Simulação de Neurônios Biologicamente Realistas em GPUs *

Raphael Y. de Camargo
Centro de Matemática, Computação e Cognição
Universidade Federal do ABC – Santo André-SP, Brasil
raphael.camargo@ufabc.edu.br

Resumo

GPUs modernas são compostas por centenas de processadores capazes de executar milhares de threads simultaneamente e estão sendo utilizadas com sucesso em diversas aplicações na área de computação de alto-desempenho. Neste trabalho apresentamos um simulador de modelos neuronais realistas capaz de simular centenas de milhares de neurônios por GPU. Resultados experimentais mostram que as simulações são executadas 5 vezes mais rápidas em GPUs, quando comparadas a CPUs modernas.

1. Introdução

Aglomerados de GPUs vêm se mostrando uma excelente alternativa na área de computação de alto-desempenho para aplicações que exigem um alto grau de paralelismo [13]. As GPUs modernas possuem centenas de núcleos, cada um deles bastante simples, mas que, quando utilizados em paralelo, geram um alto poder computacional. Devido ao relativo baixo custo de placas gráficas contendo GPUs, sua utilização é uma ótima alternativa para pesquisadores pertencentes a instituições com poucos recursos financeiros e com grande necessidade de recursos computacionais.

Uma área que requer um grande poder computacional é a neurociência computacional, que tem como objetivo ajudar no entendimento do funcionamento do sistema nervoso através de simulações detalhadas de suas partes. Nas simulações, cada neurônio é modelado por equações diferenciais (de algumas dezenas a milhares por neurônio), que descrevem a dinâmica de sua membrana celular e de seus canais iônicos e cada simulação pode conter até milhões de neurônios [2, 9]. O maior problema é o poder computacional requerido [7]. Simulações maiores estão sendo realizadas em supercomputadores, como o IBM BlueGene. Mas o custo é inviável para pesquisadores de instituições que não dispõem de milhões de dólares para a compra e manutenção destes supercomputadores.

A restrição no uso de GPUs é que estas são otimizadas para operações de ponto flutuante do tipo SIMT (Single-Instruction Multiple-Thread), onde múltiplas threads executam uma mesma instrução em paralelo. A integração numérica de um grande número de equações diferenciais, que se diferenciam pelos valores de seus parâmetros, é um tipo de aplicação adequada para a execução em GPUs. Estas integrações podem ser realizadas em paralelo nos núcleos presentes nas GPUs, enquanto os demais aspectos da simulação, como sincronização, comunicação entre neurônios e coleta de resultados, são realizadas pela CPU.

Neste artigo, analisamos a viabilidade de utilizar o poder computacional de placas gráficas para a simulação da atividade de centenas de milhares de neurônios simultaneamente. Ainda não existem trabalhos publicados na área de simulação de neurônios biologicamente realistas em GPUs e neste sentido este trabalho é original. As principais contribuições do artigo são:
(1) Desenvolvimento de um simulador baseado em GPUs para redes neurais biologicamente realistas;
(2) Avaliação experimental do desempenho, escalabilidade e precisão da simulação em múltiplas GPUs.

2. Trabalhos relacionados

Muitas classes de aplicações do tipo SIMT já se beneficiam do uso de GPUs, como no treinamento de Support Vector Machines (SVM) [5]. Neste caso, o algoritmo aplica uma função sobre os elementos de uma matriz. Como a memória compartilhada de cada processador da GPU é pequena, os autores aplicam uma política de manter duas linhas de cada vez na memória compartilhada. Li *et al.* [11] simularam sistemas estocásticos de reações químicas, que são calculadas utilizando um algoritmo de Monte Carlo, sendo o principal desafio a geração de um grande quantidade de números aleatórios. Ambos os problemas acima podem ser considerados embaraçosamente paralelizáveis.

Existem diversos trabalhos sobre o uso de GPUs na área de simulação de redes neurais artificiais [3, 8, 10]. Es-

*Financiamento CNPq, processo #550895/2007-8.

tes trabalhos foram implementados utilizando bibliotecas gráficas anteriores ao CUDA, de modo que os elementos da simulação foram mapeados em textura e as operações sobre os elementos mapeadas em operações geométricas. Nestas redes, os neurônios são bastantes simples e modelados por uma função não-linear. Bernhard *et al.* [3] simularam redes de neurônios do tipo *integrate-and-fire*, que são modelos simplicados, representados por uma única equação diferencial, que possuem algumas propriedades semelhantes aos neurônios biológicos. Em todos estes casos, cada neurônio é bastante simples e pode ser facilmente implementado em um thread da GPU, com seus dados alocados na memória compartilhada dos processadores da GPU.

3. Arquitetura e programação de GPUs

GPUs (Graphics Processing Unit) são unidades de processamento presentes em placas gráficas modernas. Cada GPU é composta por dezenas de multiprocessadores, que por sua vez possuem múltiplos processadores, permitindo o processamento de um grande número de instruções em paralelo [13]. A Figura 1 mostra a arquitetura GT200, onde cada multiprocessador possui uma pequena memória local de alta velocidade, que é compartilhada entre os processadores, e um grande conjunto de registradores. Além disso, a placa gráfica possui uma memória global, que pode ser utilizada por todos os multiprocessadores. Como exemplo, as placas GTX 295 possuem duas GPUs, com 240 processadores distribuídos em 30 multiprocessadores por GPU, 1982MB de memória global e 8192 registradores e 16kB de memória compartilhada por multiprocessador.

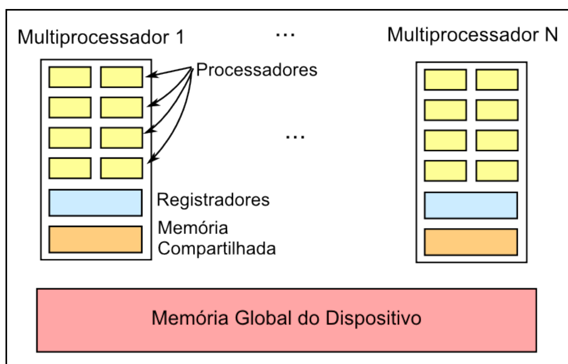

Figura 1. Arquitetura de GPUs modernas.

A programação de GPUs é feita utilizando plataformas que permitam acessar seus recursos computacionais. A mais utilizada atualmente é CUDA [12], uma extensão da linguagem C. Um programa em CUDA é iniciado e executado na CPU, mas o programador pode definir funções especiais, chamadas de *kernels*, que são executadas na GPU.

Para cada *kernel* o usuário precisa definir o número de threads que deseja executar e dividir estas threads em blocos. Vemos na Figura 2 que cada bloco é executado em um único multiprocessador, de modo que para utilizar todos os n multiprocessadores de uma GPU, é necessário criar no mínimo n blocos. Além disso, cada multiprocessador executa w threads de cada bloco simultaneamente, utilizando o modelo SIMT, onde todas as threads devem executar a mesma instrução. Consequentemente, cada *kernel* lançado para execução na GPU precisa ter no mínimo $n * w$ threads para ser executado de modo eficiente, por exemplo, $30 * 32 = 960$ por GPU em uma GTX 295.

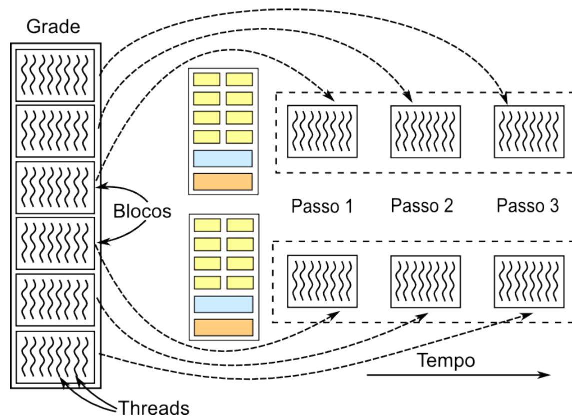

Figura 2. Execução dos blocos em um kernel.

O desafio de desenvolver programas eficientes em CUDA consiste em codificar a aplicação em um grande número de threads capazes de manter os processadores de cada GPU sempre ocupados. Como a latência de acesso à memória global é muito alta, cada thread deve manter os dados que estão sendo utilizados na pequena quantidade de memória compartilhada disponível por processador. Como veremos na Seção 5.1, estes requisitos são conflitantes e é necessário encontrar um balanço entre eles.

4. Simulação de modelos realistas de neurônios

Neurônios são células especializadas, que possuem uma membrana polarizada que mantém uma diferença de potencial da ordem de 60mV entre os meios interno e externo, com o processamento de informações ocorrendo através de alterações neste potencial. Para permitir que a dinâmica do neurônio seja simulada de modo eficiente, os neurônios são modelados como um conjunto de compartimentos isopotenciais, conectados por uma resistência radial entre os compartimentos. Cada neurônio é modelado como um circuito elétrico, onde a membrana celular é representada por um capacitor e os canais iônicos, que são proteínas que permitem a passagem de íons pela membrana, como resistências [2,9], como na Figura 3.

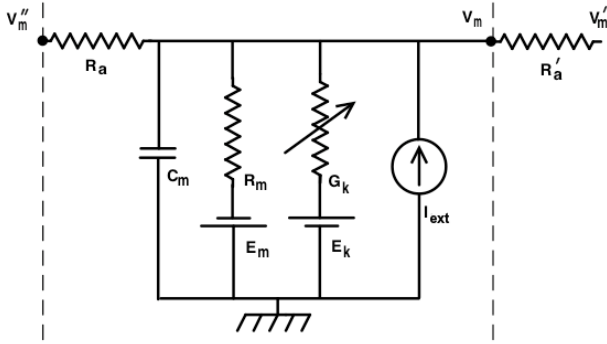

Figura 3. Compartimento de um neurônio modelado como um circuito elétrico.

A membrana possui canais iônicos passivos, de resistência constante, canais ativos dependentes de voltagem e canais sinápticos. Os canais sinápticos são o principal mecanismo de comunicação entre neurônios. Quando o potencial de membrana em um neurônio atinge um limiar, é disparado um potencial de ação [2,9], que causa a liberação de neurotransmissores na sinapse entre os neurônios. O tempo entre o disparo de um potencial de ação e a ativação do canal sináptico no outro neurônio demora em torno de 10ms. Já os canais ativos possuem um conjunto de portões, que permitem a passagem de íons, cuja dinâmica de abertura e fechamento depende do potencial de membrana e é representada por uma equação diferencial. Os canais ativos são os responsáveis pela geração dos potenciais de ação.

Cada neurônio é modelado por um sistema de equações diferenciais, uma por compartimento, que precisam ser integradas simultaneamente pelo simulador, e são do tipo:

$$C_m \frac{dV}{dt} = \frac{E_m - V_m}{R_m} + \frac{V'_m - V_m}{R'_a} + \frac{V''_m - V_m}{R_a} + I_{ion} + I_{ext}$$

onde V_m é o potencial de membrana, C_m a capacitância da membrana, R_m a resistência de membrana, R_a a resistência axial, I_{ext} é a corrente externa aplicada sobre o neurônio e I_{ion} é a soma das correntes dos canais sinápticos e ativos. V'_m, V''_m e R'_a correspondem aos compartimentos vizinhos.

Os simuladores neuronais mais utilizados são o GENESIS (GEneral NEural SImulation System) [2] e o Neuron [4]. Ambos permitem a simulação de neurônios individuais e de redes compostas por múltiplos neurônios e utilizam o método de Hines [6], descrito a seguir, para a integração das equações diferenciais. O GENESIS está atualmente na versão 2.3 e o MOOSE [1], em ativo desenvolvimento, deverá ser utilizado como núcleo do GENESIS 3.

4.1. O método de Hines

A simulação de neurônios é normalmente implementada utilizando o método de Hines [6], que permite integrar o sistema de N equações diferencias acopladas de um neurônio de modo bastante eficiente. As equações diferenciais que representam o neurônio são codificadas como um sistema linear do tipo $A * V = B$, composto por uma matriz esparsa A, contendo os coeficientes dependentes dos valores do potencial de membrana de cada compartimento (vetor V), e pelo vetor B, contendo os termos não dependentes dos potenciais. A cada passo de integração o sistema linear é resolvido e a simulação avança um intervalo de tempo dt.

A primeira otimização do método consiste em criar uma matriz A de um modo que esta possa ser triangularizada. O sistema pode então ser facilmente resolvido pelo uso do método da substituição, onde o valor V_k do último compartimento é calculado, em seguida o V_{k-1}, e assim por diante. A Figura 4 mostra este processo.

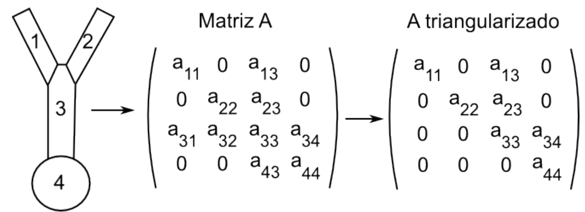

Figura 4. Matriz utilizada no método de Hines.

As demais otimizações são específicas para os canais ativos e consistem, por exemplo, em inserir na matriz A os coeficientes das equações diferenciais que representam os portões dos canais ativos. Deste modo, é possível se obter precisão de segunda ordem no cálculo das correntes dos canais ativos. No caso geral onde existem canais iônicos ativos em compartimentos arbitrários é necessário realizar a triangularização da matriz A em todos os passos de integração, uma vez que seus coeficientes terão os valores alterados em cada passo. Mas quando os canais ativos estão presentes apenas na soma, o que é verdade em grande parte dos modelos computacionais existentes, apenas o coeficiente do potencial V_k é alterado, e a triangularização não precisa ser realizada em cada passo.

5. Implementação do simulador

Realizamos a implementação do simulador em versões para a CPU e para a GPU, com o objetivo de comparar o ganho de desempenho obtido utilizando a GPU e as diferenças na precisão dos cálculos. Ambas as versões funcionam com números de ponto flutuante de precisão simples e dupla.

A implementação na CPU foi realizada em C++ e consiste de uma classe HinesMatrix que implementa o método de Hines. A matriz A é criada a partir de informações morfológicas dos neurônios que serão simulados e é, em seguida, triangularizada. O sistema linear é então resolvido

em cada passo de integração, que é dividido em três etapas: (1) determinação das correntes geradas pelos canais iônicos ativos, (2) determinação dos termos do vetor B, que são independentes dos potenciais de V e (3) atualização dos termos dependentes de V em A. Os N neurônios de cada simulação são simulados simultaneamente e os valores dos potenciais de membrana ao final cada passo de integração escritos em um arquivo de saída.

5.1. Implementação em GPU

A execução da simulação em GPU consiste em uma sequência de etapas coordenadas pela CPU, onde em cada etapa a CPU prepara a chamada do *kernel*, que implementa o método de Hines, solicita a execução do *kernel* na GPU e coleta os resultados da execução do *kernel*.

As matrizes que representam o neurônio são configuradas pela CPU no início da simulação. Em seguida é alocada memória no dispositivo (placa gráfica) e as informações dos neurônios são transferidas da memória principal do computador para a memória do dispositivo. Chamadas às funções de alocação de memória e transferência de dados são operações caras e são minimizadas, com a alocação da memória para todos os neurônios sendo realizada em uma única chamada. A transferência dos dados da simulação é realizada com uma única chamada por neurônio.

Cada chamada do *kernel* executa na GPU n passos de integração numérica, onde o valor de n é determinado por dois fatores: (1) em cada chamada do *kernel*, a GPU precisa armazenar em sua memória global dados da simulação, como o potencial de membrana nos compartimentos, que são transferidos para a memória principal do computador no fim da chamada e (2) a comunicação entre neurônios é processada pela CPU ao fim de cada chamada do *kernel*. Se o atraso na comunicação entre neurônios na rede simulada for, por exemplo, sempre maior que 10ms e utilizarmos $dt = 0.1ms$, podemos escolher $n = 100$.

Em nossa implementação, cada neurônio é simulado por uma thread distinta da GPU, com 32 threads por bloco, de modo a otimizar o uso dos processadores da GPU. A Figura 5 mostra a execução simultânea de 3 *kernels* em 3 GPUs, onde cada *kernel* precisa ser iniciado por uma thread distinta na CPU. No final de cada execução do *kernel*, a thread responsável pela execução coleta os dados da GPU para a memória da CPU, sincroniza a execução da simulação com as demais threads e, em seguida, inicia outra execução do *kernel* na GPU.

O *kernel* implementa o método de Hines, utilizando os processadores da GPU para realizar a solução do sistema linear. Esta implementação pode ser feita de forma direta, com os processadores acessando as informações dos neurônios diretamente na memória global, com poucas modificações com relação à versão em CPU. Mas na

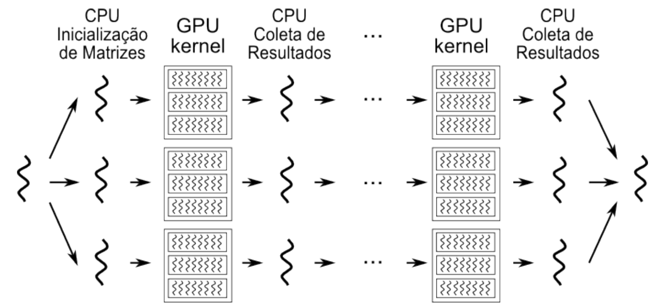

Figura 5. Execução da simulação em 4 GPUs.

implementação direta o desempenho é ruim, ficando até mesmo inferior ao desempenho da execução em CPUs, como veremos na Seção 6.5. Isto ocorre porque o acesso à memória global é lento, demorando centenas de ciclos computacionais do processador [12]. Além disso, o *cache* presente na GPU é específico para dados do tipo somente leitura. Consequentemente, nesta implementação direta, os processadores da GPU ficam a maior parte do tempo ociosos esperando pelos dados da memória global.

5.2. Otimizações

As otimização que realizamos no código para melhorar o desempenho da simulação consistem em diminuir o número de acessos à memória global, pois este foi o grande gargalo encontrado na implementação direta. Para tal, cada chamada do *kernel* realiza os seguintes passos: (1) transfere a maior quantidade possível de dados utilizados na simulação para a memória compartilhada dos multiprocessadores, (2) realiza os passos de integração numérica utilizando os dados da memória compartilhada e (3) copia de volta à memória global os dados que foram modificados.

A maior dificuldade é que a memória compartilhada disponível é muito pequena, 16 kB para cada multiprocessador, e são executados 32 neurônios por bloco, um em cada thread. Deste modo, a memória compartilhada disponível por neurônio é de apenas 512 bytes, o que impossibilita a alocação de todas as informações de cada neurônio na memória compartilhada.

Para otimizar o uso da memória compartilhada, reduzimos a quantidade de dados utilizada por neurônio: (1) a matriz esparsa A foi implementada como um arranjo linear contendo os termos não-nulos da matriz, diminuindo o uso de memória do neurônio de $O(k^2)$ para $O(k)$, onde k é o número de compartimentos, e (2) neurônios de um mesmo tipo compartilham diversas estruturas de dados, como a matriz A e informações morfológicas, de modo que basta uma única cópia desta informação na memória compartilhada de cada multiprocessador.

Finalmente, realizamos otimizações nos acessos à memória, de modo que estes pudessem ser combinados

de forma a reduzir a latência. No caso da memória global, quando diversas threads de um mesmo bloco acessam endereços contíguos de memória, o hardware da GPU agrupa os acessos em uma única requisição, processo denominado *coalescing*. Além disso, a memória compartilhada dos multiprocessadores é dividida em bancos de memória, e o acesso é mais eficiente quando não existem acessos simultâneos a um mesmo banco.

5.3. Limitações do simulador

Uma limitação importante de nosso simulador é que ainda não foram implementados canais sinápticos e mecanismos de comunicação entre neurônios. Apesar de ser uma funcionalidade complexa, não deve ocorrer um grande impacto no desempenho da simulação, pois a coordenação da comunicação é realizada ao fim de cada chamada do *kernel*, que executa em torno de 100 passos de integração numérica.

Uma segunda limitação é que, nesta versão, os canais iônicos ativos podem estar presentes apenas na soma do neurônio. No caso geral de canais ativos em qualquer compartimento é preciso realizar a triangularização da matriz A em cada passo de integração. Isto irá aumentar o uso de processamento, possivelmente melhorando o ganho em desempenho obtido pelo uso de GPUs, mas aumentará também o uso de memória compartilhada.

O simulador foi implementado com o objetivo de verificar a viabilidade de realizar simulações de neurônios biologicamente realistas em GPUs e não possui todas as funcionalidades de um simulador neuronal completo, como o GENESIS ou o Neuron. Pretendemos adaptar o *kernel* para ser executado com o simulador MOOSE [1], que é uma implementação da arquitetura do GENESIS 3 [2]. Este simulador é implementado de modo modular, sendo que a integração numérica dos neurônios é realizada por um módulo denominado `hsolve`. O objetivo é modificar este módulo de modo a utilizar o poder computacional de GPUs.

6. Experimentos

Avaliamos o simulador de modo a validar a corretude do simulador e determinar o ganho de desempenho e escalabilidade com relação ao número e complexidade dos neurônios. Os experimentos foram realizados em um computador com processador Intel Core i7 920, de 2.66GHz, 6 GB de memória RAM e duas placas gráficas nVidia GTX 295, com duas GPUs e 1892 MB de memória em cada. Utilizamos o sistema operacional Ubuntu 9.04 de 64 bits, CUDA versão 2.2 e drivers da nVidia versão 185.18.14. Utilizamos o compilador g++, configurado para gerar código otimizado através da opção -O3.

6.1. Validação do Simulador

A validação do simulador foi realizada comparando os resultados obtidos no simulador, utilizando a CPU e números de precisão dupla, com aqueles obtidos utilizando-se o GENESIS 2.3. Escolhemos o GENESIS por este ser um simulador amplamente utilizado em pesquisas científicas, com dezenas de trabalhos publicados [1] a partir de resultados obtidos no simulador.

Avaliamos diversos cenários, como a injeção de corrente em árvores dendríticas passivas, dendritos lineares passivos e em compartimentos com canais ativos. Em todos os casos, as simulações obtiveram resultados equivalentes, com diferenças no potencial de membrana menores que $0.1mV$. Estas diferenças são pequenas, dado que o potencial de membrana variou entre $-5mV$ e $90mV$, e possivelmente ocorram por pequenas diferenças de implementação.

6.2. Precisão dos cálculos

A arquitetura GT200 possui desempenho máximo quando utilizam representações de ponto-flutuante de precisão simples, fornecendo 8 unidades de processamento por multiprocessador, mas também dá suporte a representações de dupla precisão, mas com apenas 1 unidade de processamento por multiprocessador. Mas apesar da utilização de números de precisão simples trazer um desempenho teórico várias vezes superior, na prática isto não ocorre, pois a GPU não consegue manter todos os processadores sempre ativos.

Comparamos o desempenho da simulação para o caso em que utilizamos números e operações de ponto flutuante de precisão dupla e simples, tanto na execução na GPU quanto na CPU. O objetivo é verificar qual o impacto no tempo de execução gerado pelo uso de números de precisão dupla. Neste experimento utilizamos a versão otimizada da implementação em GPU e executamos as simulação utilizando um único núcleo de CPU e apenas uma GPU. Simulamos um período de 1 segundo em 10.000 neurônios.

A Figura 6 mostra que no caso de CPUs, à medida que aumentamos o número de compartimentos por neurônios, o tempo de execução cresce mais rapidamente com o uso de números do tipo double. Para neurônios de 4 compartimentos, o tempo de execução com doubles é 14% maior, enquanto com 32 compartimentos o tempo é 28% maior. Este efeito é provavelmente resultante da maior quantidade de faltas de *cache* quando usamos números do tipo double.

No caso de GPUs, a diferença relativa no tempo de execução com o uso de double e float diminui à medida que aumentamos o número de compartimentos. A diferença de desempenho no caso de 16 compartimentos é de 17%. Podemos concluir a partir destes resultados que o desempenho da simulação é fortemente dependente do tempo de

[1] Lista disponível em `http://www.genesis-sim.org`.

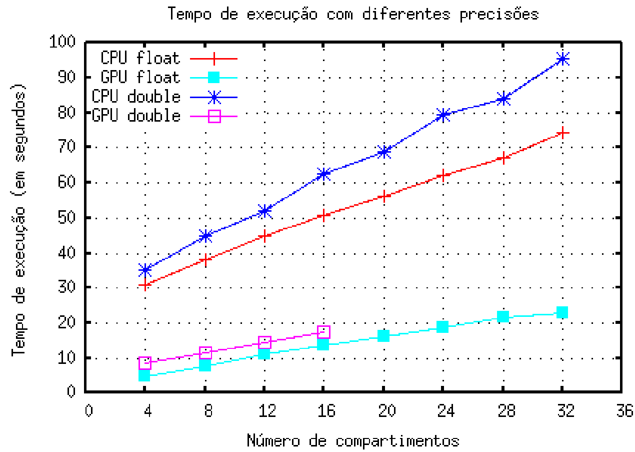

Figura 6. Execução com números de ponto-flutuante de diferentes precisões.

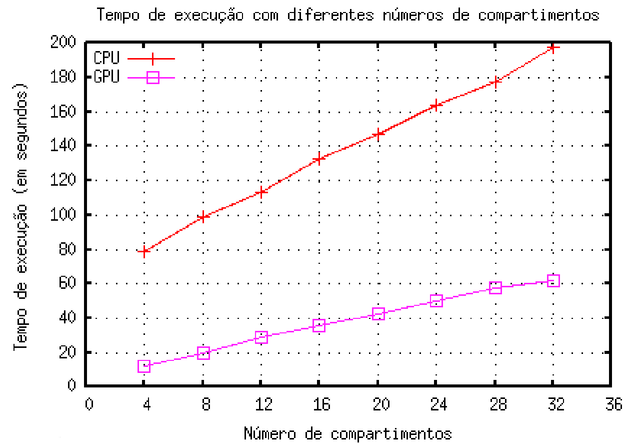

Figura 7. Tempo de execução com diferentes números de compartimentos.

acesso à memória da placa gráfica, caso contrário o desempenho com a precisão simples deveria ser pelo menos 4 vezes superior, aumentando proporcionalmente ao número de compartimentos. No caso de números de precisão dupla, tivemos que limitar o número de compartimentos a 16, em comparação a 32 no caso de precisão simples, devido ao tamanho reduzido da memória compartilhada.

Outra maneira de otimizar o desempenho seria utilizando funções matemáticas, fornecidas pela biblioteca CUDA, que possuem menor precisão mas são calculadas mais rapidamente. Mas a utilização destas funções trouxe um ganho de desempenho muito pequeno, menor que 0.5%, e diminuição na precisão dos resultados, de modo que decidimos não utilizá-las.

Também avaliamos os erros que ocorrem quando utilizamos números de precisão simples e dupla na CPU e GPU para determinar se existem diferenças significativas nos resultados. Realizamos a simulação de neurônios utilizando 4, 8, 12 e 16 compartimentos e avaliamos o erro médio em cada caso com relação à execução na CPU com números de precisão dupla.

Tabela 1. Diferença média no valor de V_m com relação à execução em CPU com números do tipo *double* após 1s de simulação.

GPU double	CPU float	GPU float
0.0002 mV	0.0063 mV	0.0352 mV

Como podemos ver na Tabela 1, o erro médio gerado pela utilização da GPU com o tipo double é praticamente zero. Quando utilizamos a GPU com o tipo float, o erro é maior mas ainda pequeno se considerarmos que o potencial de repouso dos neurônios é da ordem de -60mV. Deste modo, decidimos utilizar o tipo float como padrão nas simulações com GPUs, especialmente porque neste caso é possível simular neurônios de até 32 compartimentos.

6.3. Desempenho da simulação em GPUs

Comparamos o desempenho do simulador baseado em GPUs com relação à simulação em CPUs, para verificar se o ganho em desempenho justifica o seu uso. Avaliamos o desempenho do simulador utilizando 4 GPUs e 4 núcleos de um processador quad-core, comparando o tempo necessário para a simulação de 1 segundo de atividade neuronal. Neste, e em todas os experimentos seguintes, utilizamos números de precisão simples do tipo float tanto nas simulações com CPUs quanto com GPUs.

O número máximo de neurônios que podem ser executados simultaneamente em cada GPU é determinado pela disponibilidade de memória na placa gráfica, sendo que em nosso caso conseguimos armazenar os dados de 100.000 neurônios de 32 compartimentos ou 800.000 de 4 compartimentos por GPU. Seria possível simular mais neurônios em cada GPU, mas nesse caso, teríamos que constantemente trocar os dados armazenados na GPU para simular diferentes grupos de neurônios, o que seria proibitivamente caro. Finalmente, utilizamos um único tipo de neurônio na simulação para diminuir a quantidade de variáveis na simulação, mas múltiplos tipos podem ser facilmente simulados, com a restrição de que os neurônios de um mesmo bloco do *kernel* devem ser do mesmo tipo.

No primeiro experimento avaliamos o tempo necessário para realizar a simulação de 1 segundo de atividade neuronal de 100.000 neurônios. Utilizamos neurônios com diferentes números de compartimentos e comparamos o tempo total utilizado pela GPU com o tempo utilizado pela CPU.

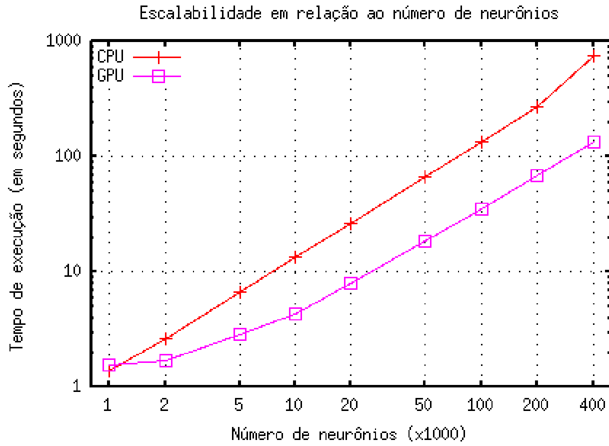

Figura 8. Tempo de execução com diferentes números de neurônios.

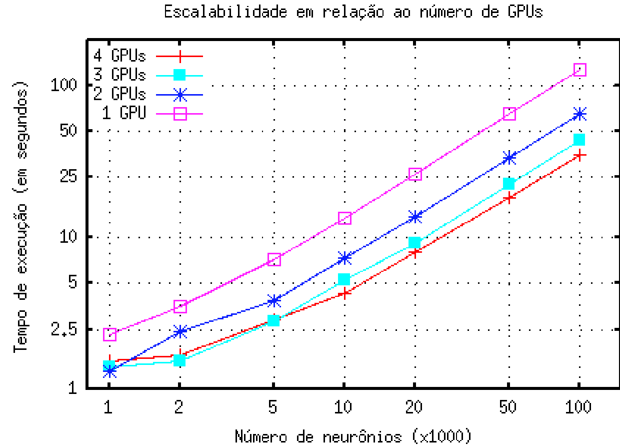

Figura 9. Escalabilidade no número de GPUs.

Vemos na Figura 7 que a simulação utilizando 4 GPUs possui um desempenho 6.3 vezes superior ao uso de 4 CPUs (12.50s e 78.98s respectivamente) quando simulamos neurônios com 4 compartimentos. No caso de neurônios de 32 compartimentos, esta melhora cai para 3.2 vezes (61.65s e 197.38s respectivamente). Este efeito ocorre porque no caso de neurônios de 16 compartimentos é necessário buscar uma maior quantidade de dados na memória global da placa gráfica, reforçando o fato de que o desempenho da simulação na GPU está sendo limitado pela latência de acesso à memória global.

Avaliamos também a variação do tempo necessário para a execução da simulação à medida que aumentamos o número de neurônios simulados. Realizamos a simulação utilizando entre 1.000 e 400.000 neurônios.

A Figura 8, em escala log-log, mostra que, para simulações com 400.000 neurônios, o tempo gasto quando utilizamos 4 GPUs foi de 137.20 segundos, enquanto com 4 CPUs o tempo foi de 752.19s, um ganho de desempenho de 5.5 vezes. Para 10.000 neurônios, o ganho foi de 3.1 vezes e para 1.000 neurônios a execução em CPUs foi mais rápida que em GPUs. Isto mostra que utilizar GPUs é vantajoso apenas no caso em que precisamos simular um grande número de neurônios, o que é esperado, dado que o alto desempenho das GPUs vêm do fato destas poderem processar um grande número de threads simultaneamente. Além disso, no caso de poucos neurônios, não existe justifica para utilizar GPUs, dado que as CPUs são capazes de simular os neurônios de modo eficiente.

6.4. Escalabilidade com o número de GPUs

Comparamos o ganho de desempenho obtido com o uso de múltiplas GPUs, simulando a execução de diferentes números de neurônios de 16 compartimentos, utilizando 1, 2, 3 e 4 GPUs. A Figura 9 mostra o tempo total de execução em cada configuração, sendo que o número total de neurônios é dividido entre as GPUs disponíveis.

Como podemos ver, no caso em que simulamos um grande número de neurônios o ganho de desempenho foi proporcional ao número de GPUs utilizadas. Para 100.000 neurônios, o tempo de execução quando utilizamos um núcleo foi de 129.67s, tempo 1.98 vezes superior à simulação com 2 GPUs, 2.93 vezes superior à 3 GPUs e 3.64 vezes superior à simulação com 4 GPUs.

O ganho para o uso de até 3 GPUs foi praticamente linear, com as diferenças geradas por: (1) GPUs são mais eficientes na simulação simultânea de um número maior de neurônios e (2) o tempo necessário para preparar a simulação é independente do número de GPUs. No caso de 4 GPUs, o ganho de desempenho foi menor porque tivemos que compartilhar a GPU com a execução da interface gráfica do Linux. Mas, no caso de máquinas dedicadas à execução de aplicações paralelas e em simulações mais longas, esta influência será menor e podemos esperar *speedups* próximos de linear.

6.5. Influência das otimizações

Neste experimento comparamos o efeito que as diferentes otimizações produziram no tempo de execução da simulação. Comparamos o tempo de execução para a simulação de 10.000 neurônios utilizando uma única GPU para diferentes números de compartimentos. Utilizamos as duas versões do simulador descritas na Seção 5.1.

Como podemos verificar na Figura 10, quando utilizamos a versão não otimizada do código, o tempo necessário para realizar as simulações cresce rapidamente à medida que aumentamos o número de compartimentos. Para 4 compartimentos, foram necessários 10.32 segundos para executar a simulação, enquanto para 32 compartimentos foram

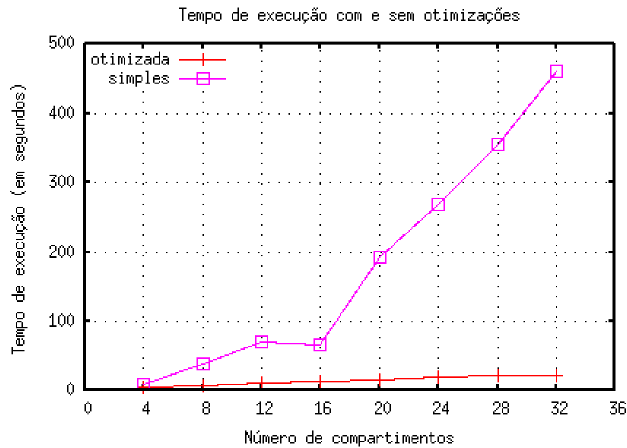

Figura 10. Impacto das otimizações.

utilizados 459.74 segundos, um tempo 45 vezes superior para um número de compartimentos 8 vezes maior.

Na versão otimizada, foram utilizado 4.87s na simulação com 4 compartimentos e 23.17 na simulação com 32 compartimentos, com um crescimento linear no tempo total de execução à medida que aumentamos o número de compartimentos. Para neurônios com 32 compartimentos, o tempo de execução foi aproximadamente mais de 20 vezes mais rápido que o caso não otimizado.

Esta diferença de desempenho pode ser explicada pelo fato que o grande limitador de desempenho nas placas gráficas atuais é a latência no acesso à memória global da placa gráfica, que é da ordem de centenas de ciclos. No caso otimizado os dados mais utilizados são colocados na memória compartilhada dos processadores da GPU, de modo que o tempo de acesso é muito menor, além das demais otimizações, como a transferência conjunta dos dados de diferentes neurônios em uma única requisição.

7. Conclusões

Neste artigo mostramos que a simulação de redes neuronais biologicamente realistas em GPUs é viável e produz importantes ganhos de desempenho, superiores a 5 vezes na simulação de neurônios de 16 compartimentos em uma máquina com 4 GPUs. Considerando que diversas placas mãe modernas oferecem a possibilidade de colocar 4 placas gráficas (8 GPUs) em um único computador, para esta máquinas podemos esperar obter um ganho superior a 10 vezes com o uso de GPUs e a possibilidade de simular 800.000 neurônios por máquina. Deste modo, com um pequeno aglomerado de GPUs com 5 máquinas, teríamos o poder computacional de um aglomerado tradicional de 50 máquinas e capacidade de simular 4 milhões de neurônios simultaneamente.

Os ganhos de desempenhos devem ser ainda mais expressivos na próxima geração de placas gráficas, dado que o poder de processamento em GPUs está aumentando muito mais rapidamente que o de CPUs [12]. Além disso, pretendemos aplicar otimizações no código que podem melhorar ainda mais o desempenho de nosso simulador, como o uso do cache de texturas da GPU.

Como trabalho em andamento, estamos desenvolvendo o simulador para eliminar as restrições existentes relacionadas à comunicação entre neurônios através de sinapses e a presença de canais iônicos ativos fora do corpo celular. O passo seguinte será incorporar a solução do método de Hines em GPUs ao simulador MOOSE. Isto permitirá que pesquisadores executem centenas de modelos de redes neuronais já existentes e facilmente criem novos modelos utilizando as ferramentas deste simulador, o que facilitará o acesso ao alto poder computacional oferecido pelas GPUs a pesquisadores não especialistas em programação paralela.

Referências

[1] U. S. Bhalla. MOOSE (multiscale object-oriented simulation environment) site. http://moose.sourceforge.net.

[2] J. M. Bower and D. Beeman. *The Book of GENESIS: Exploring Realistic Neural Models with the GEneral NEural SImulation System*. Springer-Verlag, second edition, 1998.

[3] A. Brandstetter and A. Artusi. Radial basis function networks GPU-based implementation. *IEEE Transactions on Neural Networks*, 19(12):2150–2154, 2008.

[4] N. Carnevale and M. Hines. *The NEURON Book*. Cambridge University Press, 2006.

[5] B. Catanzaro, N. Sundaram, and K. Keutzer. Fast support vector machine training and classification on graphics processors. In *ICML '08: Proc. of the 25th Int. Conf. on Machine Learning*, pages 104–111. ACM Press, 2008.

[6] M. Hines. Efficient computation of branched nerve equations. *International Journal Biomedical Computation*, 15(1):69–76, January 1984.

[7] M. Hines and N. Carnevale. Translating network models to parallel hardware in NEURON. *Journal of Neuroscience Methods*, 169(2):425–455, April 2008.

[8] T.-Y. Ho, P.-M. Lam, and C.-S. Leung. Parallelization of celular neural networks on gpu. *Pattern Recognition*, 41:2684–2692, 2008.

[9] C. Koch and I. Segev, editors. *Methods in Neuronal Modeling: From Ions to Networks*. MIT Press, 2nd edition, 1999.

[10] Kyoung-Su and K. Jung. Gpu implementation of neural networks. *Pattern Recognition*, 37:1311–1314, 2004.

[11] H. Li and L. Petzold. Efficient parallelization of the stochastic simulation algorithm for chemically reacting systems on the GPU. *International Journal of High Performance Applications*, To appear, 2009.

[12] nVidia Corporation. *CUDA 2.1 Programming Guide*, 2009.

[13] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.