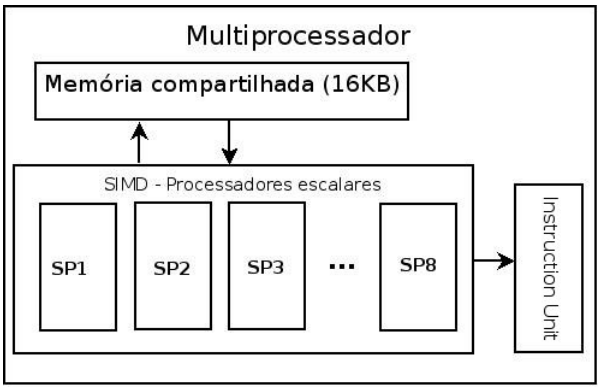

Figura 3.3: Esquema do multiprocessador

A arquitetura desse modelo é composta por dezesseis multiprocessadores com oito *scalar processors* (processadores escalares) cada, uma unidade de instruções *multithreaded* e uma memória compartilhada no próprio chip.

A figura 3.3 mostra o diagrama de um dos dezesseis multiprocessadores, cada multiprocessador possui uma memória de 16kbytes que é compartilhada pelos processadores scalares, o acesso a essa memória possui a latência de apenas 2 ciclos de clock permitindo assim um acesso local extremamente rápido. Assumindo duas operações em ponto-flutuante (uma adição e uma multiplicação) temos 2 (número de operações) x 1.688 (clock do processador) x 128 GFLOPS $\approx$ 432 GFLOPS. [12]

As unidades representadas por retângulos verdes correspondem aos SPs. Estes são completamente desacoplados do restante da GPU, mas apesar de serem independentes uns dos outros utilizam o mesmo gerador de clock de 1688MHz. O cache L1 é compartilhado entre os 16 *scalar processors* (SPs) de cada bloco, permitindo assim uma comunicação entre eles na forma de fluxo de saída.

Para gerenciar milhares de *threads* executando diferentes programas a NVIDIA adotou um novo conceito chamado *Single Instruction Multiple Threads (SIMT)* em que cada *thread* é mapeada e executada independentemente em um multiprocessador possuindo seu próprio espaço de endereçamento e registradores de estado, isso faz com que os SPs fiquem ocupados a maior parte do tempo trazendo maior utilização do hardware.

Do lado direito da figura 3.3 pode ser visto o processador de thread, esse elemento é responsável por manter os estados dos processamentos das threads que estão ativas e gerenciar o escalonamento das threads, como a idéia da arquitetura é executar massivamente as threads, esse elemento fica a maior parte do tempo ocupado.

Em uma CPU, quando ocorre um erro na busca de uma instrução/dado na memória cache (*data cache miss*) ela fica ociosa enquanto a instrução/dado é resgatada da memória principal, já na GPU esse erro não é tão grave, pois se ocorrer tal situação existem muitas threads prontas para executarem em um dos 128 SPs enquanto o dado é resgatado. Segundo os dados fornecidos pela NVIDIA a troca de contexto das threads tecnicamente não consome nenhum pulso de clock.

3.2.1 Hieraquia de memória

Esta seção apresenta a organização da memória dentro da placa de vídeo. A figura 3.4 mostra o diagrama da hierarquia de memória na arquitetura utilizada neste trabalho.

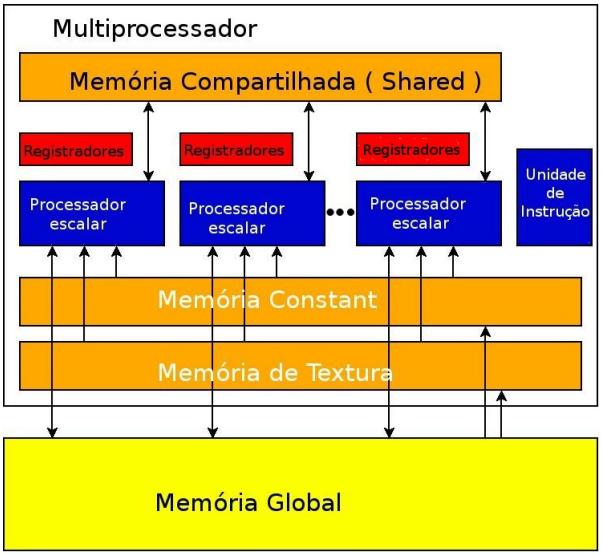

Figura 3.4: Hierarquia de memória no modelo CUDA

A memória é divida em várias regiões de acesso e latência diferenciados.

Temos, em nível local, registradores de 32 bits, por processador escalar, utilizados por cada *thread* para armazenar variáveis locais declaradas dentro do *kernel* na figura esses registradores estão representados pela região de cor vermelha.

Além dos registradores locais existe uma região de memória acessível a todos os *scalar processors*, essa região possui baixa latência e é compartilhada por cada thread de um bloco. Através dela é possível fazer cooperação entre as threads utilizando as mesmas variáveis, chamada memória *Shared*.

Outras regiões de memória de baixa latência são a Constant e Textura, elas são acessíveis a todos os processadores escalares, porém só o host pode escrever nessa região, do ponto de vista do dispositivo é uma memória apenas de leitura. A principal diferença entre as duas é que a memória Textura é essencialmente bidimen-

sional.

A área de maior espaço é a memória Global (área amarela), que é acessível a todas as threads para leitura e escrita, porém possui alta latência e o uso não coalescente (será explicado adiante) leva a uma redução de desempenho de até duas ordens de grandeza.

4 A Linguagem de Programação CUDA

Como a arquitetura do hardware é completamente diferente do que temos na CPU surgiu um novo problema: como balancear a programabilidade e o desempenho sem onerar o programador com o aprendizado de uma linguagem completamente nova? A resposta a essa pergunta veio com o aproveitamento de um padrão já dominado no mundo inteiro: o C ANSI. Surgiu então a linguagem de programação CUDA, uma linguagem que é uma extensão do C ANSI o que proporciona uma curva de aprendizado mais estreita, pois o programador pode aproveitar todo o conhecimento da linguagem C para programar em CUDA devendo dedicar somente um pouco de tempo para aprender os detalhes inerentes ao modelo de programação em fluxo.

Neste capítulo serão apresentados os detalhes da linguagem de programação CUDA, ressaltando as semelhanças entre esta e a linguagem C ANSI e demonstrando os aspectos básicos da linguagem de programação em fluxo.

4.1 CUDA, uma extensão da linguagem C

Para diminuir o tempo de aprendizado na programação da placa de vídeo, a NVIDIA adotou uma linguagem já largamente difundida em todo o mundo, o C ANSI. O objetivo dessa estratégia é aumentar a produtividade do programador, uma vez que ele precisará dedicar tempo apenas nas questões relativas à execução do código no placa de vídeo. A sintaxe da linguagem CUDA é diferente da linguagem

C ANSI apenas em certas formas de funções como __device__ e __global__ e tipos de dados adicionais, que serão explicados adiante, como: int2, int3, float2, float3, dim3, __shared__ etc.

O modelo de programação em fluxo impõe que o programador organize seu programa utilizando códigos que serão executados de forma serial na CPU, também chamada de Host, e código que será executado de forma massivamente paralela na GPU, também chamada de Device. Os trechos de código que serão executados de forma massivamente paralela são também chamados de Kernels, um *kernel* executa a mesma operação em diversos conjuntos de dados, é daí que vem a idéia de fluxo (de dados).

4.2 Tipos de dados

Como dito anteriormente, o CUDA implementa alguns tipos de dados especiais além dos tipos utilizados na linguagem C ANSI, nesta seção serão apresentados alguns detalhes.

4.2.1 Funções

4.2.1.1 __device__

Este tipo de dado é utilizado para definir uma função que será executada no dispositivo, porém não é um *kernel*. Ela só pode ser chamada no *device*, ou seja, dentro de um *kernel*. Algumas restrições se aplicam a esse tipo de função como: não poder ser recursiva, não permitir o uso de variáveis estáticas em seu corpo, não poder passar seu endereço para outra função e não é permitido uso de quantidade variável argumentos (como faz a função printf() [13] do C ANSI). Exemplo: Uma função para gerar uma seqüência de números aleatórios que serão utilizados dentro do *kernel*.

```
__device__ gerador_numeros(int quantidade, float semente, float *array)
{
    corpo da função que gera os números aleatórios coloca os valores em array.
}
```

4.2.1.2 __global__

Usado para definir uma função como sendo um *kernel*, esse *kernel* só poderá ser chamado do host e será executado no *device* de forma massivamente paralela, ou seja, as mesmas operações serão executadas por milhares de threads. Algumas restrições se aplicam a esse tipo de função como: não suportar recursão, não permitir uso de variáveis estáticas em seu corpo, não permitir uso de quantidade variável de argumentos, não retornar valores (void) e não poder ser usada em conjunto com o tipo __host__.

Exemplo: Kernel para resolver milhares de equações integro-diferenciais, cada uma associada a uma partícula completamente independente das demais.

```
__global__ resolve_equacoes(int quantidade, float *resultado)
    corpo da função que resolve as equações coloca os valores no array resultado.
```

Os kernels possuem um modo diferente de chamada que as funções do C ANSI, para chamar esse tipo de função é necessário usar a tag <<<Dg, Db, Ns, S >>> entre o nome e os argumentos da função. Os valores dentro de <<< >>> são as configurações do padrão de execução do *kernel* que é constituído por: Dg: é do tipo dim3 (será explicado adiante) e especifica a dimensão e tamanho do grid, sendo assim esse valor limita quantidade de blocos que o *kernel* irá executar (Dg.x * Dg.y), por exemplo: se o valor de Dg é 128 poderemos executar 128*128 = 16384 blocos

de threads, note que Dg.z não é utilizado para efetuar o cálculo; Db: é do tipo dim3 e especifica a dimensão e tamanho de cada bloco de threads, assim temos que Db.x * Db.y * Db.z = Número de threads por bloco; Ns: esse é um parâmetro opcional, é usado para especificar o número de bytes de memória compartilhada que é alocada por bloco dinamicamente; S: é do tipo cudaStream_t e especifica o fluxo associado, é opcional e seu valor por padrão é 0.

Para concluir vemos um exemplo para chamar o *kernel* mostrado acima se desejássemos executar 32 grids unidimensionais de 128 blocos, ou seja 4096 threads (1 thread por partícula):

```
resolve_equacao <<<32,128>>>(4096, resultado);
```

4.2.1.3 __host__

Declara uma função que será executada no host e só pode ser chamada por ele. As funções declaradas dessa forma utilizam somente códigos em C ANSI e serão executadas no host, onde a execução se dá de forma serial. Esse tipo de dado é o padrão e pode ser omitido, isto é, se uma função não for explicitamente declarada como __global__ ou __device__ ela é assumida sendo do tipo __host__.

Exemplo: função que calcula as médias dos deslocamentos e das velocidades calculadas pelo *kernel*, armazenando os resultados em dois arrays.

```
__host__ calcula_medias(float *resultado, float *x_medio, float *v_medio)
{
      função que calcula as médias dos valores armazenados no array resultado

                        e armazena nos arrays x_medio e v_medio

}
```

4.2.2 Variáveis

4.2.2.1 __device__

Variáveis declaradas como __device__ são utilizadas dentro do *device*, se não for usada em conjunto com os tipos, que serão explicados adiante, __shared__ e __constant__ a variável reside na memória global do dispositivo, possui o tempo de vida da aplicação e é acessível a todas as threads dentro do grid e ao host através da biblioteca de tempo de execução.

4.2.2.2 __constant__

Utilizado juntamente com o __device__, de forma opcional, informa ao compilador que a variável será armazenada no espaço de memória constant. Esse espaço de memória, como discutido anteriormente, é muito mais rápido que o espaço de memória global, porém tem seu tamanho extremamente limitado (16KB). O device não tem permissão de escrita nesse tipo de memória, sendo assim, apenas o host pode escrever valores nela funcionando na visão do device como uma coleção de variáveis estáticas. Da mesma forma que uma variável __device__ uma variável declarada dessa forma possui o tempo de vida da aplicação e é acessível a todas as threads dentro do grid e ao host através da biblioteca de tempo de execução.

4.2.2.3 __shared__

Utilizado juntamente com __device__, de forma opcional, informa ao compilador que a variável será armazenada no espaço de memória compartilhada. Esse espaço de memória é compartilhado por todas as threads dentro de um mesmo bloco, é uma memória de baixa latência, possui o tempo de vida do bloco e só é acessível às threads do bloco a qual pertence, apesar do benefício de ser uma memória de rápido acesso seu limite físico é de apenas 16KB. Após a utilização dessa memória pelas threads é necessário o uso de uma função, chamada __syncthreads(), que sincroniza as threads e faz a escrita da memória compartilhada na memória global, assim o dado fica acessível a todas as threads do grid. Todos os tipos de variáveis

comentadas aqui são executadas no dispositivo, a principal diferença entre elas é a região de memória em que ficam localizadas. Isso tem um grande impacto no desempenho da aplicação, uma vez que, a região de memória global é grande, porém possui uma latência também grande, já as outras regiões são pequenas, mas com baixíssima latência.

4.2.2.4 Variáveis Built-in

Além dos tipos de variáveis comuns ao C ANSI como: int, char, float, etc; o CUDA implementa outros tipos de variáveis que podem ser úteis em programas que usam variáveis multidimensionais.

char1, uchar1, char2, uchar2, char3, uchar3, char4, uchar4, short1, ushort1, short2, ushort2, short3, ushort3, short4, ushort4, int1, uint1, int2, uint2, int3, uint3, int4, uint4, long1, ulong1, long2, ulong2, long3, ulong3, long4, ulong4, float1, float2, float3, float4, double2

Variáveis desses tipos podem ser comparadas ao tipo de dados struct da linguagem C ANSI, elas agrupam vários membros dentro da mesma variável, porém no caso do CUDA esses membros são todos do mesmo tipo, por exemplo:

Suponha que na variável abaixo desejamos armazenar os valores de velocidade de uma partícula em três dimensões:

```
float3 velocidade;

velocidade.x = 5.0;
velocidade.y = 1.0;
velocidade.z = 7.5;
```

Temos aqui que a velocidade no eixo X é 5.0, no eixo Y é 1.0 e no eixo z é 7.5. Da mesma forma podemos usar as variáveis int2, int3, etc. Algumas das variáveis build-in de extrema importância no modelo CUDA são:

gridDim: Variável do tipo dim3 usada na representação de dimensões do grid.

blockIdx: variável do tipo uint3 usada na representação de um bloco dentro do grid. blockDim Variável do tipo dim3 usada na representação da dimensão do bloco.

threadIdx: Variável do tipo uint3 usada na representação do índice de uma thread dentro de um bloco.

4.3 Gerenciamento de Memória

Até o momento, a quantidade de memória de uma placa de vídeo é muito menor que a utilizada nos computadores, então surge um novo problema: como fazer o armazenamento e transferência dos dados entre a memória principal do computador e a memória off-chip presente no dispositivo? Vimos até agora que as semelhanças entre a linguagem CUDA e a linguagem C ANSI são muito grandes, e da mesma forma que é possível no C ANSI fazer alocação de uma parte da memória principal utilizando a função malloc() é possível fazer alocação de parte da memória de vídeo utilizando funções implementadas na linguagem CUDA, como o cudaMalloc().

A memória do dispositivo pode ser alocada de forma linear ou como CUDA arrays. Alocando a memória de forma linear, temos a utilização de um espaço de endereçamento de 32-bits, assim cada entidade pode ser referenciada utilizando ponteiros, já na utilização da alocação em CUDA arrays é usado um layout de memória otimizado para uso de texturas, podendo ser uni-dimensional, bi-dimensional ou tri-dimensional. Há restrições para uso de CUDA arrays, pois os mesmos só podem ser lidos pelos kernels através de operações sobre texturas, o host pode ter acesso utilizando funções de cópia de memória, como será visto adiante.

4.3.1 Funções para alocação de memória

Para alocar parte da memória do dispositivo utilizando o modelo linear utilizamos a função cudaMalloc() ou cudaMallocPitch(). Essas funções são muito semelhantes as utilizadas na linguagem C ANSI, pois da mesma forma precisamos

especificar um ponteiro para o armazenamento dos dados e a quantidade de bytes a serem alocados. Suponha que desejamos alocar 32 elementos lineares do tipo float e armazená-los num array chamado numeros_aleatorios:

```
float* numeros_aleatorios;
cudaMalloc((void**)&numeros_aleatorios, 32*sizeof(float));
```

Após essa chamada as posições de memória alocadas podem ser acessadas da mesma forma que no C ANSI, utilizando a sintaxe numeros_aleatorios[n], onde n é o índice do elemento do array.

A função cudaMallocPitch() é utilizada para alocar matrizes, tendo a certeza que a alocação é apropriadamente ajustada com requisitos específicos de computação gráfica. Esse tipo de alocação favorece o desempenho utilizando uma função (cudaMemcpy2D()) de alto desempenho para cópias de matrizes entre as regiões de memória do dispositivo, tais matrizes são geralmente utilizadas em computação gráfica, pois muitas das aplicações manipulam e exibem dados em tempo real. Essa função exige um inteiro denominado pitch que serve para acessar os elementos do array2D, vamos supor que desejamos alocar um array de elementos float de k linhas de n bytes, então:

```
float* numeros_aleatorios;
int pitch;
cudaMallocPitch((void**)&numeros_aleatorios, pitch, k * sizeof(float), n);
```

Para utilizar as posições de memória alocadas é necessário criar um *kernel*:

```
__global__ void Le_array2D(float* numeros_aleatorios, int pitch)
{
   for (int r = 0; r < largura; ++r) {
      float* linha = (float*)((char*)numeros_aleatorios + r * pitch);
      for (int c = 0; c < n; ++c) {
         float elemento = linha[c];
      }
   }
}
```

4.3.2 Funções para cópia de dados

O CUDA define algumas funções para fazer a interface entre a memória do host (memória principal) e a memória do dispositivo, vamos detalhar essas funções.

4.3.2.1 cudaMemcpy()

Utilizada para copiar dados da memória do host para a memória global da placa de vídeo ou vice-e-versa.

cudaMemcpy(void* destino, const void* origem, size_t n,enum CudaMemcpyHostToDevice);

Essa instrução copia n bytes da área apontada pelo ponteiro origem localizada no host para a área apontada pelo ponteiro destino localizada no device.

```
cudaMemcpy(void* destino, const void* origem, size_t n,enum CudaMemcpyDeviceToHost);
```

Essa instrução copia n bytes área apontada pelo ponteiro origem localizada no device para a área apontada pelo ponteiro destino localizada no host.

Chamar essa função utilizando ponteiros origem e/ou destino que não coincidam com as configurações sugeridas na chamada da função gera um comportamento indefinido. A função cudaMemcpy() retorna valores que podem ser usados para identificar eventuais problemas; os valores retornados são: cudaSuccess (executada com sucesso), cudaErrorInvalidValue (erro na quantidade de bytes), cudaErrorInvalidDevicePointer (ponteiro não alocado no device) e cudaErrorInvalidMemcpyDirection (erro na direção de cópia).

4.3.2.2 cudaMemcpy2DToArray

Utilizada para copiar dados residentes em posições de memória alocadas com cudaMallocPitch(), por exemplo:

```
cudaMemcpy2DToArray(struct cudaArray* cuArray, size_t destinoX, size_t destinoY, const void* origem, size_t spitch, size_t k, size_t n, enum cudaMemcpyDeviceToDevice);
```

Neste exemplo é copiada uma matriz (k linhas de n bytes) da região de memória apontada por origem para o array cuArray iniciando no canto superior esquerdo (destinoX, destinoY), spitch é a largura na memória (em bytes) do array apontado por origem. Esta função retorna os mesmo indicativos de erros discutidos na seção anterior.

4.3.3 Funções para liberação de memória

Da mesma forma que o C ANSI implementa funções com a finalidade de dizer ao sistema operacional que não precisa mais da memória alocada, o CUDA também o faz.

4.3.3.1 cudaFree()

Função utilizada para liberar uma região de memória da GPU previamente alocada com cudaMalloc(). É usada da seguinte forma

```
cudaFree(numeros_aleatorios);
```

Essa função é chamada no host e retorna cudaSucess, se a memória foi desalocada com sucesso, ou cudaErrorInvalidDevicePointer em caso de falha, essa falha pode ser ocasionada pela tentativa de desalocar uma região de memória não previamente alocada ou caso seja chamada duas vezes para a mesma região.

4.3.3.2 cudaFreeArray()

Função utilizada para liberar uma região de memória da GPU previamente alocada com cudaMallocArray(). É usada da seguinte forma

```
cudaFreeArray(numeros_aleatorios);
```

Essa função recebe como argumento a matriz a ser desalocada e retorna cudaSucesss, se a memória foi desalocada com sucesso, ou cudaErroInitializationError.

5 *Desenvolvimento*

O desenvolvimento desse trabalho foi dividido nas seguintes etapas:

1. Obtenção do código paralelo com MPI[14] e serial[15];
2. Projeto do modelo lógico do código para execução massivamente paralela;
3. Adaptação do gerador de números pseudo-aleatórios Mersenne Twister da biblioteca Gnu Scientific Library (GSL) para execução massivamente paralela;
4. Implementação e execução do algoritmo numérico para resolução da ELG na GPU;
5. Execução do código serial em um processador Intel Q6600 2.4 Ghz;
6. Execução do código paralelo com MPI em um cluster de 13 núcleos de processamento com processador Intel Q6600 2.4 Ghz;
7. Análise dos tempos de execução.

5.1 Projeto Lógico

O algoritmo numérico que foi usado neste projeto [15] propõe uma expansão do termo de ruído da ELG a fim de tornar a resolução eficiente do ponto de vista computacional, assim as integrações podem ser feitas de forma iterativa.

Como cada partícula precisa de um grande conjunto de números aleatórios (que estão associados às fases da função ruído), para isto foi utilizado um gerador de números pseudo-aleatórios de período longo e de alta qualidade dos números

chamado Mersenne Twister [18]. Esse algoritmo foi obtido da biblioteca Gnu Scientific Library e adaptado para executar na GPU sob a forma de um *kernel* chamado *RandomGPU*.

Para o preenchimento do array densidade de estados, foi criado um *kernel* chamado *density_power* que é executado na GPU e faz uso da memória *constant* (memória de rápido acesso).

Sabendo que no movimento browniano as partículas do ensemble são completamente independentes, a abordagem de paralelismo usada foi a de dividir as partículas entre as threads, ou seja, cada thread resolve a ELG para uma partícula distinta.

Por fim, foi utilizado um *kernel* chamado *results* que calcula as médias estatísticas e fornece ao host duas matrizes, uma contendo as velocidades quadráticas médias e outra contendo os deslocamentos quadráticos médios, essas matrizes são então armazenadas em arquivos para posterior análise gráfica.

5.2 Implementação

Como dito anteriormente, na simulação do movimento browniano correlacionado as partículas do *ensemble* são completamente independentes umas das outras. Então, foi realizada uma paralelização trivial, ou seja, cada thread é responsável pela resolução da ELG de uma partícula, sendo que os dados foram dispostos na forma de arrays bidimensionais onde cada linha corresponde a uma partícula e cada coluna a uma posição, velocidade ou fase aleatória.

O *kernel* de densidade de estados foi implementado utilizando a memória *constant*, apesar do tamanho reduzido desse tipo de memória (16Kb) ela é suficiente para armazenar os valores dos arrays d_g[NUM_WMAX] e d_exponent[NUM_WMAX], onde NUM_WMAX vale 1024. Assim, o tamanho máximo de ocupação é $1024 * 4bytes * 2 = 8192bytes = 8Kb$. Além do benefício da baixa latência inerente ao tipo de memória, foi utilizado acesso de forma coalescente. O acesso coalescente

é obtido quando cada thread acessa a posição de memória com o número de seu *thread id*. A figura 5.1 mostra como isto é feito

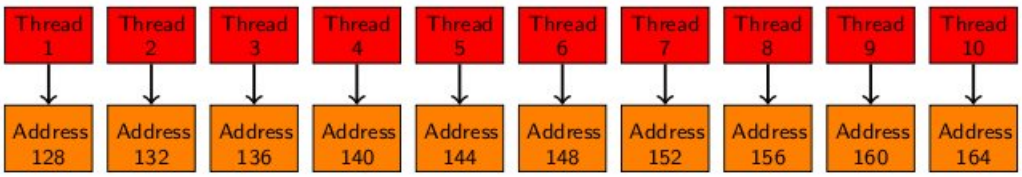

Figura 5.1: Acesso coalescente

Como pode ser visto na figura acima a *thread 1* acessa a primeira posição de memória, a *thread 2* acessa a segunda posição e assim por diante. Esse tipo de acesso reduz o impacto da alta latência da memória global em até duas ordens de grandeza, visto que, um acesso a um *float* não coalescente custa de 200 a 400 ciclos de clock enquanto que um acesso coalescente custa de 2 a 4 ciclos de clock. Apesar da redução da latência de acesso ser maior quando se usa acesso coalescente na memória global os outros tipos de memória também se beneficiam desse tipo de acesso.

O *kernel* responsável pela resolução da ELG faz uso da memória global (alta latência), pois os arrays utilizados em seu interior são demasiadamente grandes para serem armazenados na memória de baixa latência. O que esse *kernel* faz é basicamente resolver a ELG armazenando os valores do deslocamento e velocidade de cada partícula em um array bidimensional. O acesso à memória global feito por esse *kernel* é feito coalescente.

Após o cálculo do array de posições e velocidades os dados são tratados no *device* através de um *kernel* chamado *results*, que calcula os deslocamentos quadráticos médios e as velocidades quadráticas médias, esses são os resultados de interesse na análise da ELG, pois permite a caracterização de diversos fenômenos, como o tipo de difusão.

5.2.1 Gerador de números aleatórios

Como foi dito anteriormente, neste trababalho foi utilizado o gerador de números aleatórios Mersenne Twister (MT). Este gerador de números pseudo-aleatórios foi desenvolvido em 1997 por Makoto Matsumoto e Takuji Nishimura, e é baseado em recorrência linear de matriz sobre campos binários finitos e fornece números pseudo-aleatórios entre 0 e 1. O gerador de números pseudo-aleatórios MT foi escolhido por ser capaz de gerar números pseudo-aleatórios de alta qualidade, possuindo um período de $2^{19937} - 1 \approx 4.3^{6001}$ e possuindo uma correlação entre os números sucessivos desprezível.

A simulação do movimento browniano correlacionado exige que cada partícula possua uma fase aleatória, entre $-\pi$ e π, no termo de ruído. Ao utilizar a implementação do MT da NVIDIA[19], que já está pronto para executar na GPU, foi observado que a sequência de números aleatórios obtida não era realmente aleatória, analisando o código foi observado que todas as threads estavam recebendo a mesma semente aleatória, então a solução foi incluir uma variável que é única para cada thread, chamada *thread id* (tid). Multiplicando a semente aleatória pela *thread id* foi possível obter uma boa qualidade nos números. Além disso foi feita uma alteração na linha que calcula os números entre 0 e 1 fazendo com que o MT gere números entre $-\pi$ e π.

O MT da GSL possui muitos *bugfixes* e comparando o código do MT da NVIDIA foi observada uma grande diferença. Com receio de que a implementação da NVIDIA não incorporasse tais bugfixes, o MT da GSL foi adaptado à placa de vídeo. Basicamente foi feita um reaproveitamento do esqueleto do MT da NVIDIA e o corpo do *kernel* foi substituído pelo MT da GSL.

A figura abaixo mostra a estrutura do gerador de números aleatórios dentro da simulação

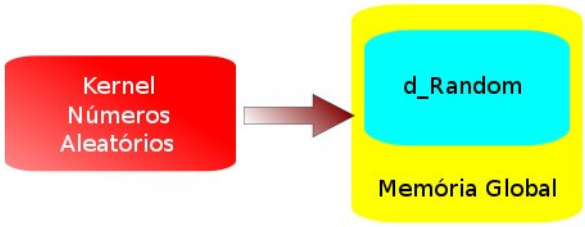

Figura 5.2: Kernel de geração de números aleatórios

Quando o *host* chama o *kernel* de geração de números aleatórios a matriz onde os dados serão armazenados é passada como parâmetro, essa matriz se chama d_Random. Como dito anteriormente, cada partícula possui o seu próprio conjunto de números aleatórios, então essa matriz é muito grande para ser armazenada em memória de baixa latência, pois estas são pequenas, então os dados são armazenados na memória global e para minimizar o efeito da alta latência é utilizado acesso coalescente.

5.2.2 Preenchimento do array de densidade de estados

Nesta etapa, o *kernel* calcula o array de densidade de estados que é utilizado no *kernel* de resolução da Equação de Langevin Generalizada. Abaixo segue o diagrama.

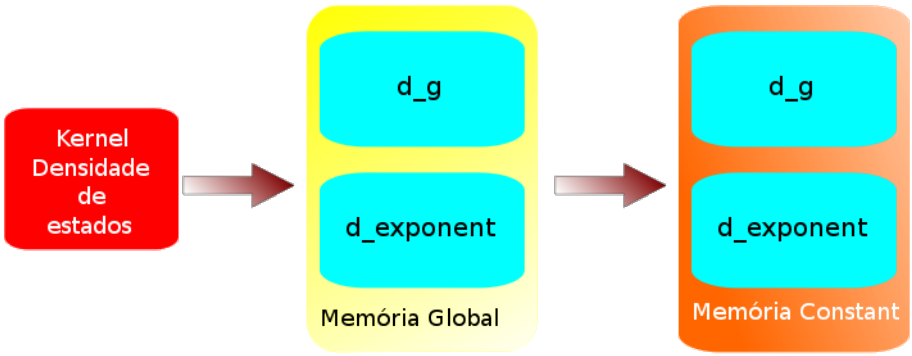

Figura 5.3: Kernel de densidade de estados

A figura 5.3 mostra que os arrays de densidade de estados são armazenados na memória *constant*, isso foi feito dessa forma porque os arrays ocupam apenas 8Kb dos 16Kb disponíveis nesse tipo de memória e como todas as threads irão ler esses dados é uma boa forma de otimizar a simulação, uma vez que essa memória possui baixa latência. Como a memória *constant* só pode ser escrita através do *host* os arrays foram primeiro escritos em memória global e depois de resgatados pelo host, através da função de cópia de memória *cudaMemCpy()*, foram escritos na memória *constant*.

5.2.3 Resolução da Equação de Langevin Generalizada

Nesta etapa o programa recebe os arrays de números aleatórios e densidade de estados gerados nos *kernels* anteriores e faz o cálculo das posições e velocidades das partículas. A figura 5.4 mostra como está organizado o *kernel* de resolução da Equação de Langevin Generalizada.

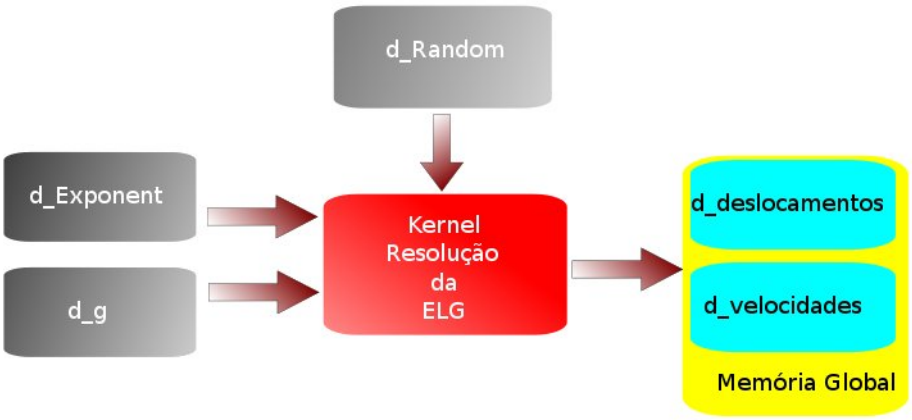

Figura 5.4: Kernel de resolução da Equação de Langevin Generalizada

Como a figura acima mostra, o *kernel* recebe como parâmetros a matriz de números aleatórios e os arrays de densidade de estados e gera na saída um conjunto de posições e velocidades das partículas, cada partícula possui o seu próprio conjunto de posições e velocidades, esses dados são grandes demais para serem

armazenados em memória de baixa latência, então foi utilizada a memória global (região amarela da figura) com acesso parcialmente coalescente. O método de paralelismo utilizado foi a paralelização trivial, isto é, cada *thread* calcula as posições e velocidades de uma partícula distinta de forma serial.

5.2.4 Tratamento de dados

Com a finalidade de tratar os dados obtidos no *kernel* anterior foi utilizado um *kernel* chamado *results* que realiza médias estatísticas e retorna como resultado o deslocamento quadrático médio e velocidade quadrática média. A organização deste *kernel* é mostrada na figura 5.5.

Figura 5.5: Kernel de tratamento dos resultados

O *kernel* mostrado na figura 5.5 recebe como parâmetros as matrizes de posições e velocidades das partículas, realiza médias sobre todo o *ensemble* e tem como saída os deslocamentos quadráticos médios e velocidades quadráticas médias, essa saída é armazenada na memória global e o acesso é coalescente. Depois que esses valores são calculados o *host* extrai essas informações da memória global e armazena em dois arquivos chamados *xqm.dat* e *vqm.dat* que guardam, respectivamente, os deslocamentos quadráticos médios e velocidades quadráticas médias.

5.2.5 Visão geral da simulação

A figura 5.6 mostra a visão geral da implementação quando todo os elementos vistos nessa seção são conectados formando um só programa.

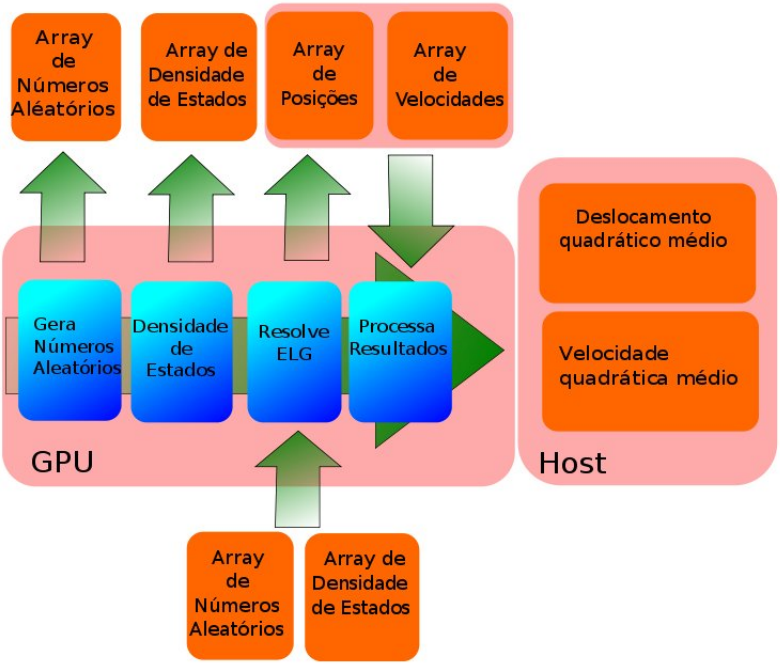

Figura 5.6: Diagrama da simulação

Na figura acima a seta grande indica a linha temporal do programa, os *kernels* são executados de forma sequencial e os resultados utilizados como parâmetros *kernels* posteriores.

5.3 Coleta de dados

Para analisar o desempenho do sistema proposto, a simulação serial foi executada de três formas distintas

Código serial: Compilado com o *Intel C Compiler* versão 10.1 e executado em um processador Intel Core 2 Quad Q6600, porém utilizando apenas um núcleo de processamento.

Código paralelo utilizando MPI: Compilado com o *MPICC* e *Intel C Compiler* versão 10.1 e executado em um cluster de 13 núcleos, sendo um deles respon-

sável apenas por receber os dados calculados pelos outros 12, equipado com processadores Core 2 Quad Q6600 e rede Gigabit.

Código paralelo utilizando GPU: Executado na placa de vídeo modelo 9800 GTX fabricado pela Nvidia Corporation e montado pela XFX, compilado com o compilador proprietário NVCC.

O compilador da Intel foi escolhido por ser capaz de otimizar o código de forma automatizada. Em todos os casos acima o tempo de execução foi obtido através de uma função *measure_elapsed_time()* que, basicamente, recebe uma estrutura do tipo *timeval* definida na biblioteca <*time.h*> e a trata de forma a obter duas componentes, uma relacionada aos segundos e outra aos microssegundos, fazendo o tratamento dessa forma é possível obter uma alta precisão na medida do tempo.

Cada experimento consistiu da medida do tempo de execução total do programa utilizando os seguintes parâmetros

- NUM_WMAX = 1024 (Número de partículas)
- T = 0.08 (Temperatura)
- G0 = 0.25 (Constante de densidade de estados)
- Kb = 1.0 (Constante de Boltzmann)
- ws = 0.5 (Frequência de corte)
- eta = 0.0 (Expoente da densidade de estados)
- TOTALTIME = 1000 (Tempo total)

O número de partículas foi variável, assumindo os valores: 128, 256, 512, 1024, 2048, 4096, 8192 e 16364. Foram realizados dez experimentos para cada número de partículas e com as médias aritméticas desses tempos de execução foi montada a seguinte tabela

Tabela 5.1: Tempo de execução do algoritmo numérico em segundos

N. de Partículas	Serial	Paralelo com MPI	Paralelo em GPU
128	86.69	30.08	29.91
256	172.93	58.27	31.39
512	345.91	113.71	32.22
1024	694.04	225.44	39.79
2048	1396.43	449.33	51.37
4096	2765.95	899.82	101.80
8192	5629.84	1793.83	202.81
16384	10990.24	3580.51	404.64

Fonte: Próprio autor

Comparar esses dados analisando apenas as proporções iniciais não é uma boa idéia, pois no inicio as taxas de entrada de saída de dados são muito altas, o que pode causar inacurácia na leitura dos ganhos reais de desempenho. A forma mais adequada de analisar os dados é calcular o speedup através da seguinte equação

$$Speedup = \frac{Tempo_{CPU}}{Tempo_{GPU}} \tag{5.1}$$

e então, plotar um gráfico monolog do speedup pelo número de partículas, pois assim poderá ser visto o ponto em que o ganho irá se estabilizar.

5.4 Análise de dados

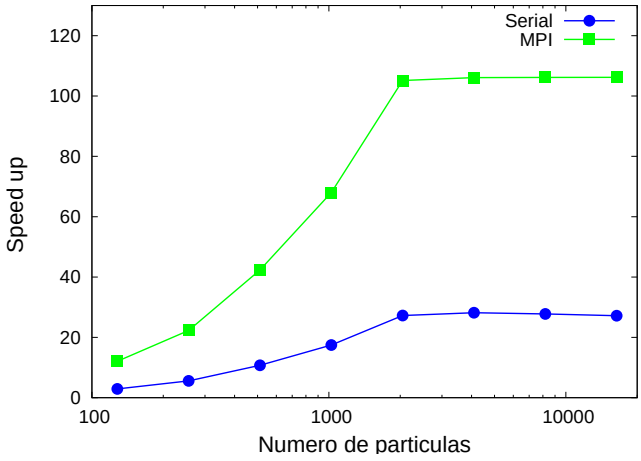

Figura 5.7: Resultado do teste de desempenho

A curva MPI foi obtida multiplicando o speed up pela quantidade de núcleos de processamento utilizados do cluster, no caso 12, isso foi feito com o objetivo de normalizar a comparação, ou seja, fazer uma comparação do cálculo de um núcleo para o código serial e um núcleo para o código MPI. Essa normalização oferece informações a cerca do tempo gasto em comunicação e disparo de threads.

A figura 5.7 mostra que a taxa de entrada e saída dos dados até aproximadamente 2048 partículas é muito alta e o speed up é crescente, porém quando a quantidade de processamento é muito superior a taxa de entrada e saída forma-se um platô e o ganho de desempenho do algoritmo executado na GPU estabiliza em aproximadamente 106 vezes em relação ao algoritmo utilizando a bibliteca MPI e 27 vezes para o algoritmo serial.

Comparando o speedup serial e MPI percebe-se uma diferença no ganho de desempenho muito grande, então é possível começar questionar suas causas. Vários fatores podem ser analisados para justificar esse fato, entre eles a taxa de comunicação entre os nós do cluster, o que leva a questionamentos a cerca das arquiteturas multicores emergentes da intel e AMD.

6 *Considerações Finais*

Pela análise de dados do capítulo anterior vemos que o uso de unidades de processamento gráfico para aplicações que exijam alto poder de processamento é uma excelente alternativa. Primeiro porque o custo é muito inferior ao de clusters de computadores e hardwares auxiliares, e segundo, porque o desempenho é incrivelmente mais alto.

O grande salto de desempenho da GPU quando comparado com o algoritmo executado em cluster utilizando a linguagem MPI abre diversas indagações sobre o gargalo de rede e custo de disparo de threads inerentes a programação paralela. Esse dado mostra que não adianta criar arquiteturas multicores se não for dada uma atenção especial no gerenciamento de processos.

Apesar do ganho de desempenho ser muito grande ao se utilizar a GPU para aplicações exigentes é preciso analisar o custo de programação envolvido, uma vez que somente as aplicações massivamente paralelas poderão tirar proveito dessa arquitetura. Há também a necessidade de uma análise minunciosa do gerenciamento de memória, programas que fazem muito acesso a disco podem encontrar um gargalo muito grande ao utilizar a memória global. Então, a aplicação ideal para ser executada na GPU é aquela que consome muito tempo de processamento e pouco tempo de escrita/leitura em memória.

APÊNDICE A – Algoritmo Numérico

Parte dos problemas encontrados na literatura deve-se aos processos numéricos de integração. Por exemplo, a integral

$$I = \int_0^t \Gamma(t-t')v(t')dt' \tag{A.1}$$

consome um tempo de processamento muito grande, pois ela precisa ser recalculada toda vez que se passa de um tempo t para um tempo $t+\Delta t$.

Como foi visto no capítulo 2, é possível escrever a função memória da seguinte forma

$$\Gamma(t) = \frac{1}{2mK_bT}\sum_i \frac{g(\omega_i)^2}{2} cos(\omega_i t) \tag{A.2}$$

substituindo essa equação na equação 2.7 é possível mostrar que a ELG pode ser escrita da seguinte forma

$$m\frac{d}{dt}v(t) = -\Sigma[\frac{g(\omega_i)^2}{2K_bT}K(t) + g(\omega_i)cos(\omega_i t + \theta_k(\omega_i))] \tag{A.3}$$

onde K(t) é dado por

$$K(t) = (cos(\omega_i t)\int_0^t cos(\omega_i t')v_k(t')dt' + sin(\omega_i t)\int_0^t sin(\omega_i t')v_k(t')dt')$$

Assim as integrais temporais podem ser calculadas de forma iterativa, isso torna possível acumular seus resultados fazendo com que seja gasto o mesmo tempo de processamento a cada passo.

Referências Bibliográficas

[1] NVIDIA Corporation. 2008. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. Versão 2.0.

[2] NYLAND, L.; HARRIS, M.; PRINS, J. 2007. *Fast n-body simulation with CUDA*. GPU Gems 3. H. Nguyen, ed. Addison-Wesley.

[3] BROWN, R. *A brief Account of Microscopical Observations made in the Months of June, July, and August, 1827, on the Particles contained in the Pollen of Plants; and on the general Existence of active Molecules in Organic and Inorganic Bodies*, Philosophical Magazine. 1828.

[4] PERRIN, J. *Brownian movement and molecular reality, translated from the Annales de Chimie et de Physique*, Londres, 1910.

[5] STACHEL, J. *O ano miraculoso de Albert Einstein: cinco artigos que mudaram a face da física*. Rio de Janeiro, UFRJ, 2001.

[6] MARION, J. B.; THORNTON, S. T. *Classical dynamics of particles and systems*. Fort Worth: Saunders College Publishing, 1995.

[7] KUBO, R. *The fluctuation-dissipation theorem*. Rep. Prog. Phys., v. 29, 1966.

[8] BAUM, D. *Computer Graphics*. p. 65, 1998.

[9] BUCK, I. Brook Language Specification http://merrimac.stanford.edu/brook, Outubro 2003.

[10] KHAILANY B. et al.. *Exploring the VLSI Scalability of Stream Processors*, 2003.

[11] IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985, IEEE, 1985.

[12] http://www.nvidia.com/object/geforce_9800gtx.html

[13] KERNIGHAN, W.; RITCHIE, M. *The C Programming Language*, 1a ed., Englewood Cliffs, NJ: Prentice Hall, 1978.

[14] VAINSTEIN, M., *A Conjectura de MOBH: Processos difusivos em sistemas desordenados*, 2003. Dissertação (Mestrado em Ciências Físicas Aplicadas). Instituto de Física, Universidade de Brasília, Brasília, 2003.

[15] MORGADO, R., *Difusão e tempo de escapamento para um sistema com memória de longo alcance*, 2001. Apêncice A. Dissertação (Mestrado em Ciências Físicas Aplicadas). Instituto de Física, Universidade de Brasília, Brasília, 2001.

[16] BUTKOV, E. *Física matemática*. Rio de Janeiro: Guanabara Koogan, 1988.

[17] HUANG, K. *Statistical mechanics*. New York: john Wiley & Sons, 1987.

[18] MATSUMOTO, M; NISHIMURA, T. *Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator*, ACM Trans. on Modeling and Computer Simulation Vol. 8, No. 1. Janeiro, 1998.

[19] PODLOZHNYUK, V. *Parallel Mersenne Twister*. Junho, 2007.