A evolução da GPGPU: arquitetura e programação

Conrado S. Miranda (070498)
Laboratório de Identificação e Controle de Sistemas Dinâmicos
Departamento de Projeto Mecânico
Faculdade de Engenharia Mecânica, UNICAMP
13083-860 - Campinas, SP - Brasil
miranda.conrado@gmail.com

ABSTRACT
Este artigo faz um estudo da evolução das GPUs utilizadas para propósito geral. Elas possuem grande aplicação em, mas não limitadas a, programas científicos. Várias mudanças na arquitetura e programação ocorreram durante os anos para satisfazer essa necessidade, mas sem distanciar da capacidade de geração de gráficos, que ainda é sua principal importância. Serão apresentadas a história da evolução destes dispositivos tanto do lado do projetista quanto do usuário.

Keywords
computação paralela, GPU, cuda, opencl

1. INTRODUÇÃO
Há vários anos, projetistas de computadores tentam manter a evolução do processamento de acordo com a lei de Moore. Essa lei prevê que a quantidade de transistores em um chip dobra a cada 18 meses. Esse aumento se deve à criação de tecnologias para diminuir os transistores e está diretamente ligada à velocidade. Nas CPUs, esse aumento se deu em grande parte na melhoria do controle para executar programas sequenciais mais rápido através de memórias cache e despacho de várias instruções por ciclo.

Poucos anos atrás, o crescimento da velocidade dos processadores diminuiu significantemente, o que levou programadores de computação paralela a procurar uma nova alternativa. Em pouco tempo descobriram a grande capacidade computacional das GPUs. GPU (Graphics Processing Unit) é um dispositivo muito mais novo do que as CPUs, possuindo apenas 13 anos de existência, tendo sido criadas em 1997 inicialmente na forma de aceleradores gráficos integrados. Seu objetivo na época da criação era gerar os gráficos que seriam mostrados nos monitores, retirando essa carga do processador. Sua característica marcante é a capacidade de processar trechos de código em paralelo. Uma análise da GPU do ponto de vista do pipeline de geração de gráficos é feita em [11] e não será tratada aqui.

Com o início da utilização da GPU para programação de propósito gerais, fabricantes perceberam o potencial desse nicho de mercado e começaram a desenvolver facilidades tanto na arquitetura quanto na própria programação. Esses desenvolvimentos levaram a uma grande utilização dessas placas em computação paralela a baixo custo, já que grande parte do custo de desenvolvimento é subsidiado pelas empresas de jogos, que procuram placas cada vez mais poderosas. Isso levou a criação do termo GPGPU (General Purpose Graphics Processing Unit), dispositivo que será analisado nesse artigo. A figura 1 apresenta uma linha do tempo das características das GPUs, apresentando características significativas incluídas nas primeiras placas de cada geração. Todas as placas listadas foram produzidas pela NVIDIA.

O artigo é organizado da maneira a seguir. A seção 2 apresenta uma análise da arquitetura da GPGPU. A seção 3 trata das técnicas de programação. A seção 4 mostra uma comparação entre CPUs e GPUs, analisando a performance. Um olhar para o caminho que deve ser seguido nessa evolução é apresentado na seção 5.

2. ARQUITETURA DE UMA GPU
Enquanto CPUs dedicam uma grande quantidade de seus circuitos ao controle, uma GPU foca mais em ALUs (Arithmetic Logic Units), o que as torna bem mais eficientes em termos de custo quando rodam um software paralelo. Em 2001, as GPUs passaram de um pipeline de função fixa para geração de imagens para um pipeline altamente programável, aumentando a capacidade de gerar efeitos visuais customizados através de sombreamento programável. Essa possibilidade de programar a GPU atraiu pessoas interessadas em executar programas altamente paralelos e de propósito geral. Essa prática se tornou cada vez mais comum, o que fez com que em 2006 fosse criada uma GPU com hardware explícito para programação de propósito geral. O primeira modificação foi a possibilidade de os programas de sombreamento escrever em qualquer parte da memória. Antes disso, os programas só podiam escrever em locais específicos para textura da memória, o que trazia grande limitações aos programas.

GPUs possuem multiprocessadores com vários núcleos que aplicam a estrutura de uma instrução para vários dados (SIMD). O uso de SIMD melhora a performance por unidade de custo, permitindo que vários dados sejam processados de forma igual em paralelo simultaneamente. Para melhorar ainda mais a performance, GPUs atuais utilizam uma extensão desse conceito que é única instrução e múltiplas threads (SIMT), o que significa que a mesma instrução é executada em várias threads diferentes, ajudando a manter o pipeline ocupado. Esse agrupamento de threads que executam a mesma instrução recebe o nome de warp.

Table 1. NVIDIA GPU technology development.

Date	Product	Transistors	CUDA cores	Technology
1997	RIVA 128	3 million	—	3D graphics accelerator
1999	GeForce 256	25 million	—	First GPU, programmed with DX7 and OpenGL
2001	GeForce 3	60 million	—	First programmable shader GPU, programmed with DX8 and OpenGL
2002	GeForce FX	125 million	—	32-bit floating-point (FP) programmable GPU with Cg programs, DX9, and OpenGL
2004	GeForce 6800	222 million	—	32-bit FP programmable scalable GPU, GPGPU Cg programs, DX9, and OpenGL
2006	GeForce 8800	681 million	128	First unified graphics and computing GPU, programmed in C with CUDA
2007	Tesla T8, C870	681 million	128	First GPU computing system programmed in C with CUDA
2008	GeForce GTX 280	1.4 billion	240	Unified graphics and computing GPU, IEEE FP, CUDA C, OpenCL, and DirectCompute
2008	Tesla T10, S1070	1.4 billion	240	GPU computing clusters, 64-bit IEEE FP, 4-Gbyte memory, CUDA C, and OpenCL
2009	Fermi	3.0 billion	512	GPU computing architecture, IEEE 754-2008 FP, 64-bit unified addressing, caching, ECC memory, CUDA C, C++, OpenCL, and DirectCompute

Figure 1: Linha do tempo da evolução das GPUs. Imagem retirada de [10].

Para manter o pipeline ocupado com várias threads, GPUs utilizam também granulação fina na execução, o que significa que a cada clock uma unidade de processamento termina a execução de uma instrução em uma thread diferente, executando ciclicamente uma instrução para cada thread. Essa técnica evita a necessidade do forwarding que ocorre nos pipelines das CPUs, o que diminui a quantidade de circuito para controle necessária. Além disso, ela é utilizada para mascarar acessos à memória global, que é uma operação extremamente lenta em GPUs. Para essa técnica ser eficiente, devemos ter uma aplicação com alto grau de paralelismo, mantendo o pipeline ocupado. Como isso nem sempre acontece e os stalls do pipeline são proibitivos, projetistas resolveram colocar memórias cache para acelerar o acesso a memórias, tecnologia que era vista somente em CPUs. Com o objetivo de melhorar ainda mais o acesso à memória, GPUs reordenam os acessos para que sejam feitos de forma mais rápida, através de agrupamento para acessar um grande bloco da memória de uma única vez, reduzindo o tráfego e latências. Se várias threads em um warp acessam posições contínuas de memória, uma única requisição é necessária, aumentando a velocidade. Para se ter idéia da grandeza do problema do acesso à memória, na NVIDIA GeForce 285 GTX, quase 7 operações de ponto flutuante podem ser executadas por um núcleo no mesmo tempo que um byte demora para ser transferido da memória externa para a GPU [4].

Como cada multiprocessador executa várias threads, eles possuem uma memória interna compartilhada de pequena e um grande banco de registradores dos quais cada thread é dona de alguns. Isso aumenta muito a velocidade da troca de informações entre threads de um agrupamento e da troca de contexto, já que não é preciso salvar os registradores. Mas o fato de a memória local ser compartilhada faz com que possa ocorrer conflitos, quando mais threads do que o possível tentam ler ou escrever da memória, que possui número de portas menor do que o número de threads que podem ser executadas.

Em um modelo simples de multiprocessador, a técnica SIMT geraria problemas em instruções de desvio condicionais. Para resolver esse problema, o hardware possui um controlador que é capaz de mascarar resultados de instruções executadas em threads, permitindo que uma warp continue a execução mesmo se houverem resultados divergentes para um desvio condicional. Para isso, basta que os resultados das threads que não teriam tomado o caminho sequencial não gerem resultados permanentes, seja gravando em registradores ou na memória. Depois, o trecho de código correspondente ao desvio tomado é executado, impedindo que as outras threads não gerem resultados. Isso faz com que algum tempo de processamento seja perdido, o que levou à criação de técnicas em software para melhorar a performance e serão descritas na seção 3. Propostas para melhora da performance envolvem também instruções condicionais, o que eliminaria vários desvios condicionais, e warps dinâmicas, que seriam capazes de se dividir quando suas threads tivessem resultados divergentes em desvios condicionais, o que eliminaria a necessidade de mascarar seus resultados.

A mérito de exemplo, apresentamos a arquitetura da GPU Fermi da NVIDIA apresentada na figura 2, onde cada multiprocessador é mostrado com mais detalhes na figura 3. Ela possui 16 multiprocessadores, cada um com 32 núcleos, totalizando 512 núcleos por chip. Seu gerenciador de threads GigaThread distribui o trabalho para os multiprocessadores de acordo com a carga dinâmica através de agrupamentos de threads, podendo rodar mais de um programa simultaneamente se apropriado. O gerenciador interno do multiprocessador gerencia threads individuais, podendo executar até 1536 threads concorrentes.

Ela possui espaço de endereçamento físico de 40-bits e

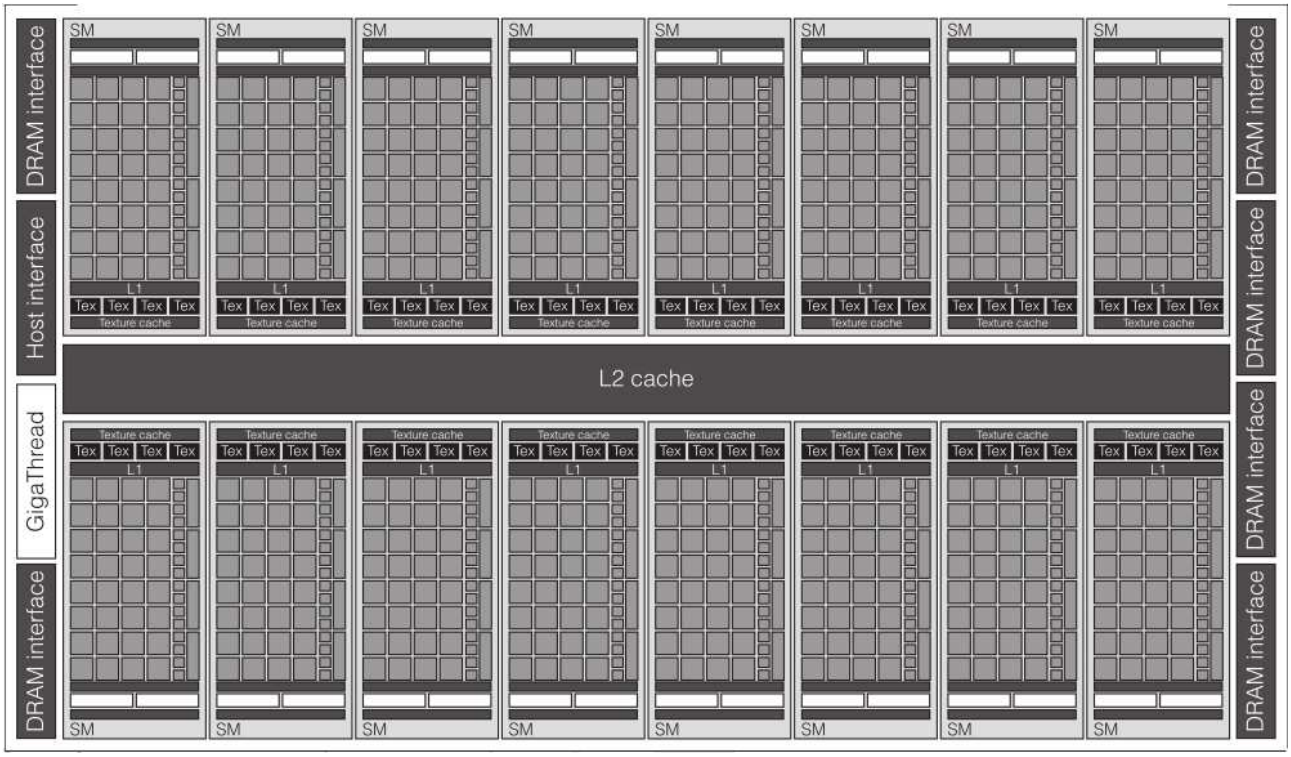

Figure 2: Arquitetura da GPU Fermi. Imagem retirada de [10].

virtual de 64-bits e uma cache L2 unificada entre os multiprocessadores de 768 kbytes, que está conectada a seis interfaces de DRAM e a interface PCIe. Os multiprocessadores possuem cache L1 de dados e memória local configuráveis, totalizando 64 kbytes, podendo ser configuradas para uma ter 16 kbytes e a outra 48 kbytes ou vice-versa, banco de registradores de 128 kbytes, cache de instrução, cache e memória de texturas e 2 grupos gerenciadores de thread e unidades de despacho de instruções. Além disso, a GPU possui também um protetor de memória para aumentar a integridade dos dados, podendo corrigir erros de 1 bit e detectar erros de 2 bits na memória, caches ou registradores. Isso prolonga muito o tempo de vida dos sistemas, o que motiva ainda mais sua utilização para servidores.

Seu conjunto de instruções inclui instruções de inteiro, endereçamento e ponto flutuante de 32 e 64-bits; acesso à memória para leitura, escrita ou atômicos; acesso a superfícies multidimensionais, que facilita a programação; predição de desvios, chamada direta e indireta de funções, utilizada em funções virtuais em C++; e sincronização.

Com 32 núcleos, é capaz de executar até 32 instruções aritméticas por clock, cada núcleo possuindo uma unidade inteira e uma de ponto flutuante. Sua unidade de ponto flutuante inclui, além dos tipos das operações básicas, instrução FMA (Fused Multiply-Add) que possuem o formato $D = A * B + C$ sem perda de precisão. Sua velocidade em operações de 64-bits de ponto flutuante é metade da velocidade em comparação às de 32-bits, valor muito acima do comum. As unidades especiais (SPU), cada multiprocessador possuindo quatro, executam uma rápida aproximação de funções como inversa, raiz quadrada da inversa, seno, cosseno, exponencial e logaritmo de valores de 32-bits.

Essa GPU possui características que há alguns anos eram atribuídas apenas a CPUs como cache e predição de desvios. Ela representa o que há de mais novo no mercado de GPGPU, sendo utilizada em servidores de grande porte, e mostra a grande evolução das GPUs visando aumento da performance de computações de propósito geral.

3. PROGRAMAÇÃO DE GPUS

Com o conhecimento da arquitetura, é possível entender o princípio de programação das GPUs. Como foi dito, a memória é um gargalo importante do sistema. Por isso, devemos organizá-la de tal maneira que o acesso possa ser feito em um único bloco contínuo, como mostrado em [5, 6]. Em programação de CPUs, as pessoas aprendem a agrupar dados que serão utilizados próximos temporalmente em estruturas para que fiquem próximos espacialmente. Com isso, se faremos a soma de dois vetores $z = x + y$, criamos um vetor de estruturas, cada uma contendo o valor de x, y e z do elemento referente. Em programação de GPUs é aconselhável utilizar o formato original, ou seja, vetores separados para x, y e z. Como a soma é executada em paralelo em várias threads, essa organização permite carregar vários elementos de um único vetor com apenas uma transferência, diminuindo o gargalo

Antes de haver ferramentas para programação explícita de GPUs, programadores utilizavam OpenGL [3]. OpenGL é uma especificação livre criada para criação de gráficos, sendo muito utilizada em ambientes *NIX e nos sistemas operacionais da Apple. Como já observado anteriormente, podemos escrever algorítimos que fazem transformação nas imagens durante sua renderização, utilizando para isso uma linguagem auxiliar utilizada para gerar sombras. Para isso, colocamos os dados necessários para a computação no lugar de texturas e utilizamos o programa escrito como se fosse um gerador de sombras para executar o cálculo necessário. Como o processamento original serve para geração de imagens, devemos redimensionar a tela a

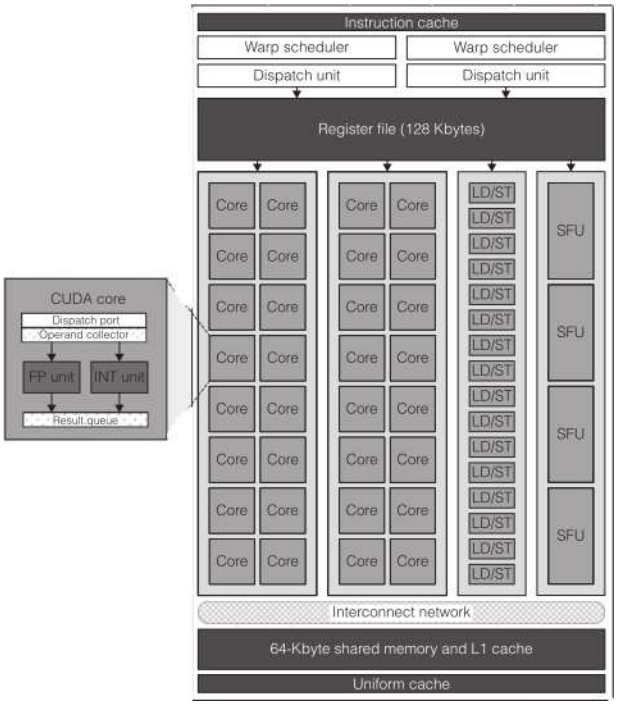

Figure 3: Arquitetura de um multiprocessador da GPU Fermi. Imagem retirada de [10].

ser criada para o tamanho dos dados e utilizar projeção ortogonal, fazendo com que todos os dados sejam executados de forma correta. Com isso, garantimos que todos os pixels serão calculados, pois todos estão sendo exibidos. Para aumentar a flexibilidade, podemos escrever essa função baseada na textura de cores ou RGBA, cada uma com suas dificuldades.

Como deve ter ficado claro, realizar computação desse jeito é extremamente trabalhoso e fácil de ocorrer erros. Por isso, a NVIDIA criou uma linguagem proprietária chamada CUDA [1], baseada nas linguagens C/C++. Por utilizar linguagem semelhante a linguagens familiares a programadores, CUDA deve uma aceitação muito rápida entre os programadores. Atualmente, mais de 300 universidades possuem cursos de CUDA em seu catálogo, a página CUDA Zone registra mais de 1000 aplicações utilizando GPU e a conferência de tecnologia de GPUs de 2009 contava com 91 pôsteres de pesquisa. São números impressionantes, principalmente se considerarmos que a primeira versão da linguagem foi publicada em 2006. Existem extensões com suporte a CUDA para programas científicos de grande porte como MathWorks MatLab, Wolfram Mathematica e National Instruments Lab View, conhecidos por sua lentidão e programação vetorial.

Em CUDA, a função a ser executada na GPU recebe o nome de kernel. Ele pode receber argumentos como valores ou ponteiros para memória globais, locais e possui uma série de constantes definidas que permitem uma thread identificar qual elemento deve ser por ela processado. O comando **blockIdx.x** disponibiliza o identificador único do bloco da thread atual, **threadIdx.x** possui o identificador único da thread atual dentro do bloco e **blockDim.x** é a dimensão do bloco atual. Assim, o cálculo **blockIdx.x*blockDim.x+threadIdx.x** fornece um identificador única a uma thread, o que permite que ela identifique qual elemento deve processar. 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. Um kernel pode ser chamado de um código em C, onde são fornecidos os tamanhos e quantidades dos blocos, além dos argumentos. A figura 4 apresenta o modelo organizacional de threads.

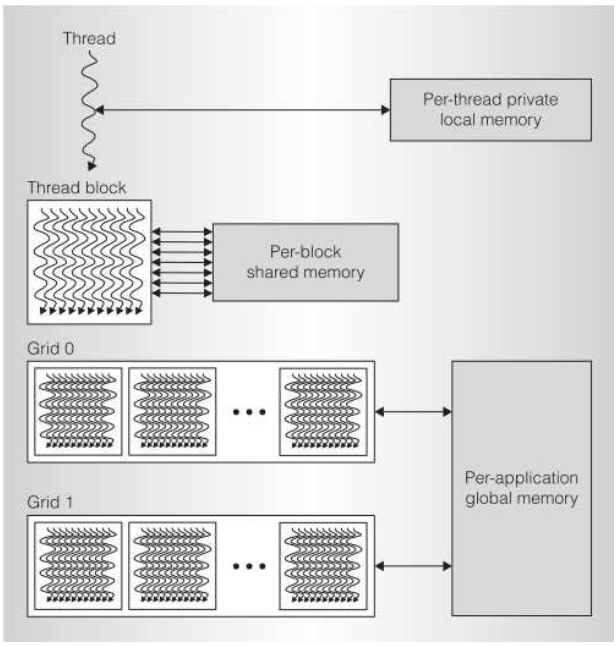

Figure 4: Modelo de execução das threads. Imagem retirada de [10].

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 são ainda separados em warps, cada uma com 32 threads, que são controladas pelo gerenciador dentro do multiprocessador. Essa hierarquia é mostrada na figura 5. 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 é bom, mas em geral prefere-se utilizar um granularidade grosseira e permitir que algumas threads não executem computação significativa, pois isso facilita o controle por parte do gerenciador. Para isso, após identificarmos o elemento a ser processado pela thread, checamos se ele está dentro do limite dos vetores utilizados. Se não, essa thread simplesmente não executa comando algum no vetor.

Cada thread possui acesso a três níveis 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. Em terceiro nível está a memória global, que é extremamente lenta. Sua utilização deve ser feita somente em caso de necessidade.

Com CUDA, pode-se criar um número de threads muito maior do que o suportado pelo hardware. Para isso, um controlador de software gerencia e crias as threads que se-

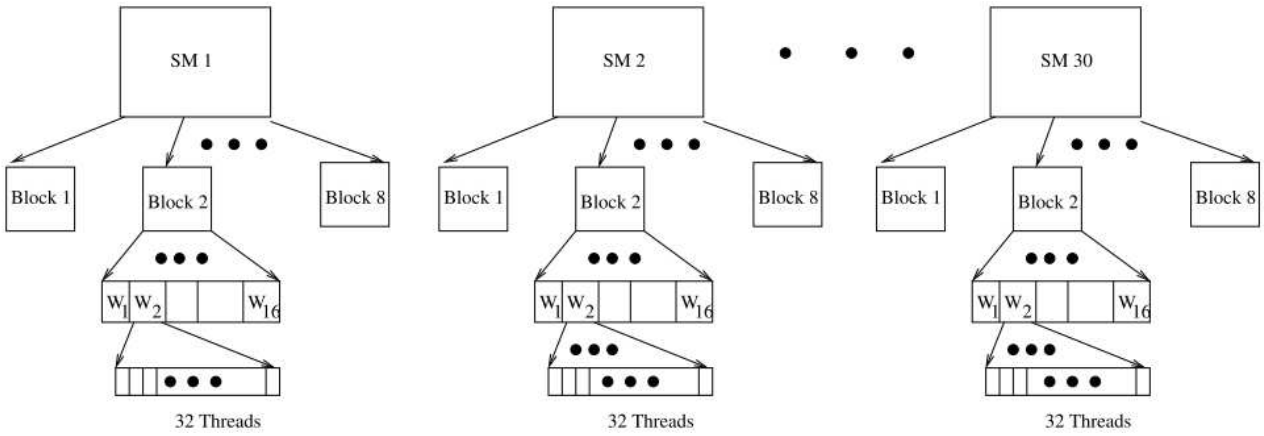

Figure 5: Hierarquia das threads segundo o modelo CUDA. SM é um multiprocessador. Imagem retirada de [12].

rão executadas. Como não se tem controle sobre o gerenciador de threads, não podemos afirmar que uma thread executará primeiro, tendo que utilizar funções de sincronização dentro do kernel quando necessário, fazendo com que seja criada uma barreira para todas as threads até que todas cheguem ao ponto de sincronização.

Alguns programas de processamento de dados possuem a característica de trabalharem com streams [7], ou conjunto contínuo de dados. Como a memória é lenta, isso poderia se tornar um grande problema. Portanto, foi desenvolvida uma técnica que utiliza dois kenels, um que executa a computação desejada e outro que realiza a transferência dos dados para a memória interna da placa. Assim, enquanto a computação ocorre em um grupo de dados, ocorre a transferência dos próximos dados a serem processados, tornando o processo muito mais rápido e mascarando a lentidão da memória.

Durante o estudo de arquitetura na seção 2, foi dito que o problema relacionado ao resultado diferente de desvios em threads de um mesmo warp poderia ser resolvido em software. Essa técnica é explicada com detalhes em [10], mas baseia-se na transformação do código em uma máquina de estados. Quando ocorre um desvio condicional, ao invés de realizar blocos de instruções que teriam seus resultados mascarados em algumas threads nos trechos que não deveriam por ela serem executados, as threads recebem um estado novo dependendo do resultado do desvio. Assim, warps se tornam agrupamentos de threads que estão no mesmo estado, permitindo threads que percorreram o mesmo caminho se separar em warps diferentes e threads de warps originais diferentes que possuem o mesmo estado serem agrupadas. Isso torna a computação mais eficiente, reduzindo o desperdício de recurso e melhorando a performance.

Como pode ser visto, CUDA foi uma grande inovação para a computação paralela, mas só funciona em GPUs da NVIDIA. Como também há uma tendência de programação paralela em CPUs, através da OpenMP que, através de comandos simples possibilita paralelizar um trecho do programa em CPUs com múltiplos núcleos, o Khronos Groups, criador do OpenGL, criou o OpenCL [2], que teve sua primeira implementação em GPU em 2008. O OpenCL possui características de programação parecidas com o CUDA, com sua separação em blocos e programação de kernels, mas com algumas vantagens: não é um padrão exclusivo da NVIDIA, sendo aberto e permitindo

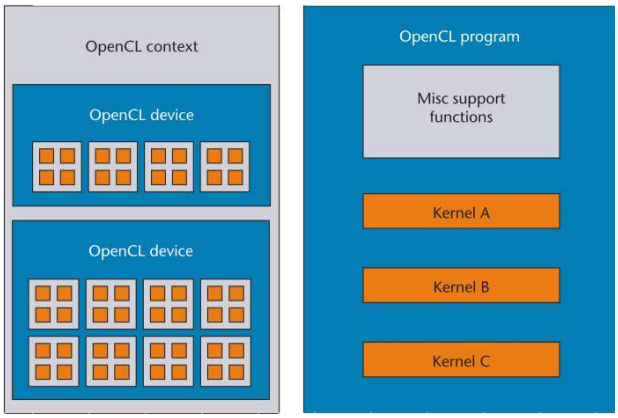

Figure 6: Modelo de programa em OpenCL. Imagem retirada de [13].

que qualquer fabricante ou usuário de um determinado processador, independentemente da arquitetura, possa rodar código escrito em OpenCL; funciona tanto em GPUs quanto CPUs, permitindo mesclar vários tipos de arquiteturas, permitindo inclusive que os processadores estejam em rede e em computadores diferentes; e permite a utilização de código vetorial para aumentar a velocidade, executando explicitamente várias operações em paralelo.

Por ser um padrão aberto e focado em computação paralela em qualquer dispositivo, um programa OpenCL é mais genérico do que um programa equivalente em CUDA, podendo rodar em quaisquer arquiteturas conectadas. Assim, um programa em OpenCL é composto pelo programa de fato com um ou mais kernels e funções auxiliares, como utilitários de sincronização, pelo lado do programador e por um contexto que engloba todos os dispositivos com poder computacional disponíveis, como mostra a figura 6. Ao se iniciar um programa em OpenCL, devemos ter apenas o sistema para o qual o programa foi desenvolvido rodando, pois ter algum dispositivo mais lento pode limitar o desempenho, pois o sistema é nivelado por baixo. Isso pode, por exemplo, diminuir o máximo de memória disponível para a execução do programa.

Podemos otimizar um programa OpenCL para uma arquitetura específica, que seria o local onde pretendemos rodar a aplicação. Isso não impede que ele rode em outros dispositivos, mas faz com que ele rode mais lentamente. Além

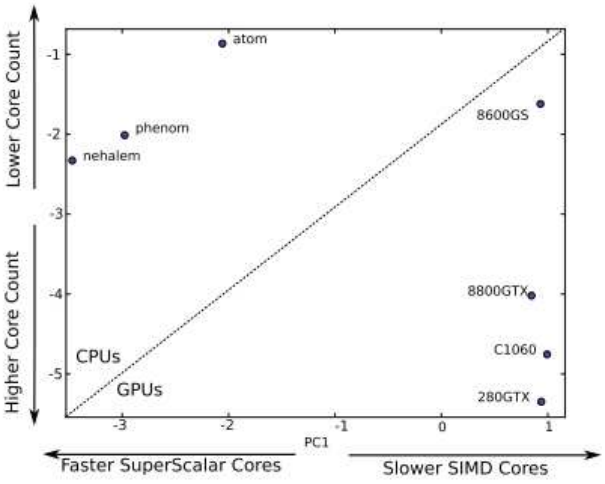

Figure 7: Comparação dos modelos entre GPU e CPU. CPUs estão à esquerda e GPUs estão à direita. Imagem retirada de [8].

disso, dispositivos diferentes podem não ter compatibilidade binária, necessitando de recompilação para rodar em diversas arquiteturas. Por esses dois motivos, OpenCL incentiva compilação em tempo de execução. Para isso, o programa é compilado para uma linguagem intermediária como acontece nos compiladores, mas os passos finais de otimização e geração de código dependentes da arquitetura ocorrem logo antes de executar o programa. Com isso, o programa poderá tirar proveito das características específicas de hardware e software que está rodando, incluindo atualizações realizadas entre a criação do código e sua execução, sendo muito mais rápido. Em aplicações com alto grau de paralelismo, esse tempo extra gasto na compilação é muito menor do que seria gasto em processamento caso o programa não fosse otimizado para a arquitetura de destino, fazendo com que o desempenho total aumente.

4. COMPARAÇÃO ENTRE CPUS E GPUS E ANÁLISE DE PERFORMANCE

Uma técnica de modelagem da performance é apresentada em [8], onde são atribuídos valores a determinadas características consideradas principais na descrição de um processador. A figura 7 mostra um gráfico cujo eixo horizontal é a velocidade do processador, estando os processadores mais rápidos para esquerda, e o eixo vertical é a quantidade de núcleos, estando os processadores com mais para baixo. Esse gráfico mostra a clara distinção entre arquiteturas de GPUs e CPUs. Ao passo que CPUs visam ter menos núcleos mais rápidos, focadas em execução serial de programas e baixa latência, GPUs são projetadas para ter um grande número de núcleos mais lentos, focados em throughput massivo e sendo ideias para computação paralela.

Uma implementação do algoritmo que calcula os K vizinhos mais próximos (KNN), muito utilizado atualmente em servidores para classificação de dados em categorias, foi desenvolvido em CUDA em [9], que também fez comparações com algoritmos sequenciais. A figura 8 foi feita com 32768 objetos de referência, oito dimensões e escolhendo os sete vizinhos mais próximos. Os dados mostram que o quick sort é mais rápido do que o insertion sort, mas nenhum deles se compara ao algoritmo executado em CUDA que é muito mais veloz.

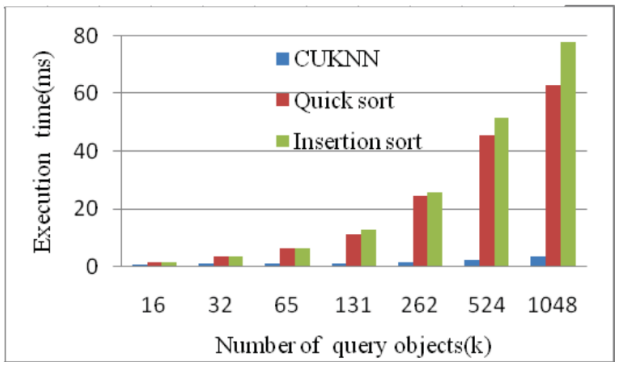

Figure 8: Comparação da performance do KNN utilizando CUDA para cálculo e ordenação, algoritmo clássico para cálculo e quick sort para ordenação e insertion sort, variando-se o número de elementos analisados simultaneamente. Imagem retirada de [9].

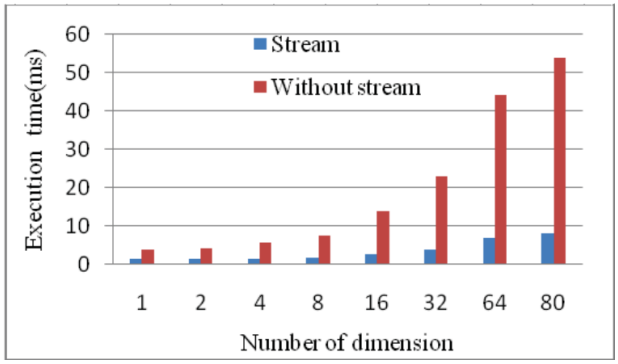

Figure 9: Comparação da performance do algoritmo KNN utilizando ou não a técnica de stream, variando-se a dimensão da análise. Imagem retirada de [9].

Na figura 9, uma análise utilizando a técnica de stream, descrita na seção 3 é feita com 262144 objetos de referência e escolhendo os sete vizinhos mais próximos. O resultado mostra que a técnica realmente é válida para "esconder"os lentos acessos à memória, continuando a execução dos cálculos enquanto se espera a memória, que representa em torno de 99% do tempo de execução no programa que não utiliza streams, comprovando a teoria de que o gargalo em GPUs é a memória.

5. CONCLUSÃO

Como pôde ser visto, as GPUs estão se tornando cada vez mais para propósitos gerais enquanto as CPUs estão suportando cada vez mais paralelismo. Apesar de estarem cada vez se semelhantes, uma nunca irá substituir completamente a outra por terem propósitos de uso diferentes, mas já existem idéias de mesclá-las, tornando um único processador ou sistema. Além disso, as GPUs terão circuito para manter coerência entre caches, técnica atualmente já empregada em processadores com vários núcleos, como já anunciado pela Intel na arquitetura Larranbee. Isso levará a um menor tempo no acesso à memória devido à utilização da cache, mas pode impactar negativamente o processamento paralelo devido ao aumento dos cache miss.

Esse artigo apresentou uma perspectiva histórica tanto do hardware quanto do software relacionados às GPUs. Apresentou também as mudanças necessárias na arquitetura e técnicas de programação para que computações paralelas pudessem ser executadas de maneira otimizada e simplificada. Uma GPU com características do estado da arte foi analisada em relação às suas precursoras e CPUs. Técnicas de programação para melhorar a performance também foram apresentadas. Em seguida, uma análise mostrou a superioridade das GPUs em programas altamente paralelos. Com isso, podemos afirmar que o futuro tende a explorar cada vez esse nicho, o que já vem criando supercomputadores baseados em GPUs, que possuem menor custo por poder de processamento e menor gasto energético.

6. REFERENCES

[1] CUDA Zone (http://www.nvidia.com/object/cuda_home_new.html), June 2010.

[2] OpenCL (www.khronos.org/opencl), June 2010.

[3] OpenGL (www.opengl.org), June 2010.

[4] T. Aamodt. Architecting Graphics Processors for Non-Graphics Compute Acceleration. *ece.ubc.ca*, pages 963–968, 2009.

[5] R. Amorim, G. Haase, M. Liebmann, and R. Weber dos Santos. Comparing CUDA and OpenGL implementations for a Jacobi iteration. *2009 International Conference on High Performance Computing & Simulation*, pages 22–32, June 2009.

[6] J. Cohen and M. Garland. Solving Computational Problems with GPU Computing. *Computing in Science & Engineering*, pages 58–63, 2009.

[7] J. F. Croix and S. P. Khatri. Introduction to GPU programming for EDA. *Proceedings of the 2009 International Conference on Computer-Aided Design - ICCAD '09*, page 276, 2009.

[8] A. Kerr, G. Diamos, and S. Yalamanchili. Modeling GPU-CPU Workloads and Systems. *Computer Engineering*, 2010.

[9] S. Liang, Y. Liu, C. Wang, and L. Jian. A CUDA-based Parallel Implementation of K-Nearest Neighbor Algorithm. *Architecture*, pages 291–296, 2009.

[10] J. Nickolls and W. Dally. The GPU Computing Era. *IEEE Micro*, 30(2):56–69, 2010.

[11] J. Owens. GPU architecture overview. *SIGGRAPH'07: ACM SIGGRAPH 2007 courses*, 2007.

[12] K. Rehman and S. Srinathan. A Performance Prediction Model for the CUDA GPGPU Platform. *research.iiit.net*, 2009.

[13] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science & Engineering*, pages 66–72, 2010.