Arquitetura e Programação de GPU Nvidia

Leandro Zanotto [*]
Instituto de computação
Universidade Estadual de Campinas
Campinas, São Paulo, Brasil

Anselmo Ferreira [†]
Instituto de Computação
Universidade Estadual de Campinas
Campinas, São Paulo, Brasil

Marcelo Matsumoto [‡]
Instituto de Computação
Universidade Estadual de Campinas
Campinas, São Paulo, Brasil

ABSTRACT

Atualmente já existe um consenso em relação à importância das *Graphical Processing Units* (GPUs) em sistemas computacionais. O crescimento de sua capacidade de processamento e quantidade de aplicações que tiram proveito de sua computação de alto desempenho vêm aumentando desde a sua criação em 1999. Sua arquitetura se baseia em um grande número de núcleos para processamento massivo paralelo com foco na eficiência energética. Essa placas também podem ser utilizadas para aplicações de propósito geral (técnica denominada GPGPU) através da programação em CUDA (*Compute Unified Device Architecture*). Nesse artigo serão abordados a arquitetura da GPU da empresa NVIDIA, sua programação utilizando-se CUDA e aplicações e ganhos reais de desempenho com seu uso.

PALAVRAS-CHAVE:

GPU, CUDA, NVIDIA.

1. INTRODUÇÃO

Desde a criação da computação, os processadores operavam como um único núcleo programável, contendo códigos executados sequencialmente. Porém, a demanda por maior desempenho fez com que empresas investissem muito dinheiro no aumento de sua velocidade, exigindo cada vez mais do silício. Para contornar o limite físico do silício, foram criados os mecanismos de pipeline, threads entre outros. Adicionar processadores em paralelo ou vários núcleos em um único chip foram outras alternativas para aumentar seu desempenho. Porém, para aproveitar-se destes tipos de processamento é preciso mudar os códigos sequenciais e buscar por técnicas de software que possam aproveitar esses novos recursos.

Ao perceber a alta demanda por um tipo de processamento específico, a empresa americana NVIDIA, conhecida pela fabricação de placas de vídeo, lançou, em 1999, sua primeira GPU: a GeForce 256 [1] e logo mais à frente, em 2000, criou as GPUs de propósito geral (GPGPU - General-Purpose Computing on Graphics Processing Units). Hoje a NVIDIA possui um total de 1 bilhão de GPUs vendidas [2].

Inicialmente, a GPU era um processador de função fixa, construído sobre um pipeline gráfico que se sobressaía apenas em processamento de gráficos em três dimensões. Desde então, a GPU tem melhorado seu *hardware*, com foco no aspecto programável da GPU. O resultado disso é um processador com grande capacidade aritmética, não mais restrito às operações gráficas.

Atualmente, as GPUs são processadores dedicados para processamento gráfico da classe SIMD (*Single Instruction Multiple Data*). GPUs são desenvolvidas especificamente para cálculos de ponto flutuante, essenciais para renderização de imagens. Suas principais características são sua alta capacidade de processamento massivo paralelo e sua total programabilidade e desempenho em cálculos que exigem um volume grande de dados, resultando em um grande *throughput*. A partir de APIs como *DirectX*, *OpenGL* e *Cg*, foi possível criar algoritmos paralelos para GPUs. Entretanto, isso requeria ao programador um grande conhecimento das APIs e da arquitetura das placas gráficas, além de ser necessário que o problema seja representado através de coordenadas de vértices, texturas e shaders, aumentando drasticamente a complexidade do programa.

Para facilitar a interface entre o programador e as aplicações GPU, a NVIDIA apresentou a *Compute Unified Device Architecture* (CUDA) em 2006. Trata-se de uma plataforma de computação paralela e modelo de programação que disponibiliza um aumento significativo de desempenho ao aproveitar o poder da GPU. Ao fornecer abstrações simples com respeito à organização hierárquica de *threads*, memória e sincronização, o modelo de programação CUDA permite aos programadores escreverem programas escaláveis sem a necessidade de aprender a multiplicidade de novos componentes de programação. A arquitetura CUDA suportas diversas linguagens e ambientes de programação, incluindo C, *Fortran*, *OpenCL*, e *DirectX Compute*. Ela também tem sido ampla-

[*]Aluno especial do mestrado em Ciência da Computação. RA 001963.
[†]Aluno regular do doutorado em Ciência da Computação. RA 023169.
[‡]Aluno especial da graduação em Engenharia da Computação. RA 085973.

mente utilizada por aplicações e trabalhos de pesquisa publicados. Hoje ela está implantada em *notebooks*, estações de trabalho, *clusters* de computação e supercomputadores.

As GPUs estão rapidamente se aperfeiçoando, tanto em programabilidade quanto em capacidade. A taxa de crescimento computacional anual das GPUs está aproximadamente em 2.3x, em contraste a isso, o das CPUs está em 1.4x [3]. Ao mesmo tempo, as GPUs estão ficando cada vez mais acessíveis. Com isso, a comunidade de pesquisa tem mapeado com sucesso uma rápida demanda por soluções computacionais complexas com um surpreendente resultado. Este fato tem posicionado a GPU como uma alternativa aos microprocessadores nos futuros sistemas de alto desempenho.

Nesse artigo serão apresentados a arquitetura e o modelo de programação para GPU, bem como a abordagem da NVIDIA para GPGPU: a CUDA. Para ressaltar os benefícios dessa abordagem, também serão apresentados alguns trabalhos científicos que aproveitaram-se dos benefícios de CUDA. O restante desse artigo está dividido em 4 seções. Na Seção 2 será apresentada a arquitetura de GPUs e na Seção 3 seu modelo de programação, incluindo a abordagem CUDA. Na Seção 4 serão apresentados alguns dos diversos trabalhos acadêmicos que utilizaram GPGPU com CUDA. A Seção 5 conclui esse artigo.

2. ARQUITETURA DE UMA GPU

Uma diferença importante entre as GPUs e as CPUs é que, enquanto as CPUs dedicam uma grande quantidade de seus circuitos ao controle, a GPU foca mais em ALUs (*Arithmetic Logical Units*), o que as torna bem mais eficientes em termos de custo quando executam um *software* paralelo. Consequentemente, a GPU é construída para aplicações com demandas diferentes da CPU: cálculos paralelos grandes com mais ênfase no *throughput* que na latência. Por essa razão, sua arquitetura tem progredido em uma direção diferente da CPU.

As GPUs da NVIDIA são divididas em vários SMs (*Streaming Multiprocessors*), que por sua vez, executam grupos de *threads*, chamados de *warps*. Cada SM possui vários núcleos, chamados de CUDA *cores*. Cada CUDA *core* possui *pipelines* completos de operações aritméticas (ALU - *Arithmetic Logic Unit*) e de pontos flutuantes (FPU - *Floating Pount Unit*). Em um SM, há uma memória cache L1 comum somente aos núcleos de um SM, e todos os *cores* de um SM têm acesso à uma memória global (*cache L2*). A Figura 1 mostra o SM da arquitetura Fermi. Note que boa parte do SM corresponde aos CUDA *cores*, caracterizando sua especialidade em operações que envolvem muitos cálculos, ao contrário das CPUs, onde a cache é predominante.

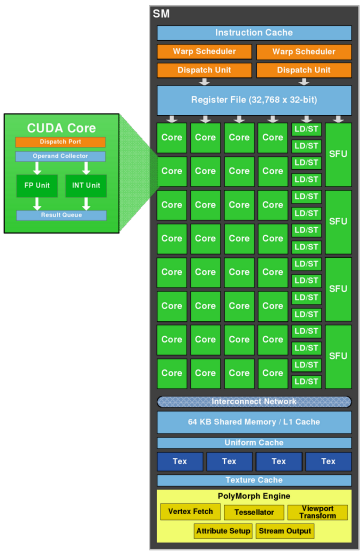

Figura 1: *Streaming Multiprocessor* da arquitetura Fermi. Extraído de [4].

2.1 A evolução da arquitetura das GPGPUs NVIDIA

2.1.1 G80

A oitava geração de placas gráficas da NVIDIA foi introduzida em novembro de 2006. A GeForce 8800 foi a primeira GPU com suporte a linguagem C, introduzindo a tecnologia CUDA. A placa gráfica introduziu a arquitetura de shaders unificada com 128 CUDA *cores*, distribuídos entre 8 SMs [5].

2.1.2 G200

Lançada em 2008, esta arquitetura da NVIDIA trouxe novas melhorias para o CUDA. Pela primeira vez foi implementado o suporte para computação com 64-bits de precisão dupla [6].

2.1.3 Fermi

Lançada em abril de 2010, esta arquitetura trouxe suporte para novas instruções para programas em C++, como alocação dinâmica de objeto e tratamento de exceções em operações de *try* e *catch*. Cada SM de um processador Fermi possui 32 CUDA *cores*. Até 16 operações de precisão dupla por SM podem ser executadas em cada ciclo de *clock* [4]. Além disso, cada SM possui:

- Dezesseis unidades de *load* e *store*, possibilitando que o endereço de fonte e destino possam ser calculados para dezesseis *threads* por *clock*.
- Quatro *Special Function Units* (SFUs), que executam instruções transcendentais, como seno, cosseno, raiz quadrada, etc. Cada SFU executa uma instrução por *thread* por ciclo. O *pipeline* da SFU é desacoplado da *dispatch unit*, permitindo que esta possa realizar o *issue* (despacho) de outra instrução enquanto a SFU está ocupada.

A Figura 2 ilustra a arquitetura GPU Fermi.

Figura 2: GPU da arquitetura Fermi. Extraído de [7].

2.1.4 *Kepler*

Lançada em 2012, a mais nova arquitetura da NVIDIA introduziu um novo modelo de SM, chamado de SMX, que possui 192 CUDA *cores*, totalizando 1536 *cores* no chip (8 SMXs). No processo de criação da arquitetura Kepler, um dos principais objetivos era obter uma melhor eficiência energética. Os transistores de 28nm foram importantes para a redução no consumo de energia, mas a alteração da arquitetura, com o uso dos SMX, foi a principal responsável pela maior redução no consumo de energia. Com essa alteração foi possível executar os CUDA *cores* em frequência maior e ao mesmo tempo reduzir o consumo de energia da GPU, fazendo com que ela aqueça menos. A Figura 3 mostra o SMX da arquitetura Kepler.

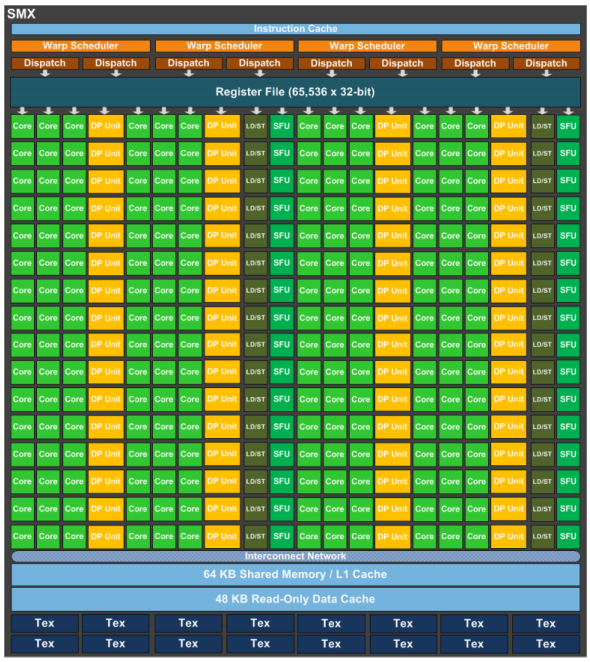

Figura 3: SMX: *Streaming Multiprocessor* da arquitetura Kepler. Extraído de [8].

Além dos 192 CUDA *cores* de precisão única, o SMX possui 64 unidades de precisão dupla, 32 *special function units* e 32 unidades de *load* e *store*. O número de unidades de precisão dupla foi significativamente aumentado no Kepler, já que essas unidades são largamente usadas em HPC (*High Performance Computing*). Além disso, o número de SFUs foram aumentadas em oito vezes em relação à arquitetura Fermi [8].

Um dos objetivos no projeto Kepler foi facilitar ainda mais aos desenvolvedores a obtenção dos benefícios da imensa capacidade de processamento paralelo da GPU. O paralelismo dinâmico no Kepler reutiliza as *threads* dinamicamente através da adaptação de novos dados, sem a necessidade de retornar os dados intermediários para a CPU. Isso permite que os programas sejam executados diretamente na GPU, já que agora os *kernels* possuem a habilidade de executar cargas de trabalho adiconais independentemente. Qualquer *kernel* pode executar outro e criar eventos, dependências e *streams* sem a interação com a CPU, como era feito nas arquiteturas anteriores. O ganho do paralelismo dinâmico é melhor ilustrado na Figura 4.

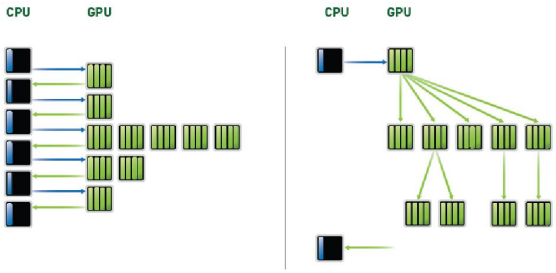

Figura 4: Sem o Paralelismo Dinâmico (à esquerda), a CPU executa cada kernel na GPU. Com o paralelismo dinâmico (à direita), a GPU Kepler GK110 pode agora executar *kernels* aninhados, eliminado a necessidade da comunicação com a CPU. Extraído de [8].

Uma inovação na arquitetura Kepler, o *Hyper-Q*, permite que múltiplos *cores* de CPU executem o trabalho simultaneamente em uma única GPU, aumentando, dessa forma, a utilização da GPU e diminuindo, assim, seu tempo ocioso. Além disso, essa arquitetura permite que ocorram até 32 conexões simultâneas entre a CPU e a GPU, ao passo que na arquitetura Fermi apenas uma conexão poderia ocorrer entre elas. A Figura 5 ilustra a diferença entre as arquiteturas Fermi e *Hyper-Q* no tocante às conexões CPU-GPU.

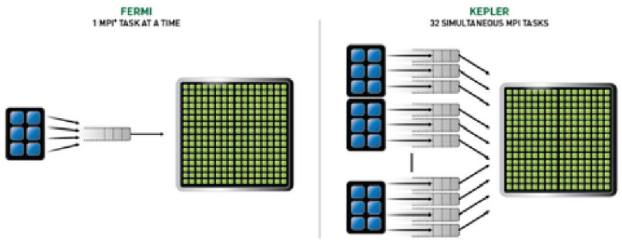

Figura 5: No modelo Fermi (à esquerda), a concorrência é limitada às dependencias *intra-stream* causadas por uma única fila de trabalho no *hardware*. Já no Kepler (à direita), os *streams* podem executar concorrentemente usando filas separadas de tarefas. Extraído de [8].

3. MODELO DE PROGRAMAÇÃO DE GPUS E A ABORDAGEM CUDA

Antes das ferramentas de programação explícita de GPUs, os programadores utilizavam o *OpenGL* [9]. Trata-se de uma especificação livre para a criação de gráficos, sendo muito utilizada em ambientes UNIX e sistemas operacionais da Apple. Porém, sua metodologia de programação é complicada e fácil de gerar erros. Para eliminar essas dificuldades, a NVidia criou a plataforma CUDA [10], baseada nas linguagens C/C++, o que tornou a plataforma altamente aceita por programadores familiarizados com a linguagem.

Em programação de CPUs, costuma-se agrupar dados a serem utilizados próximos temporalmente em estruturas, de forma que fiquem próximos espacialmente. Por exemplo, se for feita a soma entre dois vetores x e y, armazenando o resultado em z, cria-se um vetor de estruturas contendo o valor de x, y e z dos elementos correspondentes. Já em programação de GPUs é aconselhável utilizar o formato original dos vetores. Como a soma é executada em paralelo em diversas *threads*, essa organização permite carregar vários elementos de um único vetor com apenas uma transferência, diminuindo, dessa forma, o gargalo.

Em GPUs, a memória é um gargalo considerável. A título de exemplo, na NVIDIA GeForce 285, sete operações de ponto flutuante podem ser executadas por um núcleo de processamento ao mesmo tempo que um byte demora para ser transferido da memória externa para a GPU [11]. Por esse motivo, deve-se organizar a memória de forma que o acesso possa ser feito através de um único bloco contínuo [12]. A seguir apresentaremos alguns detalhes da abordagem CUDA.

3.1 Características da programação para CUDA

Em CUDA, a função a ser executada na GPU recebe o nome de *kernel*. Essa função é responsável por acessar o *hardware* onde ela é executada N vezes em paralelo em N diferentes CUDA *threads*, diferentemente de uma função C comum. Essa função pode receber argumentos como valores ou ponteiros para memórias globais, locais e também possui uma série de constantes definidas que permite uma *thread* identificar qual elemento deve ser processado por ela. Uma função *kernel* é definida utilizando a declaração _global_ e o número de *threads* que serão executadas.

Cada *thread* tem sua ID, que a identifica como sendo única e é acessível pelo *kernel* através da variável *threaIdx*. O comando *blockIdx.x* disponibiliza o identificador único do bloco da *thread* atual. A variável *threadIdx.x* possui o identificador único da *thread* atual dentro do bloco e *blockDim.x* é a dimensão do bloco atual. Temos, dessa forma, um identificador único de uma *thread* que permite a ela identificar que elemento processar. Esse identificador é calculado da seguinte maneira:

$$ID_{thread} = blockIdx.x * blockDim.x + threadIdx.x, \quad (1)$$

Cada uma dessas constantes podem ter valores em x, y e z, facilitando o trabalho do programador e sendo organizadas internamente da maneira mais eficiente. O Algoritmo 1 mostra o exemplo de um *kernel* desenvolvido em CUDA.

Algoritmo 1 Exemplo de kernel desenvolvido em CUDA. Adaptado de [13]

```
_global_ void VecAdd(float *A, float *B, float *C)
{
int i=threadIdx.x;
C[i]=A[i]+B[i]
}
int main()
{

//Chamada ao kernel com N threads, o sinal <<<>>> é
//Necessário para marcar a chamada do host (CPU) para
//o device (GPU)

VecAdd<<< 1, N >>>(A,B,C);
}
```

Os blocos gerados ao chamar um *kernel* são entregues ao gerenciador dentro de um multiprocessador, sendo distribuídos entre os multiprocessadores pelo gerenciador da GPU para manter o nível de carga igual. Internamente, os blocos ainda são separados em *warps*, cada um com 32 *threads*, que são controlados pelo gerenciador dentro do multiprocessador. Essa hierarquia é mostrada na Figura 6.

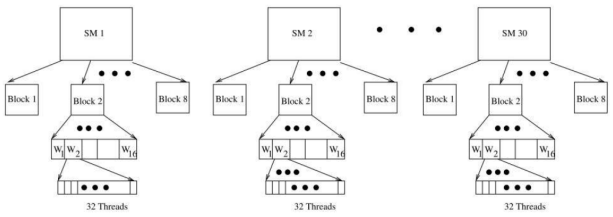

Figura 6: Hierarquia das *threads* segundo o modelo CUDA. Extraído de [14]

Portanto, a quantidade de *threads* utilizadas é o produto do número de blocos pelo tamanho de cada bloco, podendo ser maior do que o número de elementos a serem processados. Pode-se pensar que aumentar a granularidade dos blocos seria uma saída, mas em geral prefere-se utilizar uma granularidade grosseira e permitir que algumas *threads* não

executem computação significativa, pois isso facilita o controle por parte do gerenciador. Para isso, após identificar o elemento a ser processado pela *thread*, é verificado se ele está dentro do limite dos vetores utilizados, senão essa *thread* simplesmente não executa comando algum no vetor.

A variável *threadIdx* é um componente vetorial com 1, 2 ou 3 dimensões em seu índice, formando um bloco com 1, 2 ou 3 dimensões. Isso fornece uma maneira natural para se chamar os elementos computacionais em um certo domínio como vetor, matriz ou volume. O índice da *thread* e seu *threadID* se relacionam de uma maneira direta. Para um bloco de uma dimensão, eles são os mesmos. Para blocos de duas dimensões (Dx, Dy), a *thread* ID do índice da *thread* (x, y) é (x + yDx). Para 3 dimensões ela é (x + yDx + zDxDy). O número de blocos é limitado, já que eles residem no mesmo processador e possuem tamanho de memória limitado. Nas atuais GPUs, o bloco de *thread* pode conter até 2048 *threads*. Os blocos são organizados em grades com uma ou duas dimensões. O número de blocos de *threads* em uma grade é geralmente dito pelo seu tamanho de dados a serem processados ou pelo número de processadores no sistema no qual pode exceder tranquilamente.

Cada *thread* possui acesso a três níveis diferentes de memória, sendo cada um mais lento que o outro. Em primeiro lugar está a memória privada da *thread*, que não pode ser compartilhada entre *threads* e representa tanto trechos da memória interna de um multiprocessador quanto registradores. Em segundo plano, está a memória compartilhada por bloco. Todas as *threads* em um mesmo bloco possuem acesso a essa memória que fica no multiprocessador, sendo portanto bastante rápida. No terceiro nível está a memória global, que é bastante lenta. Sua utilização deve ser feita somente em caso de extrema necessidade. A Figura 7 mostra essa hierarquia e seus acessos.

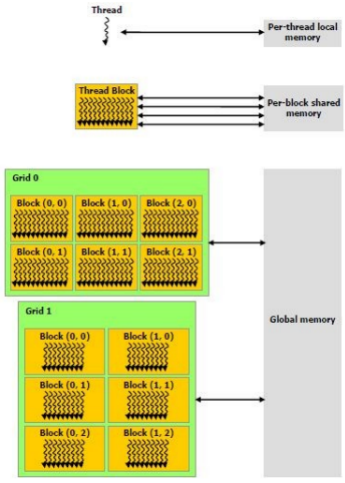

Figura 7: Acessos das *threads*, blocos e grades às memórias da GPU. Extraído de [15].

Com CUDA, pode-se criar um número de *threads* muito maior do que o suportado pelo hardware. Para solucionar esse problema, um controlador de software gerencia e cria as *threads* que serão executadas. Como não se tem controle sobre o gerenciador de *threads*, não se pode afirmar que uma *thread* executará primeiro, já que deve-se utilizar funções de sincronização dentro do *kernel* somente quando necessário, fazendo com que seja criada uma barreira para todas as *threads* até que todas cheguem ao ponto de sincronização.

Para programar com o CUDA, deve-se seguir alguns comandos e passos. A invocação do *kernel* é, conforme visto no Algoritmo 1, utilizando <<<>>>. Depois, definem-se as variáveis como a *threadIdx.x* ou *blockIdx.x*, dependendo do que se deseja fazer. Para se alocar a memória na GPU, primeiramente usa-se a alocação da memória no host (CPU) utilizando o comando em C *malloc()*. Depois, aloca-se a GPU utilizando o *cudaMalloc()*. Em seguida, copia-se o que está na CPU para a GPU utilizando o comando *cudaMemcpy()*. Assim que essas variáveis forem alocadas, deve-se liberar a memória para seu futuro uso. Em C temos comando *free()*, e no CUDA, temos o *cudaFree()*. As *threads* precisam ser sincronizadas se estão trabalhando em um bloco e precisam trabalhar com dados de um iteração anterior. Para isso, utiliza-se a função *__synchthreads()* para se colocar uma barreira até que todas as *threads* de um mesmo bloco terminem suas tarefas.

Como visto nesta Seção, CUDA foi uma grande inovação para a computação paralela, mas foi projetada pela NVIDIA somente para as suas GPUs. Porém, também há uma tendência para programação paralela em CPUs através da *OpenMP* que, a partir de comandos simples, possibilita paralelizar um trecho do programa em CPUs *multicores*. Outra alternativa é o *OpenCL*, que teve sua primeira implementação em GPU em 2008. O OpenCL possui características de programação parecidas com CUDA, como a separação em blocos e programação de *kernels*, mas com as vantagens de ser um padrão aberto e executar tanto em CPUs quanto em GPUs. Como o foco deste artigo é apenas a GPGPU com CUDA, mais detalhes sobre a *OPenCL* podem ser encontrados em [16].

4. APLICAÇÕES DA PLATAFORMA CUDA

Para se ter ideia da popularidade e aplicabilidade das GPUs CUDA NVIDIA, há atualmente 567 universidades que possuem cursos de CUDA em seu catálogo [17]. A página CUDA *Zone* registra 1296 aplicações utilizando GPU [18], e a conferência de tecnologia de GPUs de 2010 contava com mais de 100 pôsteres de pesquisa [19]. São números impressionantes, principalmente se considerarmos que a primeira versão da linguagem foi publicada há seis anos. Nessa Seção serão apresentados alguns trabalhos acadêmicos que utilizaram CUDA, ressaltando a importância que essa abordagem teve na queda do tempo de processamento.

D'Agostino *et al* [20] implementaram a abordagem SSAKE para montagem de genoma, sobreposição de grafos e *hashing* de sequências. Essa abordagem se baseia na potencialização da informação de leituras de pequenas sequências ao montá-las em sequências contínuas. Nos experimentos foi utilizada uma estação de trabalho Intel® i5-750 e uma placa de vídeo NVIDIA GTX480. Foi realizado o sequenciamento do genoma da bactéria *Escherichia coli* K-12, cuja entrada é representada por mais de 2 bilhões de sequências em um arquivo de 202MB. As diferenças de tempo de execução sequencial e em CUDA (paralela) estão dispostas na Tabela 1.

Etapa/Algoritmo	Sequencial	CUDA
Aquisição de dados	22.9	24.2
Busca de k-mer	78.6	21.5
Restrição de Consensus	1.8	3.0
Deleção da sequência e criação da saída	23.8	4.8
Tempo Total	129.2	53.5

Tabela 1: Tempo de execução em segundos das etapas do algoritmo SSAKE para o sequenciamento do genoma da bactéria *Escherichia coli K-12*. Adaptado de [20].

Karunadasa e Ranasinghe [21] utilizaram CUDA em conjunto com a interface de passagem de mensagens (sigla MPI em inglês) para acelerar aplicações de alto desempenho. Essas duas abordagens foram utilizadas em dois algoritmos bem conhecidos (Algoritmo de *Strassens* e Algoritmo do Gradiente Conjugado) executados em clusters GPU. Os autores comprovaram que foi possível obter um desempenho superior utilizando funções MPI como mecanismos de distribuição de computação entre os nós do *cluster* GPU e CUDA como o mecanismo de execução principal. Essa abordagem em *clusters* GPU permite ao programador conectar nós de GPUs através de redes de alta velocidade, visualizar cada nó GPU separadamente e executar diferentes componentes de um programa em vários nós de GPU. Esses programas foram executados em *clusters* CPU de 6 nós e de GPU com 2 nós. Os nós do *cluster* GPU utilizam a placa de vídeo NVIDIA 8800 GT com 768MB de memória. Cada programa foi executado 10 vezes e o tempo médio de execução foi calculado, conforme mostrados nos gráficos das Figuras 8 e 9 para um dos algoritmos testados.

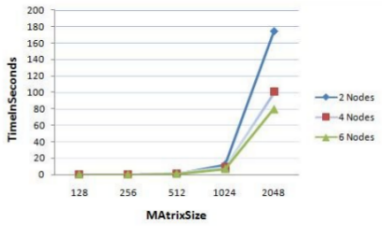

Figura 8: Tempos de execução do algoritmo de *Strassen* em diferentes configurações de clusters CPU. Extraído de [21]

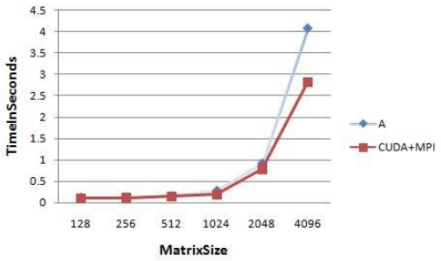

Figura 9: Tempos de execução do algoritmo de *Strassen* em um simples nó GPU (em azul) e no cluster GPU CUDA+MPI (em vermelho). Extraído de [21]

Veronese e Krohling [22] implementaram o algotitmo de otimização de enxame de partículas (sigla PSO em inglês) usando CUDA em linguagem C. O algoritmo foi testado em um conjunto de seis problemas de otimização bem conhecidos em um *benchmark*. O tempo de computação foi comparado com o mesmo algoritmo, dessa vez implementado em C e Matlab. Os resultados demonstram que o tempo de computação pode ser significativamente reduzido utilizando o CUDA em C, conforme mostrado na Figura 10.

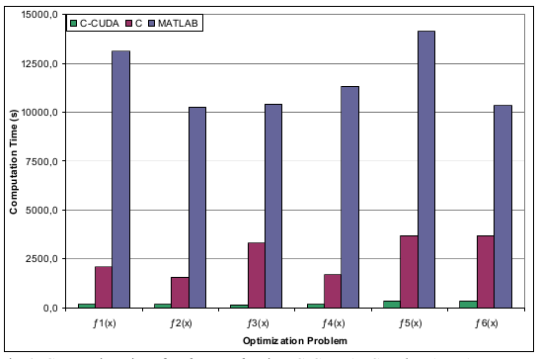

Figura 10: Tempos de execução do algoritmo de PSO usando C-CUDA, C e Matlab. Extraído de [22]

Pan *et al* [23] propuseram a implementação de diversos algoritmos existentes de segmentação de imagens médicas utilizando CUDA. O tempo de execução destes foi comparado com os executados em CPUs. Foram utilizados os algoritmos de segmentação por crescimento de regiões e por *Watershed*. Essa comparação pode ser vista nas Tabelas 2 e 3.

Hardware	Imagem de abdômen	Imagem de cérebro
Geforce 8500 GT	12.875	2.435
Geforce 6800	13.62	4.42
Pentium	21.034	não implementado

Tabela 2: Tempo de execução em segundos do algoritmo de segmentação de imagens por crescimento de regiões em duas GPUs e uma CPU. Adaptado de [23].

Hardware	Imagem de abdômen	Imagem de cérebro
Geforce 8500 GT	155.965	45.223
Pentium 4	não implementado	115

Tabela 3: Tempo de execução em segundos do algoritmo de segmentação de imagens por Watershed em uma GPUs e uma CPU. Adaptado de [23].

Conforme mostrado nessa Seção, o poder computacional das GPUs CUDA/NVIDIA estão melhorando o tempo de execução de algoritmos tradicionais. Uma tendência atual é utilizar CUDA em GPUs de máquinas de alto poder computacional, melhorando ainda mais o tempo de resposta desses algoritmos. Um exemplo de máquina deste tipo é a *Tinhae-1A*, localizada na China. Esse supercomputador é composto por 7168 GPUs, gerando um poder computacional de 2,566 petaflops e um consumo de energia equivalente a 4.04 MW,

posicionando esse supercomputador em segundo lugar nos 500 computadores mais rápidos do mundo em Novembro de 2011 [24].

5. CONCLUSÃO

Nesse artigo foi apresentada a abordagem da NVIDIA para GPGPU através da plataforma CUDA. Foi discutida a arquitetura de GPUs, enfatizando suas diferenças com a CPU e também a programação para GPUs através de CUDA. Algumas de suas características foram discutidas, assim como foi apresentada uma possível alternativa a essa abordagem: o *OPenCL*. Também foi mostrada a melhora de desempenho de alguns algoritmos tradicionais da literatura científica de diversas áreas, através da programação com CUDA.

Para o futuro, espera-se melhorar ainda mais a eficiência energética das placas gráficas, gerando menos calor e, desta forma, resultando em melhor desempenho. Um dos impactos que esse aumento na capacidade de processamento pode causar está no desenvolvimento de algoritmos que podem, por exemplo, ajudar na criação de melhores produtos, medicamentos e materiais, devido ao tempo de resposta muito superior ao que se tinha há alguns anos.

6. REFERÊNCIAS

[1] NVIDIA Corporation. GeForce 256- The world's first GPU. `http://www.nvidia.com/page/geforce256.html`, acesso em Maio de 2012.

[2] NVIDIA Corporation. Fast facts. `http://www.nvidia.com/object/fast-facts.html`, acesso em Maio de 2012.

[3] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. Gpu cluster for high performance computing. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, SC '04, pages 47–58, Washington, DC, USA, 2004. IEEE Computer Society.

[4] NVIDIA Corporation. Fermi computer architecture whitepaper. `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`, acesso em Maio de 2012.

[5] Beyond 3D. Nvidia g80: Architecture and gpu analysis. `http://www.beyond3d.com/content/reviews/1`, acesso em Maio de 2012.

[6] Beyond 3D. Nvidia gt200 gpu and architecture analysis. `http://www.beyond3d.com/content/reviews/51`, acesso em Maio de 2012.

[7] John Nickolls and William J. Dally. The gpu computing era. *IEEE Micro*, 30(2).

[8] NVIDIA Corporation. Kepler gk110 architecture whitepaper. `http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`, acesso em Maio de 2012.

[9] Khrounos Groups. Opengl. `www.opengl.org`, acesso em Maio de 2012.

[10] NVIDIA Corporation. Cuda zone. `http://www.nvidia.com/object/cuda_home_new.html`, acesso em Maio de 2012.

[11] T.M. Aamodt. Architecting graphics processors for non-graphics compute acceleration. In *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pages 963 –968, Agosto de 2009.

[12] Jonathan Cohen and Michael Garland. Solving Computational Problems with GPU Computing. *Computing in Science and Engineering*, 11(5):58–63, 2009.

[13] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010.

[14] Kishore Kothapalli, Rishabh Mukherjee, M. Suhail Rehman, Suryakant Patidar, P. J. Narayanan, and Kannan Srinathan. In *Proceedings of 2009 International Conference on High Performance Computing*.

[15] NVIDIA Corporation. *CUDA Programming Guide Version 4.2*. Santa Clara, California, 2008.

[16] Khrounos Groups. Opencl. `www.khronos.org/opencl`, acesso em Maio de 2012.

[17] NVIDIA Corporation. Cuda courses map. `http://research.nvidia.com/content/cuda-courses-map`, acesso em Maio de 2012.

[18] NVIDIA Corporation. Cuda community showcase. `http://www.nvidia.com/object/cuda-apps-flash-new.html`, acesso em Maio de 2012.

[19] Gpu technology conference. `http://www.gputechconf.com/object/gtc-posters-2010.html`, acesso em Maio de 2012.

[20] D. D'Agostino, A. Clematis, A. Guffanti, L. Milanesi, and I. Merelli. A cuda-based implementation of the ssake genomics application. In *20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, pages 612 –616, Fevereiro de 2012.

[21] N.P. Karunadasa and D.N. Ranasinghe. Accelerating high performance applications with cuda and mpi. In *International Conference on Industrial and Information Systems*, pages 331 –336, Dezembro de 2009.

[22] L. de P. Veronese and R.A. Krohling. Swarm's flight: Accelerating the particles using c-cuda. In *IEEE Congress on Evolutionary Computation.*, pages 3264 –3270, Maio de 2009.

[23] Lei Pan, Lixu Gu, and Jianrong Xu. Implementation of medical image segmentation in cuda. In *International Conference on Information Technology and Applications in Biomedicine*, pages 82 –85, Maio de 2008.

[24] Top500 list - november 2011 (1-100). `http://www.top500.org/list/2011/11/100`, Maio de 2012.