# O Potencial das Graphics Processing Units

Juan Salamanca Guillén
ra134063
Instituto de Computação
Universidade Estadual de Campinas
Campinas, SP, Brasil
juj.sca@gmail.com

Rafael Auler
ra045840
Instituto de Computação
Universidade Estadual de Campinas
Campinas, SP, Brasil
rafael.auler@lsc.ic.unicamp.br

Junior Fabian Arteaga
ra123542
Instituto de Computação
Universidade Estadual de Campinas
Campinas, SP, Brasil
jfabianarteaga@gmail.com

## ABSTRACT

As tendências atuais na indústria de microprocessadores têm colocado em evidência os modelos de programação paralela para vários núcleos como o futuro da computação de alto desempenho, uma vez que não foi possível sustentar o expressivo aumento de desempenho de um único núcleo nos últimos anos. Contudo, ainda existem grandes desafios com relação à abordagem do modelo de programação e à escalabilidade limitada de processadores de múltiplos núcleos convencionais. Explorando um outro extremo do projeto de processadores, as GPUs enfatizam programação massivamente paralela altamente escalável que pode superar o desempenho de processadores comuns em uma ordem de magnitude para seu nicho de aplicações. Inicialmente projetadas para aplicações gráficas, as GPUs têm se mostrado cada vez mais aptas para a computação de propósito geral. Este artigo apresenta uma revisão sobre a evolução deste hardware especial, bem como uma discussão sobre como programar GPUs para problemas genéricos e como esta plataforma pode impulsionar o desenvolvimento de aplicações paralelas. A colaboração de CPUs e GPUs para resolver problemas em sistemas heterogêneos mostra-se como um caminho promissor para o futuro dos sistemas computacionais.

## Keywords

Arquitetura de Computadores, GPU, GPGPU, GPU Computing

## 1. INTRODUÇÃO

As GPUs (*Graphical Processing Units*) evoluíram substancialmente nesta última década. Já não são mais só poderosos dispositivos para gráficos, mas são também processadores paralelos programáveis de alto desempenho, contando com largura de banda de memória, capacidade de cálculo e eficiência no consumo de energia que superam um processador convencional em uma ordem de magnitude para o seu nicho de aplicações.

Esta capacidade de processar tarefas de propósito geral posicionou as GPUs como alternativas atrativas a processadores comuns, dando origem ao novo termo GPGPU (*General Purpose Graphics Processing Unit*). O ato de utilizar GPUs para processamento de propósito geral também é conhecido na literatura como *GPU Computing*.

Os desenvolvimentos de processadores nos últimos anos mostraram uma tendência em explorar paralelismo. A adição de mais núcleos tomou prioridade em detrimento da melhoria do desempenho de um único núcleo. Uma GPU leva esta tendência para um extremo da exploração do espaço de projeto, oferecendo muitas possibilidades para o futuro por apresentar uma arquitetura e modelo de programação diferentes a aqueles para um único núcleo de execução.

Entretanto, as aplicações que aproveitam o poder de processamento da GPU têm que contar com determinadas características. Owens et al.[10] lista as mais importantes:

- Expressivos requisitos computacionais. Aplicações gráficas em tempo real, como jogos eletrônicos, são um exemplo, pois precisam de uma elevada quantidade de operações aritméticas para renderizar milhões de pixels por segundo;
- Altamente paralelizáveis;
- Priorizar vazão à latência. Apesar da alta vazão, tipicamente o *pipeline* gráfico é bem profundo, com uma latência de centenas de milhares de ciclos.

Em novembro de 2006 foi apresentada a GeForce 8800 GTX, um cartão gráfico importante por demonstrar a primeira GPU que traz hardware explícito para tarefas de propósito geral. Este dispositivo traz como novidade a habilidade para fazer escritas em endereços de memória arbitrários e a presença de memória no chip compartilhada entre *threads* para assim reduzir a largura de banda de memória fora do chip [5].

Como vemos, a GPU evoluiu partindo de um processador com funções fixas orientadas ao pipeline de gráficos para um poderoso processador programável, com interfaces de hardware e software que se concentram nestes aspectos de programação. Esta arquitetura distinta será discutida na seção 2. Com este novo enfoque que a GPU tem, novos modelos e ferramentas de programação têm sido desenvolvidas.

O objetivo é balancear o alto desempenho oferecido pela programação de baixo nível com a flexibilidade da programação em alto nível. Na seção 3 discutiremos os modelos programação e as técnicas usadas para implementar aplicações para GPUs.

Na atualidade procura-se por aplicações reais que destaquem as vantagens das GPU. O campo ainda não está completamente amadurecido, e futuros desafios podem surgir. Na seção 4 apresentamos um caso de estudo: *(NVIDIA)*. Já na seção 5 fazemos uma comparação com as CPUs e suas diferentes aplicações em campos distintos. Finalmente, a seção 6 conclui este artigo.

# 2. ARQUITETURA DA GPU

A arquitetura da GPU foi gradualmente se transformando em um modelo mais flexível e programável. Para descrever a arquitetura, primeiro é importante começar com o *pipeline* de gráficos da GPU e depois discutir sua evolução para uma arquitetura para execução de programas de propósito geral. Esta seção encerra com uma revisão sobre a arquitetura de uma GPU moderna.

## 2.1 O pipeline de gráficos

A entrada do pipeline são vértices num sistema de coordenadas tridimensional. No *pipeline*, estes são montados para criar a imagem que será mostrada na tela. Os passos que o pipeline segue são:

- Operações de vértices: cada vértice deve ser transformado no espaço da tela e sombreado por sua interação com as luzes da cena. Este processamento tem as características para ser feito paralelamente por hardware porque cada vértice é processado independentemente dos outros;
- Montagem de primitivas: os vértices são montados em triângulos.
- Rasterização: neste processo determinam-se quais pixels do espaço da tela são cobertos por um determinado triângulo. Cada triângulo gera uma primitiva chamada *fragmento* em cada pixel que cobre. Assim, cada pixel pode ter vários fragmentos, dos quais se pode calcular o valor de cor desse pixel.
- Operações dos fragmentos: esta etapa é a mais exigente em recursos. Utiliza-se o valor de cor de cada vértice e cada fragmento é sombreado para determinar sua cor final. Assim como com vértices, o processamento poder ser feito paralelamente para cada fragmento.
- Composição: nesta etapa produz-se a imagem final com uma cor para cada pixel.

É importante mencionar que estas operações disponíveis para vértices e fragmentos eram *configuráveis*, mas não *programáveis*. Neste pipeline anterior, só era possível ao programador controlar as posições de vértices e suas respectivas cores, bem como as luzes na cena. Contudo, não era possível modificar o modelo que determina sua interação uma vez que este ficava fisicamente codificado no hardware.

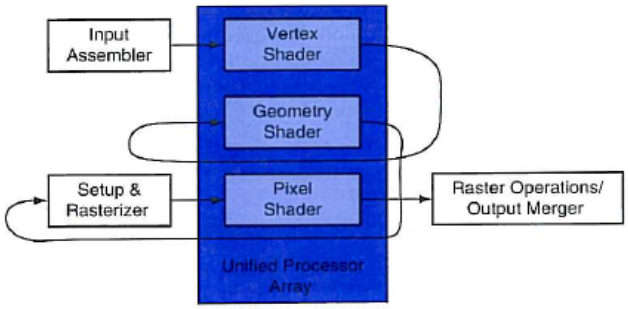


**Figure 1: O pipeline de gráficos com os estágios programáveis mapeados à matriz de processadores. Imagem retirada de Hennessy e Patterson [11].**

## 2.2 A evolução da arquitetura da GPU

Owens et al [10] mostra que a principal modificação da arquitetura nestes últimos anos foi substituir as funções fixas para os vértices e segmentos por programas que executam para cada vértice e fragmento. Estes programas, com o passar do tempo, têm aumentado sua capacidade, complexidade e, consequentemente, seu consumo de recursos. Além disso, o conjunto de instruções evoluiu ao ponto de já não ser mais necessário usar conjuntos de instruções separados para vértices e fragmentos, uma vez que as atuais GPUs suportam o modelo Shader 4.0 unificado para os *shaders* de vértices e fragmentos.

Atualmente, como o modelo dos *shaders* evoluiu e ficou mais poderoso, as aplicações para GPU aumentaram a complexidade de seus programas para vértices e fragmentos. Desta maneira, o esforço da arquitetura de GPUs foca nas partes programáveis do pipeline de gráficos. Anteriormente, a GPU contava com um pipeline de funções fixas de partes configuráveis. Agora, pode ser encarada como um dispositivo programável rodeado de unidades com funções fixas de suporte, tornando-se atrativa para executar programas altamente paralelos ou de propósito geral.

## 2.3 Arquitetura de uma GPU moderna

Conforme mencionado na introdução, uma aplicação precisa contar com determinadas características para poder usufruir do desempenho computacional de pico oferecido pela arquitetura da GPU, de forma que seu *speedup* real é baseado na capacidade do aplicativo de explorar estes aspectos.

No pipeline para gráficos construído na GPU, as saídas de cada tarefa alimentam as entradas da tarefa seguinte. Isto expõe um paralelismo no nível de tarefas em que os dados de múltiplas fases do pipeline podem ser computados ao mesmo tempo. Ainda, em cada fase é possível calcular múltiplos elementos, explorando paralelismo de dados. Para efetuar o mesmo cálculo massivamente paralelo da GPU, uma CPU precisaria executar este tipo de pipeline uma etapa por vez, divindo-o no domínio do tempo.

Entretanto, no caso da GPU, sua arquitetura divide o pipeline em espaço. Portanto, os recursos do processador são divididos entre as tarefas, e assim uma parte do processador que está trabalhando em uma tarefa pode alimentar diretamente a outra parte do mesmo que está executando uma tarefa seguinte.

Esta organização obteve sucesso para as GPUs com funções fixas porque o hardware de um determinado estágio do pipeline poderia processar múltiplos elementos ao mesmo tempo, e muitos estágios de tarefas paralelas podem estar executando em qualquer momento. Além disso, também porque o hardware de cada estágio pode ser customizado para cumprir um propósito específico. Depois, conforme os estágios com funções fixas foram sendo substituídos por estágios programáveis, também os componentes de hardware de propósito específico foram substituídos por componentes de hardware programáveis, mas sem alterar a organização que expõe paralelismo explícito. O resultado é um longo pipeline com muitos estágios, cada um acelerado por hardware de propósito específico.

A maior desvantagem desta abordagem é a necessidade de balancear o pipeline, pois a vazão depende do estágio mais lento. Por exemplo, se o programa de vértices é mais complexo que o de fragmentos, a vazão depende do programa de vértices. Entretanto, atualmente, o conjunto de instruções de fragmentos e vértices é o mesmo, de forma que todas as unidades programáveis dos estágios de pipeline compartilham uma unidade programável em hardware que pode explorar tanto paralelismo de dados como de tarefas. Por esse motivo, a arquitetura de GPU não é mais estritamente orientada ao paralelismo de tarefas no pipeline, mas agora é construída em torno do paradigma da unidade programável concentrada no paralelismo de dados. Assim é possível obter um melhor balanceamento às custas de um maior custo de hardware.

### 2.3.1 *Visão geral*

Como é mostrado por Hennessy e Patterson [11], a arquitetura das GPUs atuais está baseada numa matriz de muitos processadores programáveis. O processamento de vértices, *pixel shaders* e fragmentos são feitos na mesma unidade de processamento. Em contraste às CPUs de múltiplos núcleos, as GPUs estão focadas em executar muitas *threads* paralelamente mapeadas em múltiplas unidades de execução. A organização típica é uma matriz de *streaming processors cores* (SP) distribuídos em alguns *multithreaded streaming multiprocessors* (SM), como se pode observar na Figura 2. Cada SP tem unidades aritméticas de inteiros e pontos flutuantes que executa a maioria das instruções disponíveis em um processador convencional. Aamodt [5] explica que para obter eficiência no consumo de energia e de área, estes multiprocessadores compartilham a chamada *multithreaded instruction unit*, uma *shared memory RAM* e uma *instruction cache*.

Com relação à sua estrutura geral, as GPUs podem ser classificadas como SIMD (*Single Instruction Multiple Data*) [5, 11], ou seja, uma instrução para vários dados. Dessa forma permite-se que vários dados sejam processados paralelamente coordenados por uma única instrução. Outros trabalhos ainda categorizam GPUs mais modernas como SPMD (*Single Program Multiple Data*) [10], haja vista o fato de que a presença de execução condicional para algumas instruções pode fazer com que múltiplos processadores executem o mesmo programa, mas não necessariamente as mesmas instruções, já que algumas *threads* podem tomar outro caminho de um desvio e anular a execução do caminho não tomado. Note que, entretanto, o fluxo de instruções buscadas permanece o mesmo para todas as unidades de um SM. A diferença de comportamento se dá pela anulação da execução daquela instrução em algumas unidades de processamento.


**Figure 2: Arquitetura da GPU NVIDIA GetForce 8800 GTX. Extraído de Owens et al. [10]**

Fabricantes de GPUs podem usar também a terminologia SIMT (*Single Instruction Multiple Threads*) ao invés de SIMD para melhor esclarecer como funciona a arquitetura de uma GPU, em que uma única instrução é executada por múltiplas *threads*. Nesta visão, cada SP não trabalha somente em um dado, mas sim em uma *thread* separada. A diferença fica mais clara quando este tipo de paralelismo é comparado com aquele explorado por instruções vetoriais de extensões como MMX e SSE da Intel, exemplos clássicos de SIMD. Na prática, um conjunto de *threads* executadas em paralelo (frequentemente 32) são agrupadas para sua execução em uma unidade chamada *warp* (NVidia) ou *wavefront* (ATI). A Figura 3 mostra um *warp* de *threads* paralelas em que há uma unidade de despacho que seleciona um *warp* para que execute sua instrução seguinte. Esta unidade emite essa instrução para todas as *threads* ativas desse warp, de forma que os SPs executam instruções compartilhadas por um número de threads.

Em contraste ao SIMD, SIMT permite aos programadores escreverem código paralelo no nível de *threads* que divergem (possuem comportamento diferente) e também código paralelo ao nível de dados para *threads* coordenadas (de mesmo comportamento para vários núcleos).

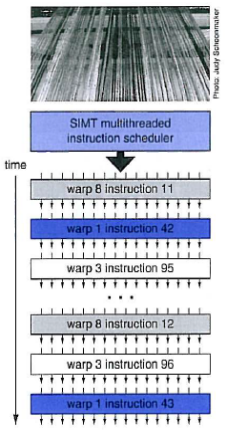


**Figure 3: Gerenciamento de SIMT multithreaded warp. Extraído de Hennessy e Patterson [11]**

É importante ressaltar que as *threads* discutidas no contexto de GPUs são diferentes daqueles das CPU comuns, já que são menos onerosas e de granularidade mais fina, ao passo que seu desempenho individualmente é mais fraco.

O conjunto de instruções presente em GPUs define tipos de instruções que dificilmente são encontradas em CPUs comuns, como, por exemplo, o cálculo de funções transcedentais (seno, cosseno e logatítmo) e operações de textura. Entretanto, também contam com diversas instruções em comum com outras CPUs: operações de ponto flutuante, inteiros, lógicas, controle de fluxo e acesso a memória [11]. Algumas GPUs atuais também tem implementado mecanismos de predição e especulação de desvios que permitem continuar a execução das *threads* mesmo quando houver um desvio condicional. Assim, os dados produzidos pela thread não são gravados na memória ou nos registradores até que o desvio seja resolvido [5].

### 2.3.2 Sistema de Memória da GPU

Com relação ao sistema de memoria que é usado em GPUs, geralmente é configurado no cartão gráfico uma hierarquia completa com vários níveis de memória: memória principal fora do *chip* (DRAM - *Dynamic Random Access Memory*), memória compartilhada, memória local e memória de constantes. O objetivo é alcançar as seguintes propriedades:

- Vários níveis de cache para reduzir o tráfico fora do *chip*;
- Grande largura de banda entre GPU e memoria;
- Compressão de dados;
- Elevada vazão.

Os subsistemas de memoria da GPU são organizados em *partições de memória*, e cada uma destas compreende um controlador de memoria independente e um ou dois dispositivos DRAM que são exclusivamente atribuídos a esta partição.

Para um funcionamento eficaz da GPU, esta precisa maximizar a disponibilidade de dados necessários para execução das *threads*. Para abordar este desafio, o uso de memórias cache (*streaming cache architecture*) nas GPUs atuais é utilizado para aumentar a eficiência do acesso a memória (por exemplo, para obter dados de textura) através de grandes blocos. Além disso, GPUs modernas são capazes de traduzir endereços virtuais em físicos, tornando necessária a presença de uma unidade MMU (*Memory Management Unit*).

A memoria global é do tipo DRAM externa que permite a comunicação entre os diferentes blocos de *threads*, de forma que o espaço de endereçamento é repartido entre todas as partições de memória. Na GPU, as *threads* compartilham memória que fica dentro dos multiprocessadores (que poderiam ser usadas como *buffer* na renderização). Portanto, cada memória compartilhada por um bloco de *thread* é visível só por esse bloco. Uma das vantagens deste tipo de memórias compartilhada é, por estar dentro do chip, a ausência de contenção elevada para seu acesso, além de permitir uma alta largura de banda para sustentar as demandas de cada SP.

As memorias locais são visíveis para uma *thread* individual. Para oferecer maior capacidade de memória, estas são implementadas em uma DRAM externa. Já o banco de registradores é dividido entre as *threads*, mas é importante observar que isto pode ser um problema caso o número de portas da memoria seja menor do que o número de threads [5].

Cada núcleo da GPU utiliza granularidade fina na execução para intercalar os *warps* em cada ciclo de clock. Com essa estratégia de multiplexação de *warps* no tempo, é possível esconder as latências de acessos a memória e de operações aritméticas complexas. Ainda, com isso é possível também sobrepor as petições de memoria em uma forma de paralelismo chamada paralelismo no nível de memoria [11].

## 3. MODELO DE PROGRAMAÇÃO DE GPU

Essencial para o sucesso de qualquer hardware que ambiciona tornar-se uma plataforma de computação de propósito geral é o apelo que oferece à comunidade de programadores nos quesitos de facilidade de uso e desempenho esperado. Fazer com que a computação paralela torne-se fácil é uma necessidade da qual depende o futuro dos microprocessadores atuais, tanto GPUs como CPUs. Para a subclasse de problemas de computação que contam com paralelismo evidente no processamento de múltiplos elementos independentes, a plataforma de processamento oferecida por GPUs é mais eficiente do que CPUs. Ainda, a GPU não é um hardware raro, uma vez que pode ser encontrada com facilidade em sistemas computacionais como estações de trabalho e servidores. Isto tornou-se mais evidente graças ao advento de processadores CMP (*Chip Multi Processors*) heterogêneos, em que um dos núcleos é uma GPU, como a família Intel i3, i5 e i7 [2].

Portanto, é importante disponibilizar linguagens que permitam o fácil acesso a este hardware para que todo o sistema computacional seja eficientemente utilizado e subsidie a disseminação da programação paralela com os característicos grandes benefícios de desempenho de GPUs.

## 3.1 Modelos Ultrapassados

Antes de discutir como o modelo de programação de GPUs se encontra atualmente, é necessário discutir como as GPUs eram programadas anteriormente para entender depois como a abstração deste hardware evoluiu.

A GPU surgiu como um mecanismo acelerador de funções gráficas para o emergente mercado de jogos eletrônicos, que acompanhou de perto a evolução da capacidade computacional de computadores de propósito geral e consoles especializados para reprodução de jogos. De maior atenção é o período em que os jogos eletrônicos e a indústria do cinema criaram uma demanda expressiva para o uso de algoritmos de computação gráfica, em que um ponto de vista de um ambiente tridimensional é manipulado e apresentado ao usuário em um vetor de pixels que forma a imagem bidimensional.

Esta categoria de algoritmos caracteriza-se por:

- Realização de cálculos aritméticos independentes para determinação da cor final de cada pixel. Note que não existe a necessidade de serialização: um pixel $x$ pode ser calculado antes, depois ou ao mesmo tempo que outro pixel $y$.
- Encadeamento de sucessivos passos de cálculos que atuam com os dados apresentados na saída do passo imediatamente anterior.

Essas duas características mais importantes criam um grande potencial para aumento do desempenho através da exploração de paralelismo de dados (item 1) e de tarefas (item 2). O primeiro é obtido aumentando o número de unidades aritméticas em relação a um processador tradicional, e o segundo é obtido através da criação de um *pipeline* de tarefas em hardware.

Entretanto, como não era objetivo da indústria oferecer um hardware para computação de propósito geral, as primeiras GPUs eram fortemente restritas ao domínio da computação gráfica.

A comunidade científica ocupou-se, nessa primeira fase de GPUs, a mostrar o potencial destes novos processadores, mapeando outros problemas de computação, como cálculo da mecânica de fluidos, para uma GPU. Neste modelo de programação primitivo, o resultado das tarefas de computação eram invariavelmente armazenados como pixels, ao passo que os cálculos eram construídos como operações do tipo *shader* para determinação da cor do pixel. O resultado final então, ao invés de ser interpretado como uma imagem, era recuperado pela CPU e aproveitado no modelo de simulação de mecânica de fluidos.

Tais primeiros usos de GPUs para problemas externos ao domínio da computação gráfica chamaram a atenção dos programadores e a demanda por um modelo de programação genérico, independente da terminologia de gráficos aumentou, mobilizando os fabricantes de GPUs a apresentar soluções cada vez mais flexíveis para programar este hardware.

## 3.2 OpenCL

OpenCL (*Open Computing Language*) é uma linguagem baseada em C99 (ISO ISO/IEC 9899:1999) que provê uma camada de abstração para que programadores possam facilmente usufruir de quaisquer composição de arquiteturas heterogêneas incluindo CPUs e GPUs em um sistema. OpenCL não é restrito a aplicações gráficas e foi um passo importante para formarlizar uma abordagem de programação para que aplicações com abundância de paralelismo possam operar e escalar em diversos dispositivos de CPUs de múltiplos núcles e GPUs. A linguagem foi concebida em um consórcio de empresas chamado Khronos Computing Working Group que inclui a Apple, AMD, Intel e Nvidia e sua versão 1.0 foi lançada em 18 de novembro de 2008. Atualmente está em sua versão 1.2 [3] anunciada em 15 de novembro de 2011.

A unidade básica de computação em OpenCL é chamada de `work-item` [7]. De uma maneira geral, um sistema computacional compatível com OpenCL irá tentar processar o maior número de `work-item`s em paralelo. Os `work-item`s possuem uma pequena e rápida memória privada interna e são organizados em `work-groups`. Todos os `work-item`s (ou *threads*) de um `work-group` podem compartilhar dados de maneira rápida e implementar sincronização de baixo custo. Ainda, as computações de diferentes `work-groups` não podem ser sincronizadas com respeito às de outro grupo. Finalmente, o conjunto de todos `work-group`s formam o `ND-range`, que compartilham um espaço de memória global.

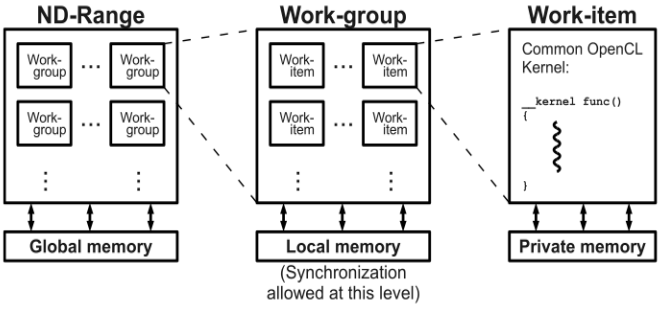


**Figure 4: Conceitos chave da linguagem OpenCL. Figura extraída de Zhongliang et al. [7]**

Para melhor entender as decisões de organização de OpenCL, convém apresentar como um código OpenCL é mapeado em uma GPU real. Para este exemplo, iremos considerar a família Evergreen da AMD (Radeon série HD5000). Esta GPU possui 20 unidades de processamento com caches L1 privadas para cada uma. Todas juntas compartilham caches L2 e o acesso ao barramento da memória global de sistema.

Cada unidade de processamento possui 16 *stream cores*, que imitam um processador de propósito geral bastante simplificado com capacidade de execução VLIW de 5 *slots*. Portanto, este sistema computacional possui desempenho de pico estrutural de 1600 operações por ciclo. A simplificação em relação a uma CPU está no limitado controle de fluxo: todos os 16 *stream cores* compartilham a mesma unidade de busca de instruções, o que significa que todas estão fadadas a executar a mesma instrução.

Neste cenário, OpenCL irá mapear cada `work-item` em um *stream core* distinto, ao passo que quando há `work-group`s distintos, cada grupo é mapeado em unidades de processamento diferentes. Um código de exemplo para soma de dois vetores é apresentado a seguir:

```
__kernel void
  vectorAdd(__global const float * a,
            __global const float * b,
            __global       float * c)
  {
      // Indice do vetor
      int nIndex = get_global_id(0);
      c[nIndex] = a[nIndex] + b[nIndex];
  }
```

Cada `work-item`, `work-group` e `ND-range` possui seu próprio identificador, e o *runtime* OpenCL oferece meios de se obter estes valores. No código acima, o identificador do `work-item` é usado para indexar o elemento do vetor a ser somado. Desta maneira, este *kernel* (código comum de um `work-group`) irá somar um vetor cujo tamanho é igual ao número de `work-item`s lançados. O código hospedeiro, que lança as tarefas para execução, deve ser codificado em uma função separada e organiza o resultado obtido com a computação realizada pelos *kernels*. Um esboço do código hospedeiro é apresentado a seguir:

```
int main(void){
 // Informa ao runtime que queremos despachar
 // a tarefa para uma GPU
 cl_context context = clCreateContextFromType(0,
                      CL_DEVICE_TYPE_ GPU, ...
 // get the list of GPU devices associated
 // with context
 clGetContextInfo(...
 // create a command-queue
 cl_cmd_queue cmd_queue = clCreateCommandQueue(...
 cl_mem memobjs[1];
 // Aloca buffer de entrada
 memobjs[0] = clCreateBuffer(...
 // Cria o programa e o kernel
 cl_program program = clCreateProgramWithSource(...
 cl_int err = clBuildProgram(...
 cl_kernel kernel = clCreateKernel(...
 // Configura argumentos do kernel e o executa
 err = clSetKernelArg(...
 err = clEnqueueNDRangeKernel(....
 // Lê resultado
 err = clEnqueueReadBuffer(
 ...
}
```

## 4. CASO DE ESTUDO: NVIDIA

NVIDIA é uma empresa multinacional que fabrica peças de computador, e é mais popularmente conhecida por sua série de placas de vídeo *GeForce*. Neste segmento, concorre diretamente com a empresa AMD, que produz a série de placas de vídeo *Radeon*.

As primeiras GPUs foram projetadas como aceleradores gráficos, suportando somente pipelines de função fixa específicos. A partir do final dos anos 90, o hardware tornou-se cada vez mais programável, culminando na primeira GPU NVIDIA em 1999. Menos de um ano após a NVIDIA ter cunhado o termo GPU, artistas e desenvolvedores de jogos não eram os únicos a realizar trabalhos inovadores com a tecnologia: pesquisadores estavam tirando proveito de seu excelente desempenho para ponto flutuante.

Como citado na seção anterior, a programação para as GPUs da primeira geração não era nada fácil, nem mesmo para aqueles que conheciam linguagens de programação gráficas como a OpenGL. Os desenvolvedores eram obrigados a mapear cálculos científicos para problemas que poderiam ser representados por triângulos e polígonos.

Em 2003, uma equipe de pesquisadores liderada por Ian Buck anunciou o Brook, o primeiro modelo de programação de ampla adoção a ampliar a linguagem C com construções de paralelismo de dados. Usando conceitos como fluxos, kernels e operadores de redução, o compilador Brook e um sistema de tempo de execução revelaram a GPU como um processador de propósito geral em uma linguagem de alto nível. E, ainda mais importante, os programas em Brook não apenas eram mais fáceis de serem codificados que os códigos de GPU ajustados manualmente, mas também eram sete vezes mais rápidos que os códigos similares existentes.

Logo depois, a NVIDIA sabia que um hardware extremamente rápido tinha que ser combinado a ferramentas intuitivas de software e hardware, e por isso convidou Ian Buck para juntar-se à empresa e começar a desenvolver uma solução para executar o C na GPU de forma fluida. Juntando o software e o hardware, a NVIDIA apresentou a CUDA *(Compute Unified Device Architecture)* em 2006, a primeira solução do mundo para computação de propósito geral em GPUs [4]. O CUDA não é uma nova linguagem de programação, mas uma biblioteca C. Com algumas restrições, qualquer código C pode rodar na GPU. No CUDA é vista como um co-processador genérico, e não apenas gráfico, não sendo mais necessário criar um contexto OpenGL para acessá-la.

A grande novidade é a maneira eficiente que o CUDA juntamente com a arquitetura da GPU, possibilita o desenvolvimento de aplicações que podem explorar ao máximo o paralelismo de dados [1]. Em uma dada aplicação, um mesmo trecho de código é executado em paralelo para pequenos blocos de dados, com a existência de várias pequenas caches e níveis de hierarquia de memoria que escondem a latência de acesso a estes blocos como é mostrado na Figura 5.


**Figure 5: Diferença arquiteturais entre CPU e GPU [1].**

Desde a sua introdução em 2006, CUDA têm sido amplamente utilizada através de milhares de aplicações e trabalhos de pesquisa. Atualmente, mais de 300 milhões de GPUs instaladas em notebooks, workstations, clusters e supercomputadores utilizam CUDA. Na página oficial (CUDA Zone) estão disponíveis milhares de aplicações acelerados por GPU aplicadas em diferentes áreas, tais como: astronomia, biologia, química, física, manufatura, finanças etc. CUDA é uma plataforma de computação em paralelo e um modelo de programação que permite um aumento no desempenho computacional, aproveitando a energia da unidade de processamento gráfico (GPU).

Os desenvolvedores de software, cientistas e pesquisadores estão descobrindo usos amplamente variados para a computação com GPU CUDA. Aqui estão alguns exemplos [4]:

- **Identificação de placas ocultas em artérias:** Ataques cardíacos são a maior causa de mortes no mundo todo. A Harvard Engineering, a Harvard Medical School e o Brigham & Women's Hospital se reuniram para usar GPUs com o objetivo de simular o fluxo sanguíneo e identificar placas arteriais ocultas sem fazer uso de técnicas de imageamento invasivas ou cirurgias exploratórias.

- **Análise do fluxo de tráfego aéreo:** O National Airspace System (Sistema de Espaço Aéreo Nacional) gerencia a coordenação do fluxo de tráfego aéreo em âmbito nacional. Modelos computacionais ajudam a identificar novas maneiras de aliviar congestionamentos e manter o tráfego de aeronaves fluindo de forma eficiente. Utilizando o poder computacional das GPUs, uma equipe da NASA obteve grande ganho de performance, reduzindo o tempo de análise de dez minutos para três segundos.

- **Visualização de moléculas:** Uma simulação molecular denominada NAMD (dinâmica molecular em nanoescala) alcança um grande aumento de performance com o uso de GPUs. Essa aceleração é resultado da arquitetura paralela das GPUs, que permite que desenvolvedores NAMD migrem partes de aplicativos com alta demanda computacional para a GPU utilizando o CUDA Toolkit.

Na Tabela 1 apresentamos uma lista comparativa das 5 últimas chips lançadas pela NVIDIA

**Table 1: Características das cinco últimas chips da NVIDIA.**

| Chip Gráfico | Clock | Memória | Taxa de Transf. Memória | Pixels por Clock |
|---|---|---|---|---|
| GeForce GTX 690 | 915 MHz ×2 | 256 bits ×2 | 192.2 GB/s ×2 | 1536 ×2 |
| GeForce GTX 680 | 1.006 MHz | 256 bits | 192.2 GB/s | 1536 |
| GeForce GTX 645 | 776 / 1.552 MHz | 192 bits | 91.9 GB/s | 288 |
| GeForce GTX 640 | 950 / 950 MHz | 128 bits (GDDR5) | 80 GB/s | 384 |
| GeForce GTX 640 | 720 / 1.440 MHz | 192 bits (DDR3) | 43 GB/s | 144 |

# 5. COMPARAÇÃO ENTRE CPUS E GPUS

Uma técnica de modelagem da performance entre uma CPU e uma GPU é apresentada em [8], onde são atribuídos valores a determinadas características consideradas principais na descrição de um processador. A Figura 6 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 ideais para computação paralela.

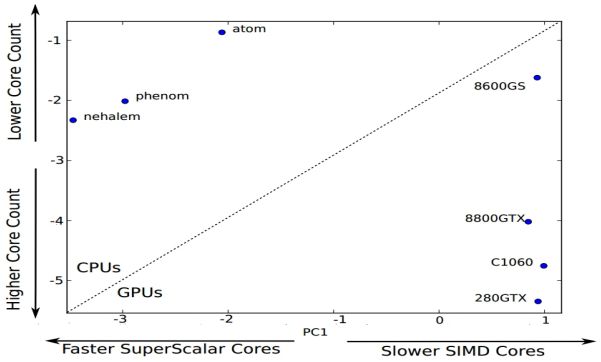


**Figure 6: Comparação dos modelos entre GPU e CPU.**

Uma outra aplicação para medir as diferenças entre CPUs e GPU é a implementação do algoritmo que calcula os K vizinhos mais próimos (KNN), muito utilizado atualmente em servidores para classificação de dados em categorias. Esse algoritmo foi desenvolvido em CUDA em [9] e é conhecido como CUKNN. A Figura 7 foi feita com 32768 objetos de referência de oito dimensões e escolhendo os sete vizinhos mais próximos. O algoritmo KNN tem uma etapa de ordenação que em [9] foi implementada serialmente, usando Insertion Sort e Quick Sort, e em paralelo usando CUKNN. 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.

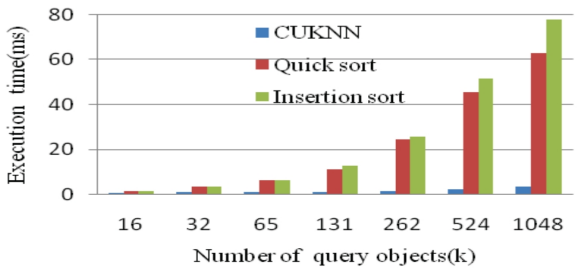


**Figure 7: Comparação entre GPU e CPU no algoritmo KNN.**

A Tabela 2 apresenta outra comparação entre CPUs e GPUs [12]. Nessa tabela temos o problema da ordenação de números implementada em um CPU utilizando a função *std:sort* e implementada em um GPU usando a função *Bitonic Merge Sort* (uma versão do merge sort implementada em paralelo).

| *std:sort* *Pentium 4 3.0 GHz* | | | *Bitonic Merge Sort:* Float Data *NVIDIA GeForce 6800 Ultra* | | |
|---|---|---|---|---|---|
| N | Sorts/Sec | Keys/Sec | N | Sorts/Sec | Keys/Sec |
| $256^2$ | 82.5 | 5.4 M | $256^2$ | 90.07 | 6.1 M |
| $512^2$ | 20.6 | 5.4 M | $512^2$ | 18.3 | 4.8 M |
| $1024^2$ | 4.7 | 5.0 M | $1024^2$ | 3.6 | 3.8 M |

**Table 2: Comparação do problema da ordenação de dados utilizando a função *std:sort* numa Pentium 4 e utilizando a função *Bitonic Merge Sort* numa NVIDIA GeForce.**

Um outro exemplo do uso da GPU é encontrado em [6], onde os autores utilizaram GPU para a execução de simulações de Diferenças finitas no domínio do tempo (Finite Difference Time Domain – FDTD), para resolver as equações de Maxwell em campos eletromagnéticos. Dentre os experimentos realizados, os autores verificaram que uma GPU GeForce 8800 GTX era 429 vezes mais rápida que uma CPU Opteron 270. Enquanto que, uma Core 2 Duo T7600 (2.33 Ghz.) foi unicamente 2 vezes mais rápida que a CPU.

# 6. CONCLUSÕES

Esse artigo apresentou uma visão geral do potencial das GPUs. Na primeira parte temos descrito a arquitetura de uma GPU, indicando o pipeline dos gráficos, a evolução da arquitetura e mostramos em detalhe a arquitetura de uma GPU moderna. Logo depois, descrevemos o modelo de programação de GPU onde detalhamos como eram programadas anteriormente e como são atualmente programadas usando a linguagem OpenCL, que foi um passo importante para formalizar uma abordagem de programação em paralelo.

A programação em GPU não é realizada seguindo o modelo tradicional de programação de CPU, mas para obter maior eficiência devemos trabalhar com o modelo conhecido como SIMD ou SIMT. Para conseguir altas velocidades com a GPU é necessário: Formatar os vetores em arranjos bidimensionais, Executar processamento de grandes quantidades de dados e Executar grande número de operações simples por itens de dados. Finalmente, mostramos um caso de estudo do uso da GPU pela empresa NVIDIA em diferentes aplicações, e apresentamos comparações entre CPUs e GPUs em distintas aplicações.

A abordagem da NVIDIA com CUDA é de uma linguagem de alto nível e fácil de utilizar para desenvolver aplicações reais. Uma das maiores vantagens de utilizar GPU é que o custo é muito menor em comparação com um computador equivalente para realizar a mesmas tarefas. Além disso, as GPUs tem um poder de processamento maior e um gasto energético menor.

# 7. REFERENCES


[1] NVIDIA: CUDA. *Programming Guide 2.0*. NVIDIA, 2008.

[2] Core i5 and i3 with on-chip GPUs launched, May 2012.

[3] OpenCL 1.2 specification, May 2012.

[4] NVIDIA: CUDA zone. *History of GPU Computing*. http://www.nvidia.com/, Jan. 2012.

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

[6] S. Adams, J. Payne, and R. Boppana. Finite Difference Time Domain (FDTD) Simulations using Graphics Processors. In *Proceedings of the 2007 DoD High Performance Computing Modernization Program Users Group Conference*, pages 334–338, Washington, DC, USA, 2007. IEEE Computer Society.

[7] Z. Chen, D. Kaeli, N. Materise, P. Mistry, D. Schaa, R. Ubal, K. Ziabari, B. Jang, N. Rubin, P. Lopez, S. Petit, and J. Sahuquillo. *The Multi2Sim Simulation Framework*, version 3.3 edition.

[8] A. Kerr, G. Diamos, and S. Yalamanchili. Modeling GPU-CPU workloads and systems. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 31–42, New York, NY, USA, 2010. ACM.

[9] S. Liang, Y. Liu, C. Wang, and L. Jian. A CUDA-based parallel implementation of K-nearest neighbor algorithm. In *Cyber-Enabled Distributed Computing and Knowledge Discovery, 2009. CyberC '09. International Conference on*, pages 291 –296, oct. 2009.

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

[11] D. Patterson and J. Hennessy. *Computer organization and design: the hardware/software interface*. The Morgan Kaufmann Series in Computer Architecture and Design, 2008.

[12] M. Tri. GPUs - Graphics Processing Units. Institute of Computer Science - University of Innsbruk, 2008.