Overview da plataforma CUDA

Pedro Geraldo M. R. Alves

12 de Junho de 2014

1 Introdução

No final de 2006, a NVIDIA apresentou *CUDA*, acrônimo de *Compute Unified Device Architecture*, uma plataforma de computação paralela de propósito geral que transforma suas *GPUs*, *Graphic Processor Units*, em *GPGPUs*, *General Purpose Graphic Processor Units*. Seu objetivo é permitir a solução de importantes problemas computacionais de maneira mais eficiente do que são resolvidos em *CPUs*.

Utilizando principalmente extensões das linguagens de programação C e C++, com CUDA o programador consegue fazer uso da *GPU* da máquina para processamento geral, não apenas gráfico. Esse, por sua vez, tem altíssimo poder de processamento paralelo devido as motivações de seu desenvolvimento.

Em seu início, o mercado de placas gráficas tinha como objetivo principal a oferta da performance necessária para consumidores de jogos virtuais. Os softwares consumidos por esse grupo são caracterizados por requererem processamento de grandes quantidades de dados de forma independente e com resultado sendo entregue tão rapidamente quanto possível.

Por isso, a *GPU* evoluiu como um processador com altissimo poder de paralelismo, sendo capaz de lidar com centenas ou milhares de *threads* em paralelo realizando operações aritméticas complexas e com uso intenso de memória. As figuras 1 e 2 demonstram a performance de *GPUs* nesse tipo de operação em comparação com *CPUs*.

A especialização de *GPUs* é em paralelismo de dados. Esse tipo de problema é caracterizado pelo processamento de forma independente de elementos do problema original. Esses costumam fazer uso intenso de operações aritméticas. *GPUs* são fortes nisso por possuirem maior quantidade de *ALUs*, como visto na figura 3.

Processamento gráfico é um exemplo óbvio de tarefa que pode ser classificada como um problema de paralelismo de dados com intenso uso aritmético.

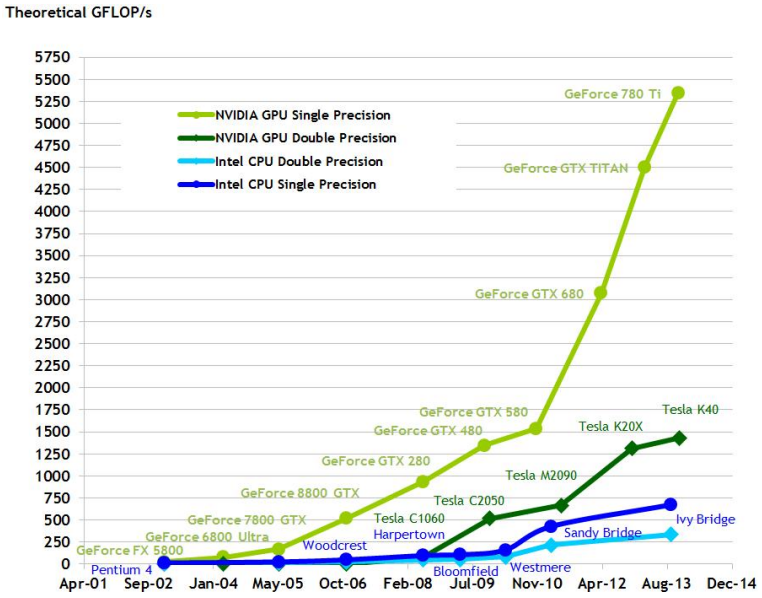

Figura 1: Comparativo de desempenho de processamento em operações de ponto flutuante entre CPU e GPU [1].

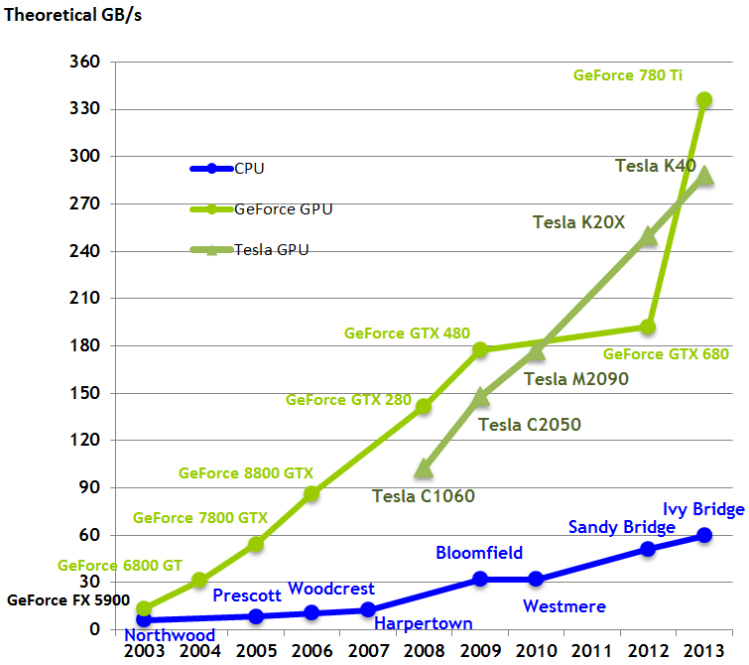

Figura 2: Comparativo de desempenho de banda de memória entre CPU e GPU [1].

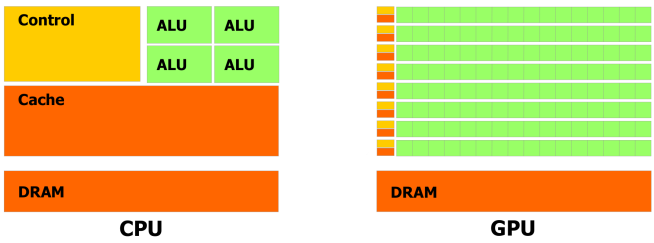

Figura 3: Esquema de estrutura interna de CPUs e GPUs [1].

Ele pode ser dividido em blocos de pixels e mapeados em *threads*. Contudo, mesmo ao nos distanciarmos dos problemas gráficos, ainda encontramos diversos outros exemplos em simulações físicas, geofísicas, financeiras e até biológicas.

O custo benefício de fazer uso dessa plataforma para processamento paralelo é justificado pela alta performance somada ao baixo custo das unidades gráficas. Além disso, seguindo a tendência da última década, temos o processamento paralelo como estratégia central na evolução do desempenho computacional em um momento onde o simples aumento do clock já não é mais factível para alcançar melhor desempenho.

2 Modelo de programação

O modelo de programação adotado para a plataforma CUDA visa simplificar sua aplicação em diferentes sistemas, que podem possuir uma ou múltiplas *GPUs*, cada uma com quantidades diferentes de *threads* operando sobre diferentes configurações de hardware. Dessa forma, não é transparente para a aplicação ou para o sistema operacional a coordenação da execução paralela, mas apenas para a *GPU*.

O programador deve manipular três tipos de abstrações: agrupamento de *threads*, uso dos diferentes tipos de memórias disponíveis e sincronização. As ferramentas para isso se consolidam como extensões da linguagem.

Essas abstrações, por requererem conhecimentos profundos do programador quanto ao funcionamento do hardware da placa gráfica, implicam em uma lenta e penosa curva de aprendizado para o uso ótimo. Contudo, também guiam o programador no processo de dividir seu problema em blocos menores e favorecendo o paralelismo de dados e tarefas, principal alvo dessa plataforma.

3 CUDA Threads

"If you were plowing a field, which would you rather use: two strong oxen or 1024 chickes?",

Seymour Cray, Father of the Supercomputer.

Threads são fluxos de processamento independente e portanto com possibilidade de execução paralela por um ou mais processadores. No caso de *GPUs*, sua execução é feita por *streaming multiprocessors* [2].

Na documentação fornecida pela NVIDIA e na comunidade CUDA, *threads* frequêntemente são chamados "CUDA *Threads*". O uso de um nome diferente para um conceito que transcende *GPUs* serve principalmente para lembrar os programadores que, apesar de semelhantes, existem diferenças profundas no funcionamento de CUDA *Threads* e *threads* executados pela *CPU*. Por simplicidade, nesse trabalho não haverá distinção entre os termos a menos que seja explicitado.

Um CUDA *Thread* é classificado como "peso-leve" (ou *lightweight* em inglês). Apesar das *GPUs* serem capazes de lançar e executar em paralelo milhares de *threads*, sua performance individual é baixa e não tem a mesma flexibilidade de seus primos que rodam em *CPUs*. Ele nem mesmo é capaz de realizar comunicação direta com outro CUDA *Thread*, apenas indireta fazendo uso de alguma das memórias compartilhadas disponíveis, como visto na seção 4. Por isso, o algoritmo executado por esses *threads* não pode ter dependência de outros *threads* e nem mesmo pode ser muito caro computacionalmente.

Uma confusão comum em iniciantes envolve a tentativa de reproduzir algoritmos criados para "*CPU threads*" em *CUDA Threads*. Seguindo a citação de Seymour Cray, este é comparável a um exército de 1024 galinhas (ou muito mais hoje em dia), enquanto aquele é comparável a dois (ou quatro) bois. Você não pode submeter uma galinha a mesma carga de trabalho que submete um boi. Ela é mais limitada. Da mesma forma, um *CUDA Thread* é mais limitado do que um *CPU Thread*.

Essas restrições nos levam a um fato crítico do uso da plataforma. O uso de CUDA só faz sentido em problemas de paralelismo de dados.

Como pode ser visto nas seções 4 e 5, o uso de CUDA carrega *overhead* devido principalmente ao acesso a memória, seja durante a execução do código na *GPU* ou nos momentos de cópia entre a memória principal da máquina e a memória da placa gráfica.

Dessa forma, precisamos fortalecer a afirmação anterior. Podemos dizer que o uso de CUDA só faz sentido em problemas de paralelismo de dados

e que possuam tamanho considerável para conseguir esconder a latência do acesso à memória.

3.1 Hierarquia de Threads

Em CUDA, cada *thread* faz parte de um bloco de *threads*. Esses blocos possuem até três dimensões, atribuindo assim até três coordenadas para cada *thread* dentro do bloco. Da mesma forma, cada bloco faz parte de um *grid* de blocos. Esse possuindo até duas dimensões e portanto atribuindo duas coordenadas de identificação para blocos. Essa estrutura pode ser vista na figura 4 e permite, por exemplo, a implementação da hierarquia de memória descrita na seção 4.1.

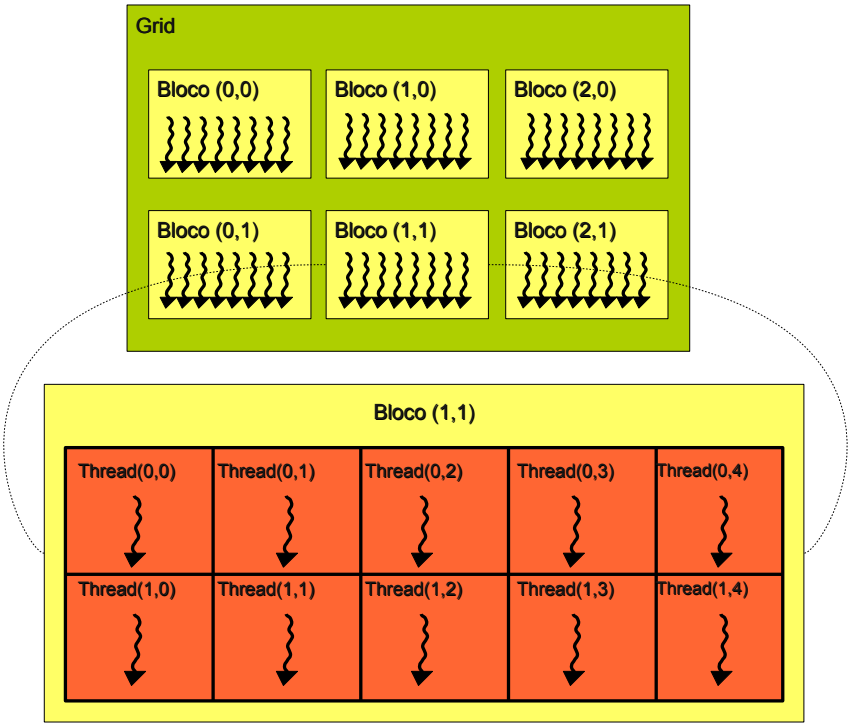

Figura 4: Esquema estrutural de threads em um kernel, onde temos um grid bi-dimensional composto por blocos bi-dimensionais.

Apesar de completamente virtual, essa estrutura facilita a codificação da divisão de trabalho pelos *threads*. Como pode-se ver na seção 3.3, é importante que exista balanceamento do trabalho executado por *threads* dentro de um *warp*.

3.2 Kernels

Na extensão da linguagem C, criada pela NVIDIA para tirar proveito da plataforma CUDA, todos os *threads* de uma execução iniciam executando uma mesma função, chamada *Kernel*.

Uma função *kernel* é o ponto de partida do paralelismo por CUDA e o grande diferencial desse modelo de programação em relação aos modelos de paralelismo pela *CPU*. Estes tem *threads* criados isoladamente e com percursos individuais.

Um *kernel*, quando chamado, é executado N vezes por N diferentes *threads* na *GPU*. Isso é bem diferente do que acontece, por exemplo, em uma implementação de paralelismo pela *CPU* com a biblioteca *pthread*, onde cada *thread* é criado individualmente e pode executar funções completamente diferentes em momentos diferentes.

3.3 Warps

O gerenciamento de *threads* em CUDA frequentemente é explicado fazendo-se referência ao fluxo de um rio. Seu comportamento ideal esperado tem todas as moléculas de água com vetor direção e velocidade idênticos. Caso exista desvios ou obstáculos, todo o fluxo sente o efeito e tem sua performance comprometida.

Apesar da capacidade de executar muito mais *threads* do que uma *CPU*, ainda assim é evidente que existe um limite de *threads* que podem ser executados simultâneamente dentro de um *streaming multiprocessor*. Chamamos o conjunto de *threads* em execução dentro de um *SM* em dado instante de *warp*.

Uma vez que a função *kernel* é chamada, a quantidade de *threads* especificada em seus parâmetros precisa ser alocada através dos *SMs* disponíveis. Isso é feito dinâmicamente com o mapeamento de *warps* em *SMs*.

A NVIDIA classifica CUDA como SIMT (*Single Instruction, Multiple Thread*). No início da execução de um *warp*, é feito o *fetch* de instruções. Esse é feito não para os *threads* individualmente, mas para todo o *warp*. Dessa forma, a ocorrência de *branchs* dentro de um *warp* tem implicações sérias de performance.

A estratégia para lidar com *branchs* é, caso um *thread* não siga o mesmo caminho que o *warp*, ele é marcado como *null op* e seu processamento é feito em um *branch* divergente junto de todos os outros *threads* daquele *warp* que tiveram o mesmo comportamento.

Dessa forma, para o melhor aproveitamento do poder de paralelismo da plataforma CUDA, o programador se vê obrigado a re-escrever o problema

que tenta resolver de forma a quebra-lo em blocos tão granulares e simples quanto possível, evitanto o uso de condicionais quando possível. Quando necessário, para otimização de desempenho, espera-se que fiquem nos limites de *warps*. Assim evita-se a criação de *warps* divergentes e o disperdício de recursos com *threads* sendo marcados como *null op*.

Uma questão interessante é que, caso seja inevitável fazer uso de condicionais em um algorítmo, pode ser que a performance não seja perceptivelmente afetada mesmo se o programador não for capaz de encontrar um agrupamento adequado de *threads* em *warps*.

No caso do cálculo de autovalores, por exemplo, existe diversos casos de *branchs* aninhados. Contudo, a occorrência de divergência é baixa, possibilitando que a eficiência da execução desse algoritmo na *GPU* seja maior do que o esperado para um algoritmo com tantas condicionais aninhados.

O grande desafio do uso de CUDA na solução de um problema é lidar com os *warps*. O programador precisa manter em mente durante todo o tempo o funcionamento do hardware da *GPU*, e com isso modelar sua solução do problema agrupando *threads* que provavelmente seguirão o mesmo caminho em caso de *branch*, para evitar quebra de fluxo.

4 Memória

Realizar o processamento de dados dentro de uma *GPU* depende de um fato bastante óbvio: o dado precisa estar na *GPU*.

O modelo de memória usado por CUDA propõe uma separação entre a memória da *GPU* e a memória principal da máquina. Mesmo que eventualmente não exista uma separação física (no caso de uma *GPU* integrada por exemplo), ainda assim haverá a necessidade do programador fazer a cópia dos dados entre as memórias.

4.1 Hierarquia de Memória

Todo *thread* possui basicamente três espaços de memória para realizar leituras e gravações.

Local memory Restrita à cada *thread*.

Shared memory Compartilhada com todos os *threads* de um bloco.

Global memory Compartilhada com todos os *threads*. Mesmo aqueles de execuções de *kernels* diferentes.

Além desses, existem também as memórias *constant* e *texture*. Ambas assumem modo apenas-leitura para *threads* em execução na *GPU* e tem a mesma regra de compartilhamento que a memória global.

No caso das memórias locais e compartilhadas, o tempo de vida dos dados armazenados é igual ao tempo de vida do *thread*. No caso da memória global, *constant* e *texture*, o tempo de vida é igual ao tempo de execução do programa.

A hierarquia de memória descrita pode ser vista na figura 5.

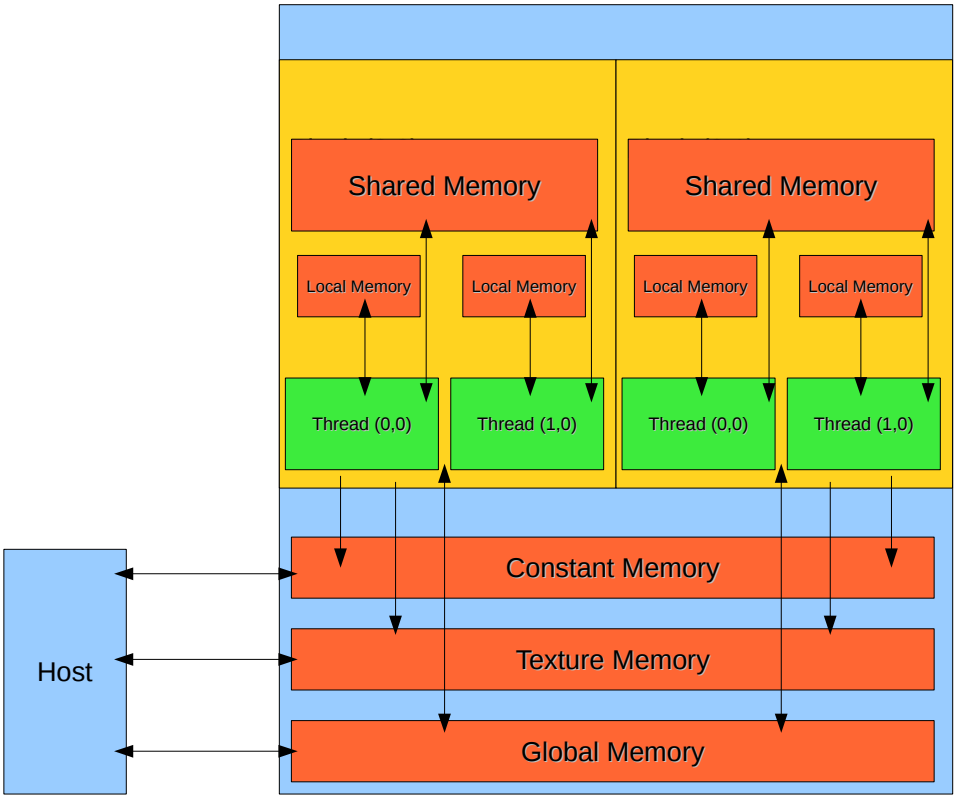

Figura 5: Esquema da hierarquia de memória exposta ao programador na plataforma CUDA. Toma-se como exemplo a visibilidade das memórias para um grid de dois blocos, cada um com dois threads

5 Comparação de Performance

A adoção de uma tecnologia depende da demonstração que essa pode trazer ganhos de eficiência na resolução de uma classe de problemas ou até mesmo

em um único problema de escopo amplo. Para analisarmos os possíveis ganhos que a plataforma CUDA pode trazer, utilizo as referências [3] e [4].

Nesse trabalho não focarei nos detalhes de implementação mas nos resultados obtidos.

5.1 Calculando a Raiz Quadrada

GPUs possuem uma grande quantidade de *ALUs*. Por isso, um problema bastante simples e que pode realçar os ganhos de performance dessa plataforma é no cálculo da raiz quadrada.

A referência [3] faz uso de uma *GPU* NVIDIA Tesla C870 e uma *CPU* Intel Xeon E5410. Em ambos, um *array* de inteiros é gerado, a raiz quadrada de cada elemento é calculada e armazenada em um segundo array de mesmo tamanho.

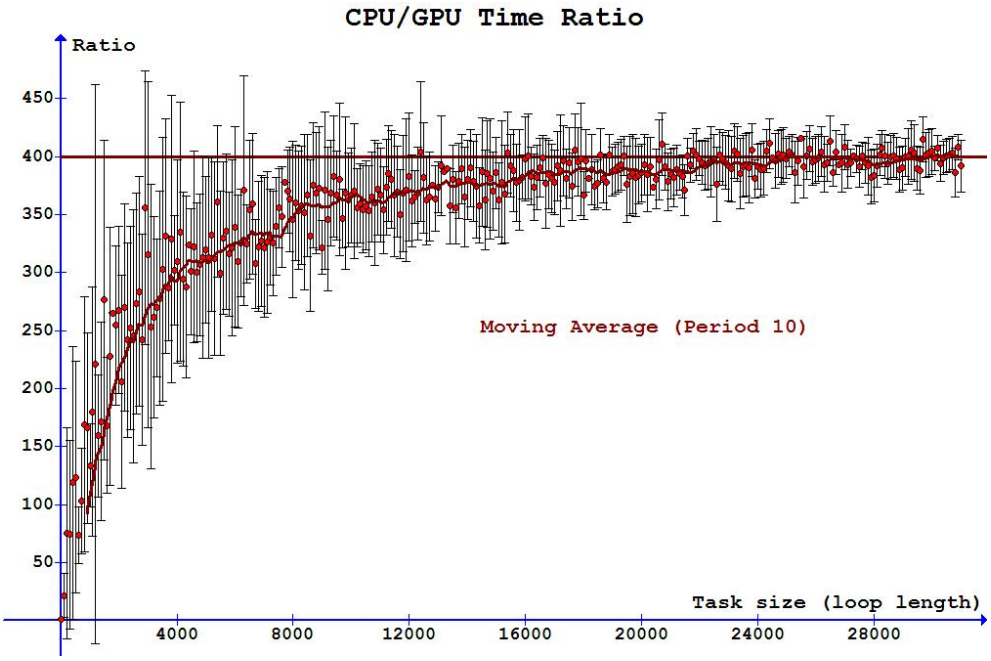

Figura 6: Razão dos tempos de CPU e GPU para arrays de diferentes tamanhos. Pode-se observar a assintota se fixando na razão de 400.

O desempenho médio dessa operação realizada com CUDA, como pode ser visto na figura 6 chegou a um *speedup* de 400. Tal desempenho já era esperado visto a enorme diferença na quantidade de *ALUs*.

Uma observação interessante encontrada nesse benchmark é que, para tarefas pequenas, o *overhead* gerado pela transferência de dados entre as

memórias torna o processamento pela *GPU* mais lento do que pela *CPU*. Esse *overhead* se tornou desprezível quando o tamanho do problema aumentou, sugerido pela estagnação do crescimento do *speedup* após esse atingir 400.

5.2 Benchmarks mais realistas

O cálculo da raiz quadrada, apesar de ser uma operação importante, por si só não pode ser usado para demonstrar a aplicabilidade da plataforma CUDA em aplicações reais. Para isso, a referência [4] demonstra o uso da plataforma em três problemas classificados como "The Dwarf Mine": Álgebra Linear Esparsa, Métodos Espectrais e MapReduce.

Os hardwares comparados são: Intel Xeon E5530 (formado por dois *cores* Nehalem), Intel Core 2 Duo E8500 e NVIDIA GTX 280. Nos três casos, compara-se o *speedup* obtido no paralelismo por *OpenMP* (no caso das *CPUs*) ou CUDA (no caso da *GPU*) em relação a execução sequêncial.

A implementação que usa a *GPU* foi feita através do porte de código de um conjunto de benchmarks para *CPUs* chamado NAS Parallel Benchmarks [5] que utiliza *OpenMP* para paralelismo. Os testes feitos possuem tamanho diferente e são classificados pelas letras W, A, B, C e D, onde W é o menor tamanho e D o maior.

Problemas de Álgebra Linear Esparsa são baseados em matrizes esparsas e suas propriedades. Essas matrizes são gigantes e formadas em maior parte por zeros. Por isso, sua representação é feita de forma a otimizar o espaço de memória consumido. Os acessos de memória realizados em matrizes nesse formato não são ordenados, o que não privilegia o processamento paralelo. Isso pode ser confirmado pelo pequeno *speedup* obtido pelas três implementações paralelas. No caso do sistema com dois *cores* Nehalem, o overhead é tão grande que a implementação paralela chega a ser mais lenta do que a solução sequencial, como visto na figura 7.

Problemas de Métodos Espectrais costumam apresentar baixa intensidade de cálculos aritméticos devido ao intenso uso de memória. Contudo, possuem comportamento previsível com pouca existência de condicionais, permitindo bom desempenho com plataforma CUDA, como visto na figura 8.

Problemas de MapReduce são caracteristicos por intenso paralelismo com pouca ou nenhuma comunicação entre as *threads*. Esse é o cenário perfeito para a aplicação de CUDA, o que pode ser confirmado pelo impressionante *speedup* de até 42, visto figura 9.

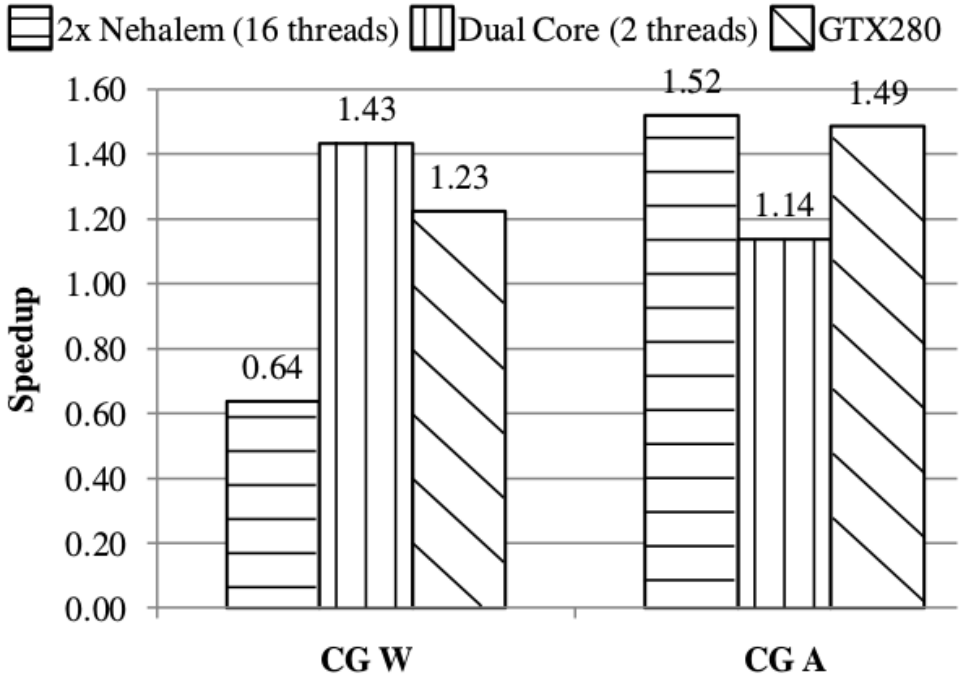

Figura 7: Speedup no uso de paralelismo em um problema de Álgebra Linear Esparsa. São testados dois sistemas com processamento por CPU e um com processamento por GPU em comparação com a performance em uma execução sequencial.

6 Aplicações

A NVIDIA fornece em [7] uma relação de mais de 200 aplicações distribuídas por diversos campos. Entre eles, podemos citar dinâmica de fluidos, simulação de modelos climáticos, ramos da criptografia como o de cripto-moedas e traçamento de raios aplicado em estudos geofísicos.

Em especial, esse último pode ser exemplificado por um trabalho anterior feito pelo mesmo autor desse artigo em [6].

7 Conclusão

A plataforma CUDA é uma solução elegante para explorar um tipo de hardware que, devido a sua natureza, se mostrou bastante eficiente na solução de diversas classes de problemas.

Contudo, seu uso ótimo depente do domínio por parte do programador

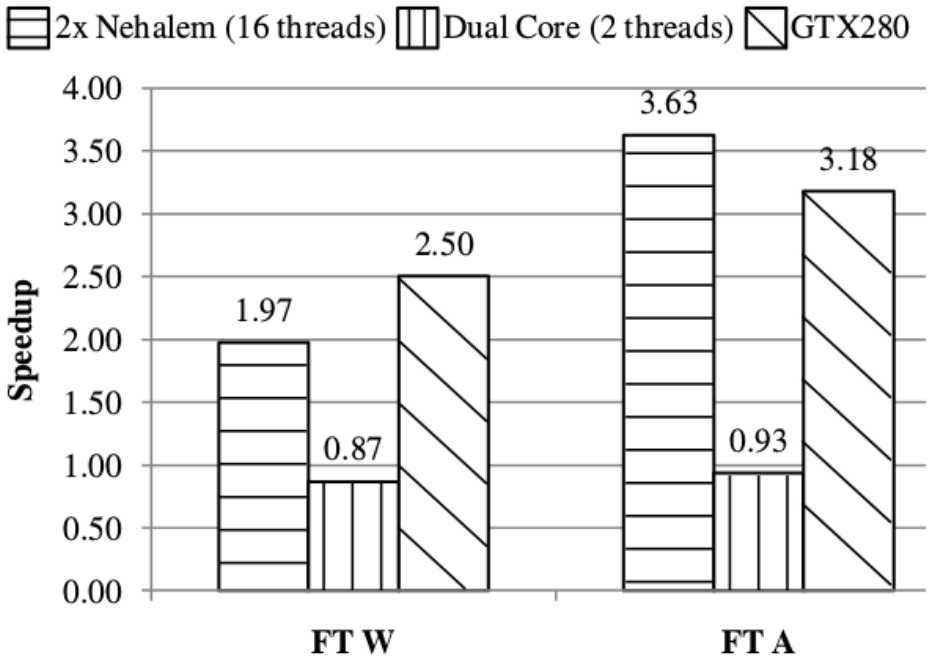

Figura 8: Speedup no uso de paralelismo em um problema de aplicação do teorema espectral. São testados dois sistemas com processamento por CPU e um com processamento por GPU em comparação com a performance em uma execução sequencial.

de um modelo de programação relativamente complicado e pouco intuitivo para iniciantes ou aqueles que já são acostumados com paralelismo pela *CPU* utilizando outras bibliotecas e plataformas. Essa lenta curva de aprendizado, contudo, é superada pelos vultuosos valores de speedup que podem ser obtidos.

Com o foco no desenvolvimento da plataforma, a NVIDIA tem oferecido gerações de placas gráfica cada vez mais voltadas para o processamento de propósito geral. Isso permite uma simplificação do modelo de programação a cada iteração do SDK. Pode-se esperar que em alguns anos a facilidade da implementação de soluções com CUDA seja comparável a *OpenMP*, mas mantendo a performance muito superior ao usar *GPGPUs*.

Referências

[1] NVIDIA Corporation, *CUDA C PROGRAMMING GUIDE*. PG-02829-001_v6̇0̇, February 2014.

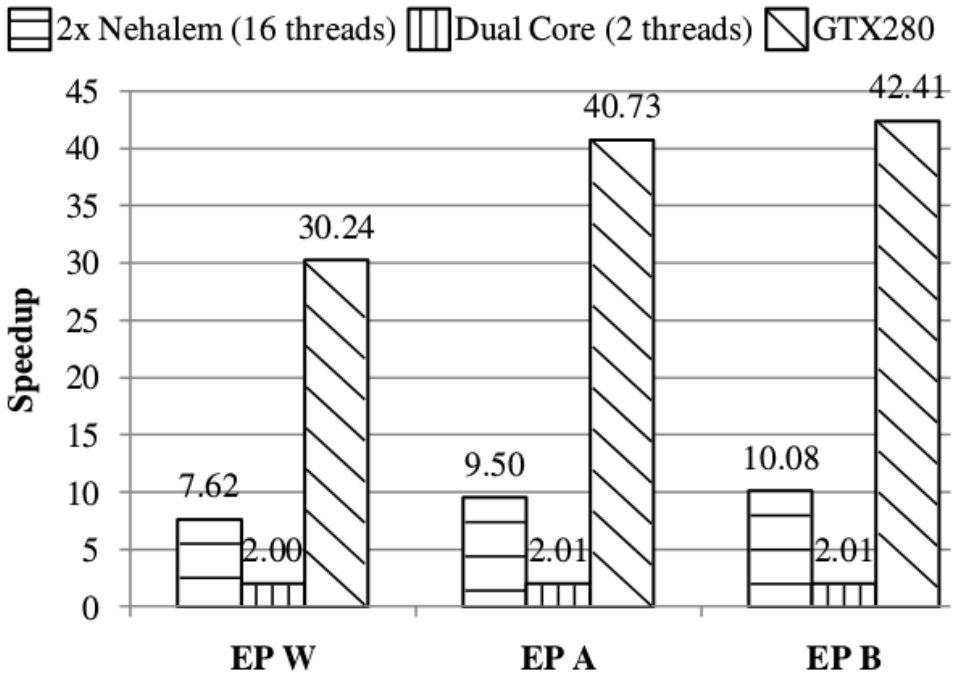

Figura 9: Speedup no uso de paralelismo em problemas do tipo MapReduce. São testados dois sistemas com processamento por CPU e um com processamento por GPU em comparação com a performance em uma execução sequencial.

[2] Nicholas Wilt, *The CUDA Handbook*. Pearson Education Inc. , 2012.

[3] Abhranil Das, *Process Time Comparison between GPU and CPU*. Hamburger Sternwarte, University of Hamburg, July 20, 2011.

[4] Laercio L. Pilla, Philippe O. A. Navaux, *Evaluating CUDA Compared to Multicore Architectures*. Institute of Informatics - Federal University of Rio Grande do Sul - Porto Alegre, Brazil.

[5] H. Jin, M. Frumkin, and J. Yan. *The OpenMP implementation of NAS parallel benchmarks and its performance.*. NASA Ames Research Center, Technical Report NAS-99-011, 1999.

[6] Pedro G. M. R. Alves, Ricardo Biloti, *Traçamento de raios em GPGPUs*. Universidade Estadual de Campinas & INCT-GP, Brasil, 2011.

[7] NVIDIA Corporation, *GPU-ACCELERATED APPLICATIONS*.