4
Descrição das Alterações para Aceleração do SIESTA por GPU

Inicialmente avaliou-se a possibilidade de trabalhar com os principais algoritmos de código fonte aberto empregados na química quântica. Entre eles esta o MPQC (*Massively Parallel Quantum Chemistry Program*), que emprega métodos tais como o Hartree-Fock e a Teoria do Funcional da Densidade (Janssen, Nielsen, *et al.*, 2010). Outro pacote considerado foi o Quantum ESPRESSO (*opEn-Source Package for Research in Electronic Structure, Simulation, and Optimization*), que possui cálculos de estrutura eletrônica realizados com a Teoria do Funcional da Densidade, empregando ondas planas e pseudopotenciais (DEMOCRITOS, 2009). O YAeHMOP (*Yet Another extended Huckel Molecular Orbital Package*), que é um método *tight-binding* usado para obter uma rápida descrição qualitativa (Landrum e Glassey, 2001). O escolhido para utilização no trabalho foi o SIESTA (*Spanish Initiative for Electronic Simulations with Thousands of Atoms*), o qual usa DFT, pseudopotenciais, orbitais atômicos numéricos e funções de base localizada para aumentar a sua eficiência (Soler, Artacho, *et al.*, 2002). Este programa empregado na química quântica possui 281 arquivos, totalizando 142023 linhas de código fonte em Fortran. Devido a sua complexidade, apenas partes com maior custo computacional serão portadas para execução em GPU.

Conforme o descrito no item 2.7, o SIESTA possui uma maneira particular de representar os termos da energia total. Uma densidade atômica é definida para reescrever os termos do pseudopotencial local e do potencial de Hartree. Isto é feito com o objetivo de eliminar os termos de longo alcance, aumentando assim a eficiência computacional. As funções de bases atômicas localizadas, usadas no SIESTA foram escolhidas para viabilizar o desenvolvimento de um método *ab initio* DFT autoconsistente de ordem N. Pois, o emprego de ondas planas como funções de base impossibilitaria a determinação do hamiltoniano autoconsistente

em $O(N)$ operações. Por outro lado, funções tais como a Transformada de Fourier ainda necessitam de ordem N log(N) (Soler, Artacho, *et al.*, 2002).

Neste Capítulo, as partes do código SIESTA com potencial para execução em GPU são listadas e são apresentadas algumas alterações realizadas para testes e comprovação de desempenho.

4.1. Identificação das Partes Adequadas ao Paralelismo de Dados

O termo da eq. (2.42), visto no Capitulo 2, relacionado às contribuições para a energia total da parte não-local do pseudopotencial e o termo de cálculo da energia cinética dos elétrons, são operações repetitivas sobre dados diferentes e poderiam ser estudados para execução acelerada por GPU. Uma matriz de densidade é usada para as demais integrais da eq. (2.42), como a energia de troca e correlação, o potencial e a energia de Hartree e a parte local do pseudopotencial do hamiltoniano do SIESTA.

Inicialmente foi estuda a melhor forma de se reescrever em CUDA, os termos referentes ao cálculo do potencial de Hartree e da energia de Hartree, com a resolução da Equação de Poisson na GPU, e a contribuição para as forças de deslocamento atômicas, devidas ao potencial de Hartree.

A próxima etapa será o estudo do termo da energia de troca e correlação e a parte local do pseudopotencial do hamiltoniano do SIESTA. Isto permitiria trabalhar a matriz de densidade na memória da GPU. Somente seriam necessárias transferências de memória para os parâmetros de inicialização e os resultados das integrais. É importante reduzir as transferências de memória entre CPU e GPU devido à latência da interface PCI Express.

4.1.1. CUFFT

A biblioteca CUFFT (*CUDA Fast Fourier Transform*) possui funções para a realização de Transformadas de Fourier, no sentido direto ou no sentido inverso, sobre matrizes de dados reais ou complexas com uma, duas ou três dimensões (NVIDIA, 2007b). Para emprego no SIESTA, são adequadas as funções apresentadas na Tabela 2.

Tabela 2 – Funções da Biblioteca CUFFT adequadas para emprego no SIESTA.

1	cufftExecR2C	Transformada de Fourier no sentido direto para matrizes tridimensionais.
2	cufftExecC2R	Transformada Inversa de Fourier para matrizes tridimensionais.

4.1.2. CUBLAS

A biblioteca CUBLAS é uma implementação das funções básicas de álgebra linear BLAS (*Basic Linear Algebra Subprograms*) para execução paralela, em GPUs, usando CUDA (NVIDIA, 2007a). A Tabela 3 lista e descreve brevemente algumas das funções CUBLAS que poderiam substituir funções de álgebra linear no código SIESTA.

Tabela 3 – Funções da Biblioteca CUBLAS com possibilidade de emprego para o SIESTA.

1	cublasDgemm	calcula o produto de duas matrizes A e B, multiplicando o resultado por um escalar *alpha*. A seguir soma com o produto de uma matriz C por um escalar *beta*.
2	cublasIsamax	encontra o máximo de um vetor. Se o máximo não for um ponto único, o ponto retornado é o que possui o menor índice.
3	cublasSasum	calcula a soma dos valores absolutos de um vetor.
4	cublasSaxpy	multiplica um vetor por um escalar.
5	cublasScopy	copia os elementos de um vetor x para um vetor y.
6	cublasSdot	calcula o produto escalar de dois vetores.
7	cublasSscal	multiplica um vetor x por um escalar *alpha*.
8	cublasCaxpy	multiplica um vetor de números complexos x por um escalar *alpha* e em seguida adiciona um segundo vetor de números complexos y.
9	cublasCscal	multiplica um vetor de números complexos x por um escalar *alpha*.
10	cublasCswap	troca os elementos de um vetor de números complexos x, com os elementos de outro vetor de números complexos y.

4.1.3. MAGMA

A biblioteca de álgebra linear MAGMA (*Matrix Algebra on GPU and Multicore Architectures*) foi projetada para ser similar ao LAPACK (*Linear Algebra PACKage*) em nível de funções suportadas (Tomov, Nath, *et al.*, 2009). Algumas das suas funções, listadas na Tabela 4, apresentam compatibilidade com funções empregadas no SIESTA.

Tabela 4 – Funções da Biblioteca MAGMA com possibilidade de emprego para o SIESTA.

1	magma_dpotrf_gpu	calcula a fatoração de Cholesky de uma matriz real simétrica positiva.
2	magma_zpotrf_gpu	calcula a fatoração de Cholesky de uma matriz Hermitiana complexa positiva.
3	magmablas_dtrsm	resolve equações matriciais em GPU, com matrizes X e B, m por n, triangulares superior ou inferior e sendo possível aplicar o operador de transposição à matriz A. $op(A) * X = alpha * B$ $X * op(A) = alpha * B$

A biblioteca MAGMA também possui outras funções de grande utilidade no meio científico, tais como um conjunto de funções CUDA para resolver sistemas lineares baseados nas decomposições LU, QR e Cholesky (Tomov, Nath, *et al.*, 2009).

4.1.4. CULA

Outra biblioteca de álgebra linear, acelerada por GPUs, com aplicação no meio científico é a CULA (EM Photonics, 2010). Algumas das principais funções disponibilizadas, para números reais e para números complexos, são listadas na Tabela 5.

Tabela 5 – Funções da Biblioteca CULA utilizadas no meio científico.

1	SGESV e CGESV	resolve um sistema geral de equações lineares AX=B.
2	SGETRF e CGETRF	calcula a fatoração LU de uma matriz geral.
3	SGEQRF e CGEQRF	calcula a fatoração QR de uma matriz retangular geral.
4	SGELS e CGELS	resolve um sistema de equações lineares por mínimos quadrados.
5	SGGLSE e CGGLSE	resolve um sistema linear com restrições por mínimos quadrados.
6	SGESVD e CGESVD	calcula a decomposição em valor singular de uma matriz retangular geral.

As funções de resolução de sistemas lineares das bibliotecas MAGMA e CULA poderiam ser empregadas para calcular a inversa de matrizes reais ou complexas, usadas no código SIESTA.

4.2. Alteração de Partes do SIESTA

A última versão estável do código fonte do SIESTA, originalmente escrito em Fortran, foi utilizado como ponto de partida para estudar e alterar trechos de código para GPU. O paralelismo de dados das GPUs é acessado através da linguagem CUDA, e a linguagem C é usada como uma interface entre Fortran e CUDA, conforme o ilustrado na Figura 25. O sistema operacional utilizado foi o Linux CentOS 5.5.

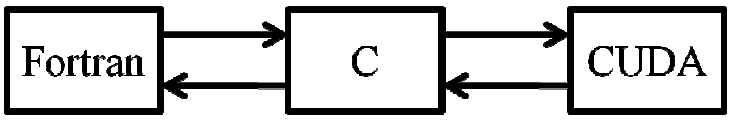

Figura 25 – Interface entre Fortran e CUDA.

4.2.1. Potencial de Hartree, Energia de Hartree e Forças de Deslocamento Atômicas

Inicialmente, o termo referente ao cálculo do potencial de Hartree foi reescrito usando a linguagem CUDA, empregando a resolução da Equação de Poisson na GPU. O quarto termo do lado direito da eq. (2.42), potencial de Hartree, $\delta V_H(r)$, é produzido pela diferença das densidades atômicas e auto-consistente $\delta\rho(r)$. A solução da equação de Poisson na forma original implementada no SIESTA permite encontrar o potencial de Hartree associado com a variação da densidade.

A Equação de Poisson é uma equação diferencial parcial muito utilizada na eletrostática, na engenharia e na física teórica. No espaço euclidiano esta equação é freqüentemente escrita como:

$$\nabla^2\varphi = \mathrm{f} \tag{4.1}$$

Considerando o sistema de coordenadas cartesianas por simplicidade, esta equação pode ser reescrita como:

$$\left(\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}+\frac{\partial^2}{\partial z^2}\right)\varphi(\mathrm{x},\mathrm{y},\mathrm{z}) = \mathrm{f}(\mathrm{x},\mathrm{y},\mathrm{z}) \tag{4.2}$$

Para problemas que envolvem condições periódicas de contorno, o uso da Transformada de Fourier permite transformar as equações diferenciais em equações algébricas, as quais são resolvidas com operações algébricas normais. Em seguida a transformada inversa é utilizada para trazer a solução novamente para o espaço real. A Figura 26 ilustra o emprego deste processo para a resolução de equações diferencias.

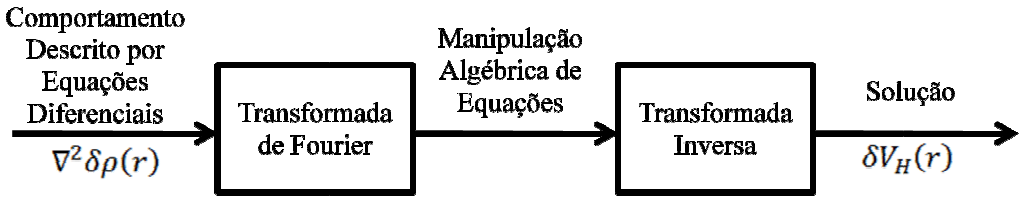

Figura 26 – As equações diferenciais são transformadas em equações algébricas.

A energia de Hartree e a contribuição para as forças de deslocamento atômicas, devido ao potencial de Hartree, estão combinadas na mesma seção do código da Equação de Poisson e também são computadas em GPU. Cada elemento da matriz de densidades $\delta\rho(r)$ contribui individualmente para a energia

de Hartree e para as forças de deslocamento atômicas. Assim, o cálculo da energia de Hartree e das forças de deslocamento atômicas em GPU envolve, para cada variável calculada, somatórios sobre todos os elementos da matriz de densidades.

O somatório paralelo realizado na memória compartilhada de uma GPU consiste na redução do vetor de dados pela metade em cada operação dos elementos do hardware paralelo. A Figura 27 ilustra a realização de um somatório paralelo em GPU para um único veto de dados.

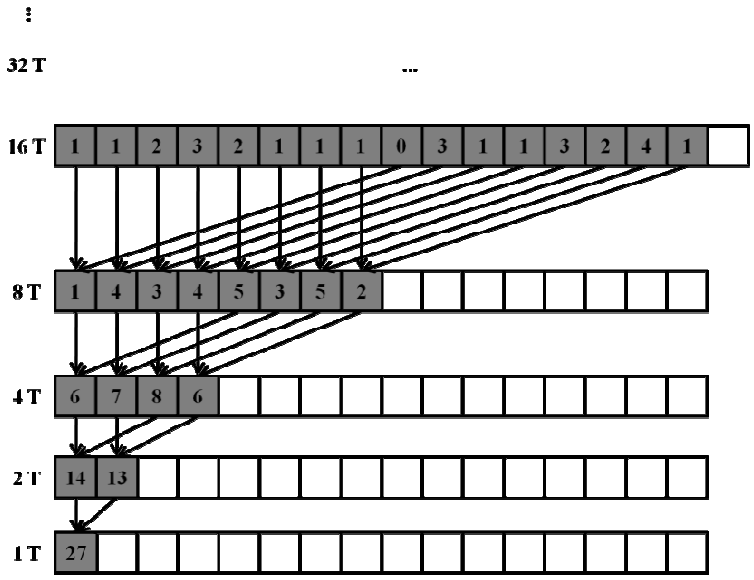

Figura 27 – Somatório paralelo realizado na memória compartilhada da GPU para um único vetor de dados.

4.2.2. Implementação Paralela para GPUs

A Equação de Poisson usada no SIESTA está baseada em condições periódicas de contorno e é resolvida no domínio da freqüência. Inicialmente são encontrados os componentes da Transformada de Fourier da densidade eletrônica $\delta\rho(r)$. A solução é realizada no domínio da freqüência e um somatório da contribuição de cada elemento da matriz de densidades produz a energia de Hartree e as forças de deslocamento atômicas. O potencial de Hartree, $\delta V_H(r)$, é em seguida transformado para o espaço real conforme o ilustrado na Figura 28.

A Transformada de Fourier originalmente empregada no código SIESTA é o algoritmo GPFA (*Generalized Prime Factor FFT*) desenvolvido por Temperton (1992). Este algoritmo era originalmente empregado para transformar a matriz de densidades $\delta\rho(r)$ do espaço real para o domínio da freqüência e para transformar

o potencial de Hartree, $\delta V_H(r)$, calculado no domínio da freqüência, para o espaço real.

Na Equação de Poisson para execução em GPU foi substituído o algoritmo GPFA pelo algoritmo CUFFT (*CUDA Fast Fourier Transform*) desenvolvido pela NVIDIA (2007b). Assim, a Transformada de Fourier da matriz de densidades $\delta\rho(r)$ e a Transformada Inversa de Fourier do potencial de Hartree, $\delta V_H(r)$, são realizadas na placa gráfica. Algoritmos GPU mais eficientes para a Transformada de Fourier poderiam contribuir para aumentar o desempenho desta etapa do código SIESTA, tal como o algoritmo proposto por Nukada, Ogata, *et al.* (2008).

Para o cálculo do potencial de Hartree, $\delta V_H(r)$, propriamente dito, foi escrito um *kernel* baseado na função seqüencial original do SIESTA. A primeira etapa consistiu na tradução do algoritmo da linguagem Fortran para a linguagem C. Em seguida estudou-se a melhor forma de tirar proveito do paralelismo de dados oferecido pela GPU. As funções de transferência de dados entre CPU e GPU foram introduzidas e os laços originais foram substituídos pelo paralelismo de hardware da GPU. Foram realizados testes para avaliar os efeitos das otimizações de acesso à memória global, de acesso à memória compartilhada e de transferência através do barramento PCI Express, conforme o descrito no Capítulo 3, nos itens 3.5.1, 3.5.2 e 3.5.3, respectivamente.

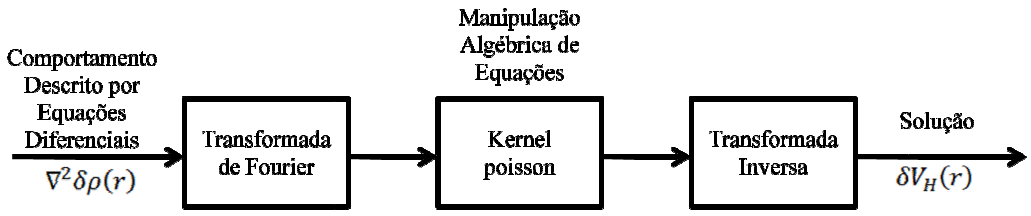

Figura 28 – Solução no domínio da freqüência.

A CPU tem uma unidade de instruções para cada núcleo de processamento e a utilização de 240 *threads* num *cluster* com 240 núcleos de processamento poderia ser adequada.

Nas GPUs o paradigma é diferente, pois a GPU possui somente uma unidade de instruções para cada 8 elementos de processamento (um multiprocessador), o tempo necessário para mudar de instrução é suficiente para aplicá-la sobre 4 *threads* de cada elemento de processamento. Ou seja, cada instrução carregada na unidade de instruções pode ser aplicada em 32 *threads*, um

warp, executando em 8 elementos de processamento. Assim, as *threads* de GPUs foram ser projetadas para serem muito leves.

Além disso, existem os tempos de latência no acesso à memória global, à memória compartilhada e aos registradores. Com 32 *threads* por bloco não existe a possibilidade de executar instruções em outros *warps*, quando ocorre uma instrução com grande latência. Assim, o multiprocessador fica ocioso esperando o dado ser lido ou escrito nas memórias. Por isso, a NVIDIA recomenda manter um mínimo de 192 *threads* por bloco para que o multiprocessador possa ter sempre instruções disponíveis, reduzindo o tempo ocioso do multiprocessador (NVIDIA, 2009b). Desta forma, um número mínimo de *threads* deveria ficar em torno de 192 *threads* / bloco x 30 multiprocessadores = 5760 *threads*. Contudo, melhor desempenho será obtido se o problema for pensado de forma a dividi-lo por um número maior de *threads* na GPU.

Na primeira versão do *kernel* para a resolução da Equação de Poisson do SIESTA na GPU foi atribuída uma *thread* para cada elemento da matriz de densidades $\delta\rho(r)$ (Figura 29). Neste caso, cada bloco com 256 *threads* utiliza 18 registradores e 3616 bytes de memória compartilhada. O Nível de Ocupação do Multiprocessador é de 75 %, conforme pode ser observado na Figura 30.

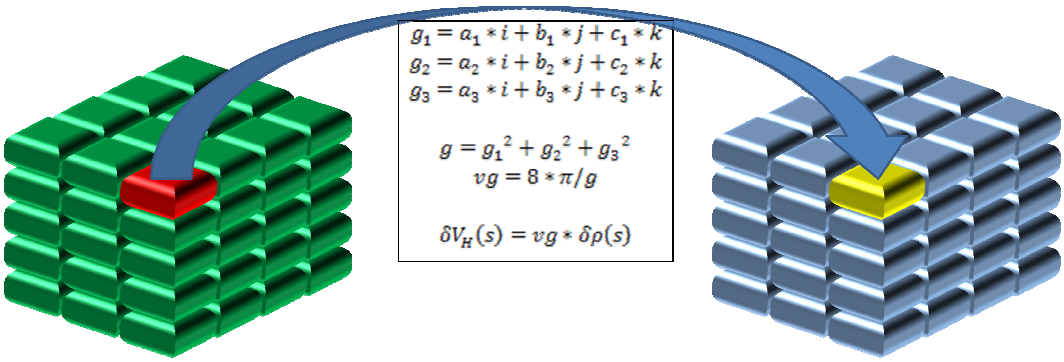

Figura 29 – Uma *thread* para cada elemento da matriz de densidades.

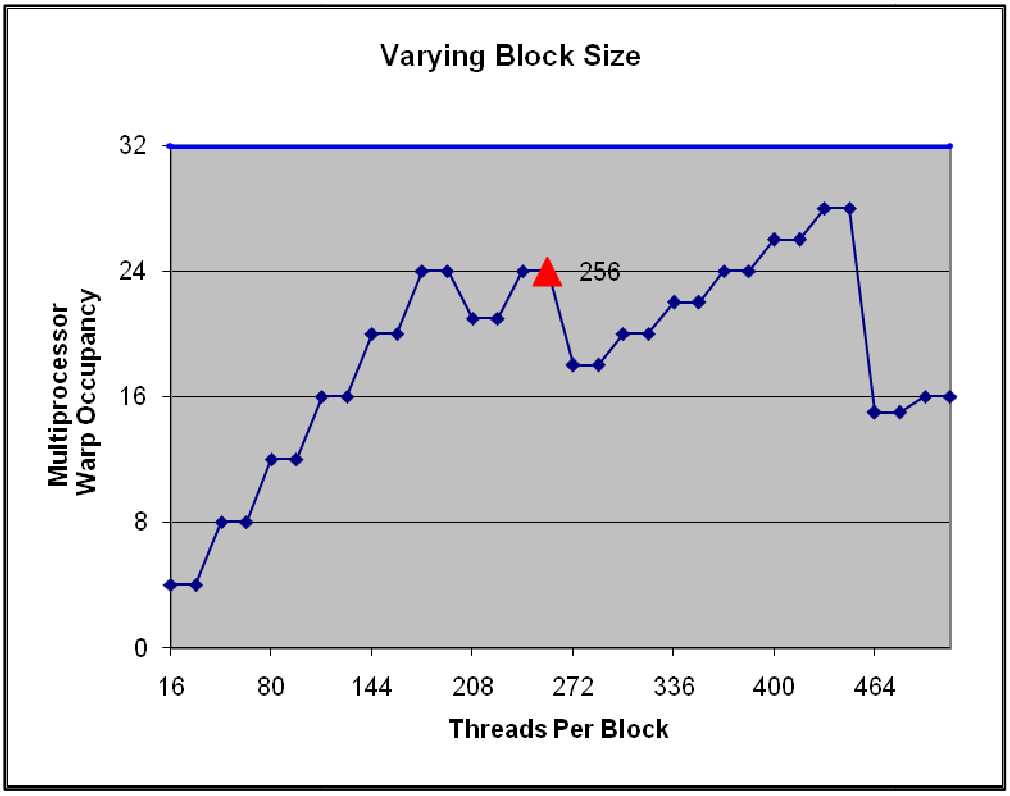

Figura 30 – Nível de Ocupação do Multiprocessador para primeira versão do *kernel* da Equação de Poisson.

Foi desenvolvida uma segunda versão do *kernel* para resolução da Equação de Poisson na GPU, onde cada *thread* processa um vetor de dados da matriz de densidades $\delta\rho(r)$ (Figura 31). Cada bloco com 256 *threads* utiliza 28 registradores e 7200 bytes de memória compartilhada. A intensidade aritmética de cada *thread* é maior, por outro lado o Nível de Ocupação do Multiprocessador é reduzido para 50 %, conforme o apresentado na Figura 32.

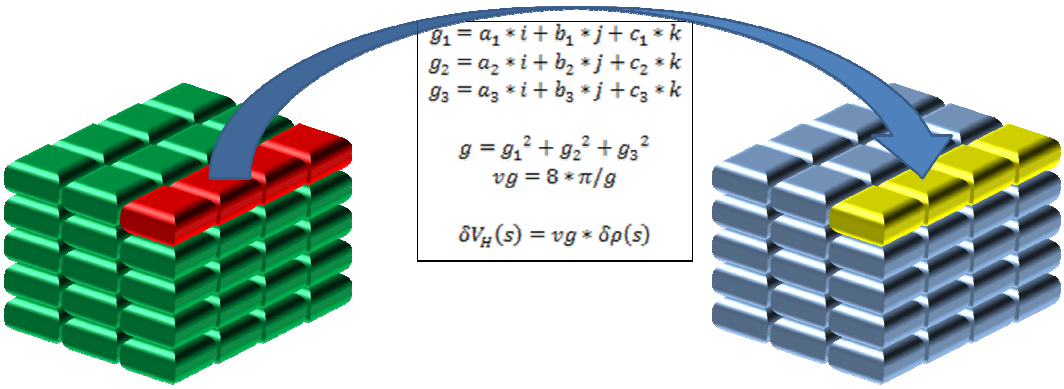

Figura 31 – Uma *thread* processa um vetor de dados da matriz de densidades.

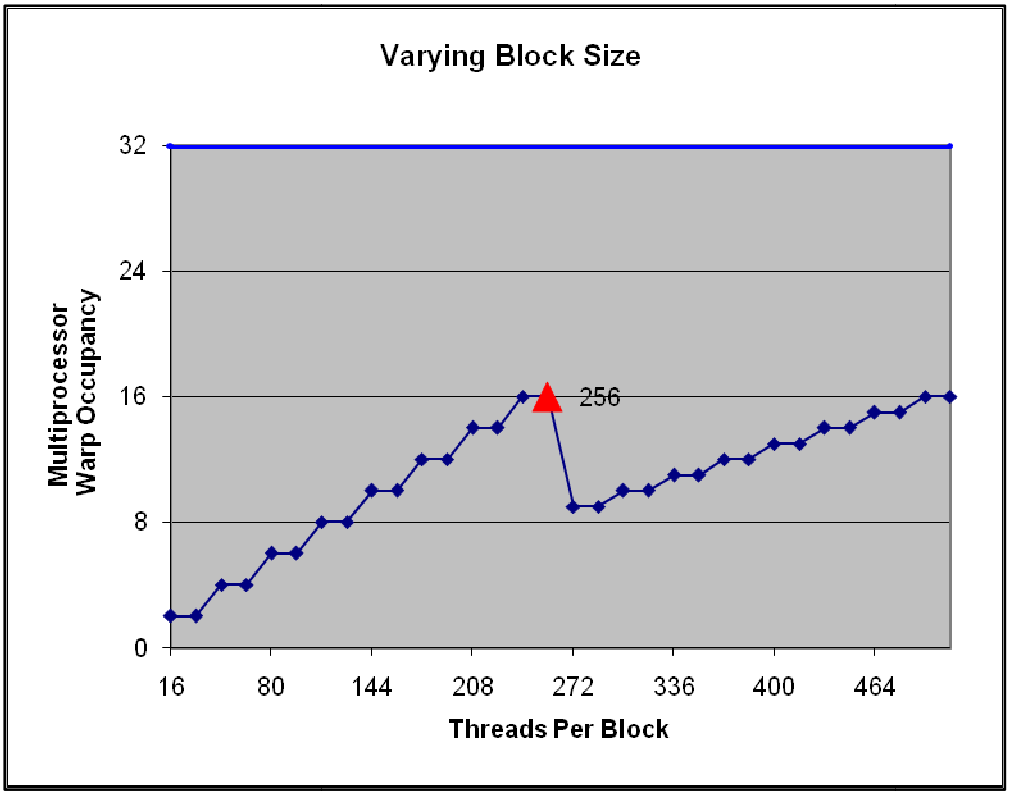

Figura 32 – Nível de Ocupação do Multiprocessador para segunda versão do *kernel* da Equação de Poisson.

O *kernel* usado para resolução da Equação de Poisson realiza pré-somatórios das contribuições dos elementos da matriz de densidades $\delta\rho(r)$ para os cálculos da energia de Hartree (um somatório) e das forças de deslocamento atômicas (seis somatórios). Os pré-somatórios são realizados para cada bloco. Assim, para obterem-se os valores finais, foi utilizado um segundo *kernel*, o qual utiliza 13 registradores e 3620 bytes de memória compartilhada para cada bloco com 256 *threads*. O Nível de Ocupação do Multiprocessador é de 100 %, conforme o ilustrado na Figura 33.

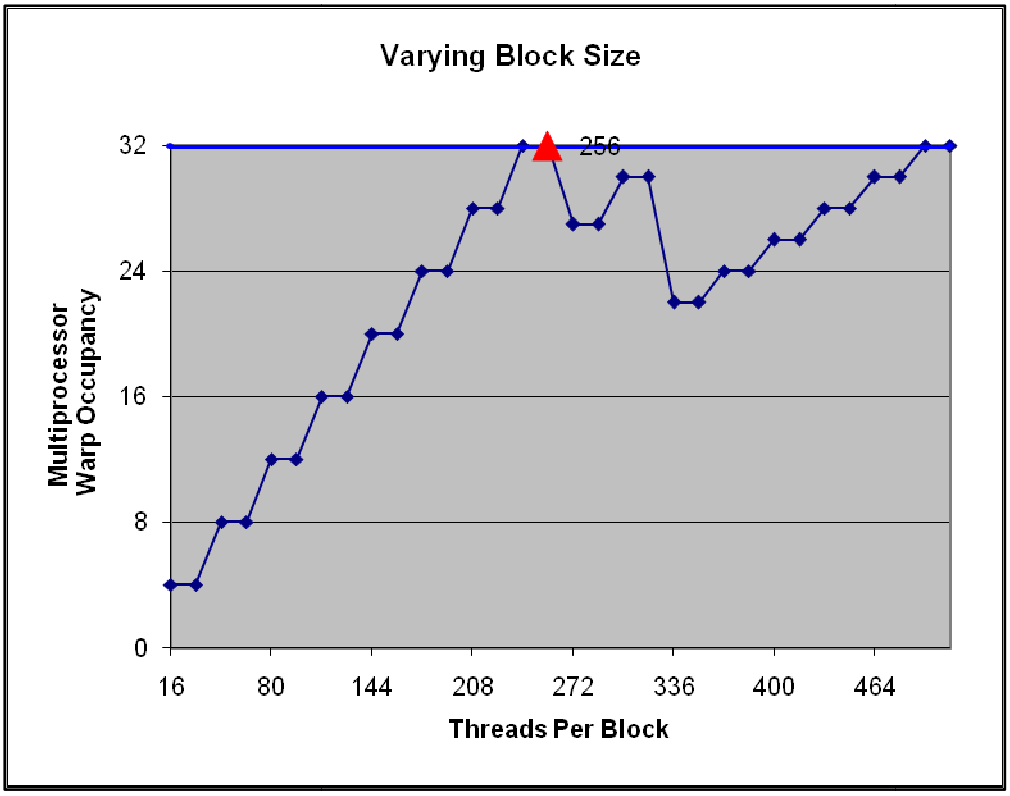

Figura 33 – Nível de Ocupação do Multiprocessador para o *kernel* dos somatórios da energia de Hartree e das forças de deslocamento atômicas.

A parte paralela dos sete somatórios, realizados de forma combinada no *kernel* da Equação de Poisson, ou no *kernel* para obtenção dos valores finais, poderia ser constituída de sete replicas do padrão ilustrado na Figura 27. Contudo, se isto fosse feito, teríamos um elevado número de operações realizadas com *warps* incompletos. A Tabela 6 mostra que, para cada bloco de *threads*, teríamos 35 operações de adição realizadas com o *warp* incompleto.

Tabela 6 – Operações de adição com *warp* incompleto por bloco, replicando sete vezes o padrão ilustrado na Figura 27.

			Operação com *Warp*	
N Total Adições / Bloco	***Threads* Ativas**	**Adições / *Thread***	**Completo**	**Incompleto**
896	128	7	28	
448	64	7	14	
224	32	7	7	
112	16	7		7
56	8	7		7
28	4	7		7
14	2	7		7
7	1	7		7
			49	**35**

Assim, com o objetivo de reduzir as operações de adição realizadas com *warps* incompletos, os sete somatórios são realizados seguindo o padrão ilustrado na Figura 34. Neste caso, ao invés de declarar sete espaços na memória compartilhada, um único espaço de memória é declarado. Os elementos pertencentes a um mesmo somatório são armazenados mantendo-se seis posições não usadas na memória. Conforme o apresentado na Tabela 7, o número de operações de adição com *warps* incompletos foi reduzido para 5.

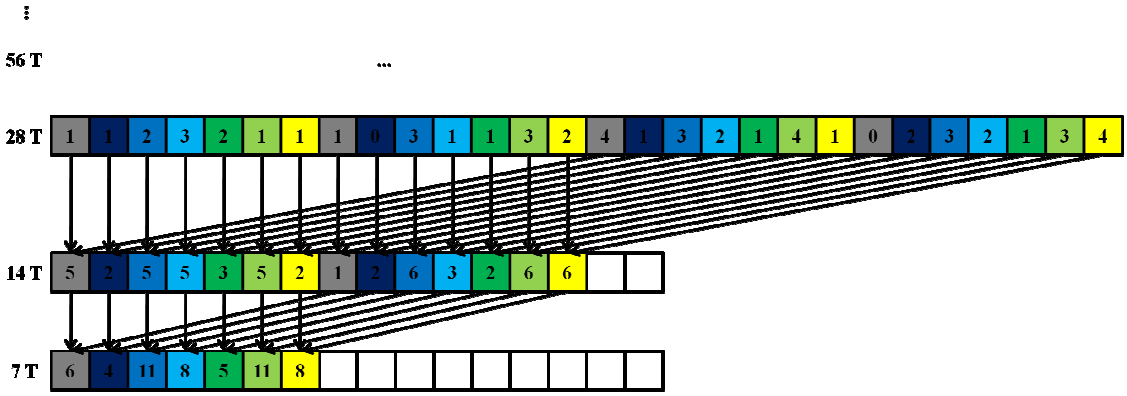

Figura 34 – Somatório paralelo de sete vetores na memória compartilhada da GPU.

Tabela 7 – Operações de adição com *warp* incompleto por bloco, com o padrão ilustrado na Figura 34.

			Operação com *Warp*	
N Total Adições / Bloco	***Threads* Ativas**	**Adições / *Thread***	**Completo**	**Incompleto**
896	256	3	24	
	128	1	4	
448	256	1	8	
	192	1	6	
224	224	1	7	
112	112	1	3	1
56	56	1	1	1
28	28	1		1
14	14	1		1
7	7	1		1
			53	**5**

O somatório paralelo dos sete vetores, da forma como o ilustrado na Figura 34, realiza um padrão de acesso linear na memória compartilhada da GPU e, portanto, não existem conflitos de banco. Na etapa anterior ao somatório paralelo, as *threads* que calculam os valores a serem somados, acessam a memória compartilhada segundo um padrão de seis intervalos entre cada elemento acessado. Conforme o ilustrado na esquerda da Figura 35, este tipo de acesso também está livre de conflitos de bancos.

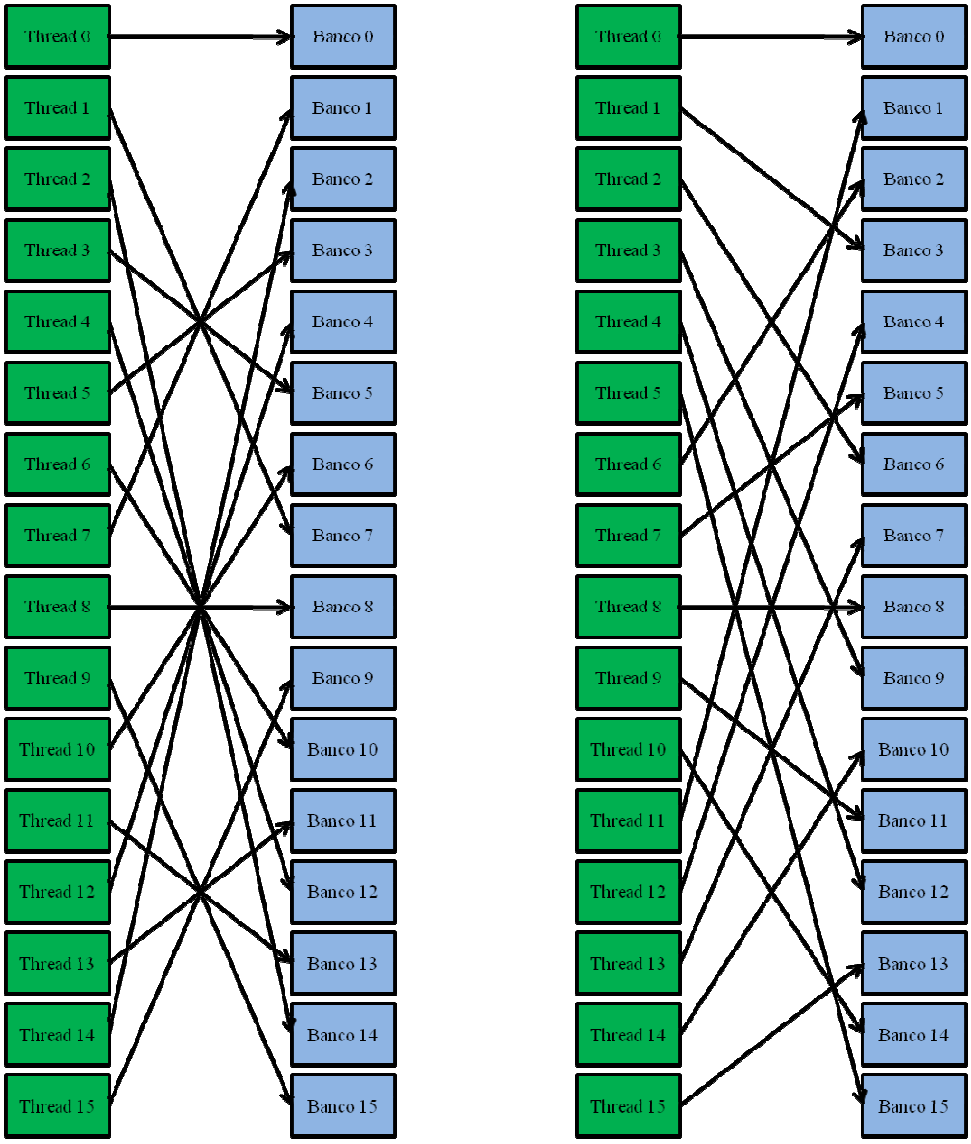

Figura 35 – Padrões usados de acesso à memória compartilhada, sem conflitos de banco.

4.2.3. Cálculo do Dipolo Elétrico

Uma versão CUDA desta função foi escrita para substituir a função original do SIESTA, onde os laços foram substituídos pelo paralelismo de dados da GPU. As contribuições dos elementos da matriz de densidades $\delta\rho(r)$ para o Dipolo Elétrico são calculadas no *kernel* e em seguida, um somatório paralelo em GPU é realizado. Cada bloco deste *kernel*, com 256 *threads*, utiliza 18 registradores e 3104 bytes de memória compartilhada. O Nível de Ocupação do Multiprocessador, apresentado na Figura 36, é de 75 % para este caso.

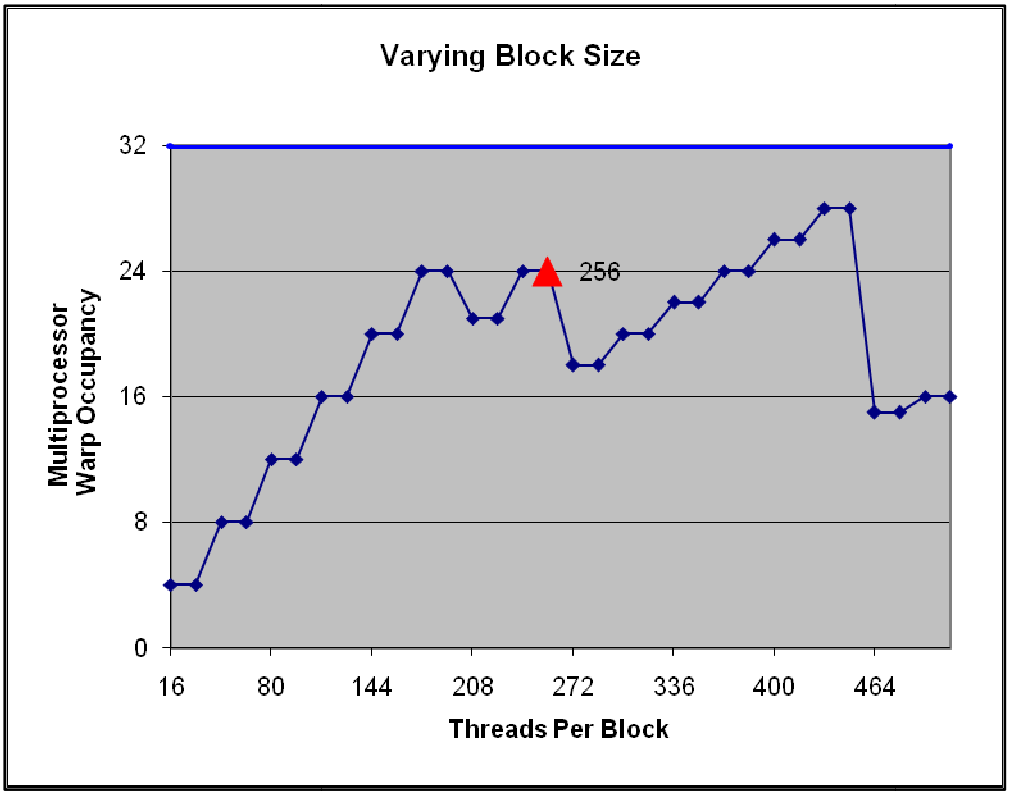

Figura 36 – Nível de Ocupação do Multiprocessador para o *kernel* de Cálculo do Dipolo Elétrico.

Para o cálculo do Dipolo Elétrico são necessários a realização de três somatórios. O padrão ilustrado na Figura 27 poderia ser repetido três vezes, contudo, teríamos 15 operações de adição realizadas com *warps* incompletos (Tabela 8).

Tabela 8 – Operações de adição com *warp* incompleto por bloco, replicando três vezes o padrão ilustrado na Figura 27.

			Operação com *Warp*	
N Total Adições / Bloco	***Threads* Ativas**	**Adições / *Thread***	**Completo**	**Incompleto**
384	128	3	12	
192	64	3	6	
96	32	3	3	
48	16	3		3
24	8	3		3
12	4	3		3
6	2	3		3
3	1	3		3
			21	**15**

A realização dos três somatórios de forma combinada, usando o padrão ilustrado na Figura 37, reduz o número operações de adição realizadas com *warps* incompletos para 5, conforme os cálculos apresentados na Tabela 9. Neste caso,

os elementos pertencentes a um mesmo somatório são armazenados mantendo-se duas posições não usadas na memória.

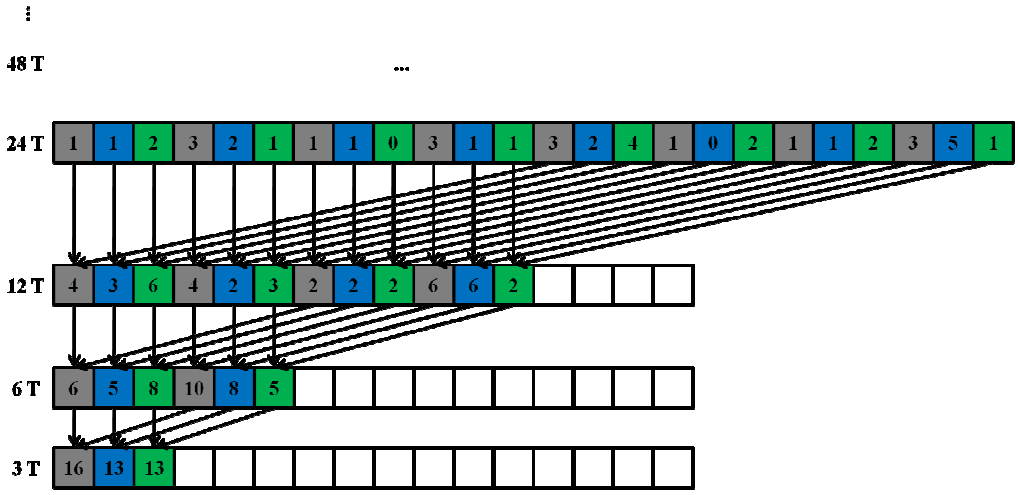

Figura 37 – Somatório paralelo de três vetores na memória compartilhada da GPU.

Tabela 9 – Operações de adição com *warp* incompleto por bloco, com o padrão ilustrado na Figura 37.

			Operação com *Warp*	
N Total Adições / Bloco	***Threads* Ativas**	**Adições / *Thread***	**Completo**	**Incompleto**
384	256	1	8	
	128	1	4	
192	192	1	6	
96	96	1	3	
48	48	1	1	1
24	24	1		1
12	12	1		1
6	6	1		1
3	3	1		1
			22	**5**

Durante a etapa do somatório paralelo dos três vetores, Figura 37, o acesso à memória compartilhada é realizado de forma linear, sem conflitos de banco. O cálculo dos elementos a serem somados, na etapa anterior, emprega um padrão de acesso com dois intervalos entre cada posição acessada na memória compartilhada. Este padrão está ilustrado na direita da Figura 35 e, como pode ser visto, também está livre de conflitos de bancos.

4.2.4. Reordenação de Dados

O SIESTA emprega duas formas de organizar os elementos da matriz de densidades, $\delta\rho(r)$, e do potencial de Hartree, $\delta V_H(r)$. Assim, a função de reordenação de dados faz o mapeamento entre estas duas organizações diferentes. Antes da execução da Equação de Poisson é feita a reordenação da matriz de densidades e após é feita a reordenação do potencial de Hartree. A reordenação é realizada também antes e depois do Cálculo do Dipolo Elétrico. O principal motivo de executar a reordenação de dados em GPU é a economia de transferências de memória através do barramento PCI Express. Pois ela é executada sobre os mesmos conjuntos de dados empregados para as demais funções GPU anteriormente descritas.

Nesta função foi utilizada a otimização de transferência de memória, conforme o descrito no item 3.5.3. Assim, são empregados múltiplos fluxos de processamento e a transferência assíncrona de dados através do barramento PCI Express. Isto é feito com a finalidade de redução do tempo total de execução pela sobreposição dos tempos de transferência de memória com o tempo de processamento da reordenação.

O *kernel* da Reordenação de Dados necessita de 18 registradores para cada bloco com 256 *threads*. O Nível de Ocupação do Multiprocessador para este caso é de 75 %, conforme o apresentado na Figura 38.

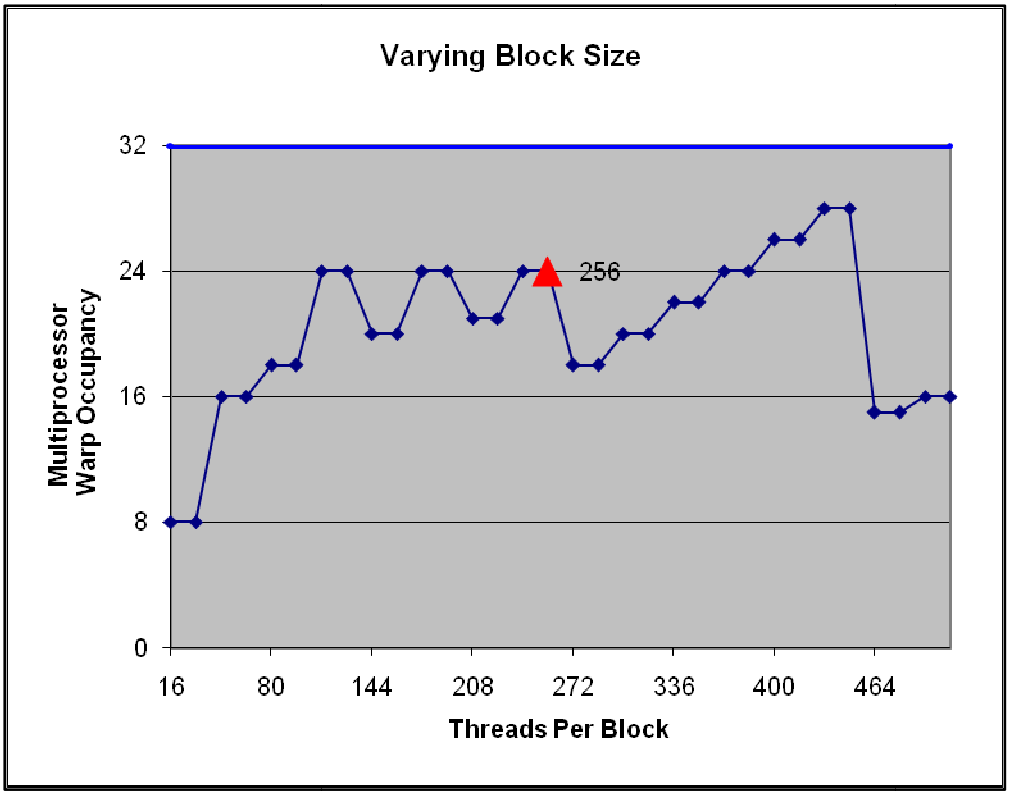

Figura 38 – Nível de Ocupação do Multiprocessador para o *kernel* da Reordenação de Dados.

Os Níveis de Ocupação do Multiprocessador empregados nos *kernels* descritos e utilizados neste trabalho estão dentro dos limites recomendados pela NVIDIA. Pois é sugerido um valor mínimo de 18.75 % para dispositivos com capacidade de cálculo 1.2 ou superior. Por outro lado, o aumento do Nível de Ocupação do Multiprocessador além do limite de 50 % não traz benefícios práticos para o aumento de desempenho (NVIDIA, 2009a).