

# Comparação de Eficiência entre as plataformas OpenCL e CUDA em GPUs NVIDIA

Thiago de Gouveia Nunes

Supervisor: Prof. Doutor Marcel P. Jackowski

14 de outubro de 2012

# Sumário

<b>1</b>	<b>Introdução</b>	<b>3</b>
1.1	Motivação . . . . .	3
1.2	Objetivos . . . . .	3
1.3	Problemas a serem resolvidos . . . . .	3
<b>2</b>	<b>Conceitos e Tecnologias</b>	<b>4</b>
2.1	High-Performance Computability . . . . .	4
2.2	GPU . . . . .	4
2.3	CUDA . . . . .	5
2.3.1	Modelo de Plataforma . . . . .	5
2.3.2	Modelo de Programação . . . . .	6
2.3.3	Hierarquia de Memória . . . . .	7
2.4	OpenCL . . . . .	8
2.4.1	Modelo de Plataforma . . . . .	8
2.4.2	Modelo de Execução . . . . .	9
2.4.3	Modelo de Memória . . . . .	10
2.4.4	Modelo de Programação . . . . .	11
<b>3</b>	<b>Atividades Realizadas</b>	<b>12</b>
3.1	Comparação de eficiencia . . . . .	12
3.1.1	Como fazer a comparação? . . . . .	12
3.1.2	Montagem dos kernels . . . . .	12
3.2	Comparação das abstrações . . . . .	12
3.3	Comparação entre os arquivos .ptx . . . . .	12
<b>4</b>	<b>Resultados</b>	<b>13</b>
4.1	Comparação de eficiencia . . . . .	13
4.1.1	Kernel memory-bound . . . . .	13
4.1.2	Kernel processing-bound . . . . .	13
4.2	Comparação das abstrações . . . . .	13
4.2.1	Semelhanças . . . . .	13
4.2.2	Diferenças . . . . .	13
4.3	Comparação dos .ptx . . . . .	13
<b>5</b>	<b>Conclusões</b>	<b>14</b>
<b>6</b>	<b>Bibliografia</b>	<b>15</b>

# **1 Introdução**

## **1.1 Motivação**

Em Computação de Alto Desempenho (HPC) existe uma parcela de supercomputadores montados com base em placas de processamento gráfico (GPU). O termo GPGPU (General-purpose computing on graphics processing units) é usado para denotar o uso de GPUs para executar programas de mais amplo espectro.

Duas linguagens são muito utilizadas atualmente para programação em ambientes GPGPU, OpenCL (Open Computing Language) e CUDA (Compute Unified Device Architecture). OpenCL é reconhecido como a melhor linguagem para ambientes GPGPU genéricos, enquanto o CUDA é a melhor para ambientes NVIDIA.

## **1.2 Objetivos**

O objetivo do estudo é comparar a eficiência de programas escritos nessas duas linguagens rodando em uma placa NVidia GeForce GTX 460 e comparar o modo com que eles abstraem uma GPU, tornando possível rodar programas genéricos na mesma.

## **1.3 Problemas a serem resolvidos**

Para realizar a comparação de eficiência entre as linguagens é necessário desenvolver dois algoritmos para testes, um que verifique a capacidade de processamento da linguagem e outro a capacidade de manipular memória. Além disso, é necessário comparar as abstrações para entender de onde vem a diferença de desempenho entre as linguagens.

## 2 Conceitos e Tecnologias

### 2.1 High-Performance Computability

HPC nasceu da necessidade de poder computacional para resolver uma série de problemas, entre eles:

- Previsão climática
- Modelação molecular
- Simulações físicas
- Física quântica

Até o final dos anos 90 todos os supercomputadores tinham como base processadores vetoriais. Só no final da década seguinte, com o aumento do desempenho das GPUs, que alguns supercomputadores começaram a usar GPUs como suas fontes e processamento.

### 2.2 GPU

A primeira GPU foi a GeForce 256, lançada em 1999. O hardware seguia um pipeline com 2 processos, um que aplicava transformações em vértices e outro em pixels. Em 2001, a GeForce 3 trouxe o primeiro processador de vértices programável. Em 2005 a primeira GPU com um processador unificado, usado tanto para operações em pixels como em vértices, foi lançada para o console Xbox 360. Para unificar os 2 processos do pipeline num único processador foi necessário generalizar esse processador, e essa generalização abriu as portas para programas genericos executarem na GPU.

A placa usada para os testes desse trabalho, a GeForce GTX460, usa a arquitetura Fermi, a segunda mais nova arquitetura da NVIDIA para GPUs. Essa arquitetura separa o fluxo de execução baseando-se no tipo de aplicação que será executada nela. Existe um fluxo para aplicações gráficas e outro para aplicações genéricas, que é o foco desse trabalho. Vamos estudar o fluxo generico abaixo.

A placa contém um escalonador implementado em hardware para threads. Ele é responsável por escalonar as threads que serão executadas nos streaming multiprocessors (SM). Os SM são conjuntos de 48 processadores, um pequeno bloco de memória própria, um cache de instruções e 8 unidades de funções especiais. A Geforce GTX 460 tem 7 SMs, totalizando 336 processadores.

O código que será executado em cada processador é chamado de **kernel**. Então, ao executar um kernel na GPU, o hardware criará threads, cada uma executando esse mesmo kernel mas com dados diferentes, e cada thread será escalonada para um processador diferente. Como as threads são distribuidas pelos SMs varia com a linguagem usada para se comunicar com a GPU.

Outra parte importante do hardware é a memória. A memória na GPU é limitada em relação à da CPU. GPUs tem, em média, 1GB de memória, enquanto CPUs tem 4GB. Outro fator é a transferencia de dados da memória principal do computador para a memória principal da GPU. A transmissão é feita por um barramento PCI Express, com velocidades de até 16GB/s, dado que o barramento esteja sendo usado somente para isso. Essa transmissão é

a parte mais lenta de todo o processo de execução na GPU e dado isso, em alguns casos é mais viável executar na GPU um pedaço do seu programa que seria executado na CPU do que retornar os dados computados na GPU para a CPU, executar esse pedaço específico, e passá-los de volta para a GPU para mais operações e novamente retornar esses dados para a CPU no final, passando duas vezes a mais pelo PCI Express.

Cada SM tem um bloco de memória de 64KB. Esse bloco pode ser configurado para 16KB de memória compartilhada e 48KB de cache L1 ou vice versa. A memória principal da placa é de 1024MB com conexões de 256 bits. A placa também tem um cache L2 de 512KB.

É importante conhecer a memória física da placa para qual se está programando por que acesso a memória é um dos maiores modificadores no desempenho de um programa paralelo. O acesso a memória é concorrente, mas ao utilizar caches e leitura/escritas em blocos podemos minimizar a taxa com que leituras/escritas conflitantes são feitas. Mas ainda sim é necessário atenção ao escrever um kernel. Dada a estrutura do hardware da GPU, é melhor deixar threads que façam operações sobre posições de memória próximas no mesmo SM, assim elas podem utilizar a memória compartilhada do mesmo, que além de ser mais rápido do que buscar os dados na memória principal, não cria um padrão de buscas frequentes na memória principal, que acabaria criando uma fila de acesso das threads e diminuiria o desempenho do programa.

## 2.3 CUDA

*Compute Unified Device Architecture* (CUDA) é uma arquitetura de programação para GPUs criada pela NVIDIA. Ele adiciona suas diretrizes para as linguagens C, C++, FORTRAN e Java, permitindo que elas usem a GPU. Esse trabalho usa o CUDA junto com a linguagem C. A versão 1.0 do CUDA foi disponibilizada no início de 2007. Atualmente só existe um compilador para CUDA, o `nvcc`, e ele só dá suporte para GPUs NVIDIA.

Para uma função executar na GPU ela precisa ser invocada de um programa da CPU. Chamamos esse programa de *Host* e a GPU onde o kernel irá executar de *Device*.

O CUDA implementa um conjunto virtual de instruções e memória, tornando os programas retroativos. O compilador primeiro compila o código em C para um intermediário, chamado de PTX, que depois será convertido em linguagem de máquina. Na conversão do PTX para linguagem de máquina o compilador verifica quais instruções o *device* suporta e converte o código para usar as instruções corretas. Para obter o maior desempenho possível, é importante saber para qual versão o código final será compilado, pois na passagem do código de uma versão maior para uma menor não existe a garantia que o algoritmo seguirá as mesmas instruções, o compilador pode mudar um conjunto de instruções para outro menos eficiente, ou em alguns casos, algumas instruções não existem em versões mais antigas do hardware.

### 2.3.1 Modelo de Plataforma

A inicialização dos recursos que o CUDA necessita para a comunicação com a GPU é feita no background da aplicação no momento da primeira chamada de alguma das diretivas do CUDA. Essa primeira diretiva terá um tempo maior

de execução que chamadas subsequentes a mesma diretiva. Na inicialização o CUDA identifica os *devices* existentes e escolhe um deles para ser o responsável pelas execuções posteriores.

O próximo passo é a alocação de memória no *device*. As operações de leitura de memória de um kernel são feitas somente na memória de um *device*. A alocação dessa memória é feita pelo *host*, usando `cudaMalloc()`. Para copiar a memória do *host* para o *device* ou vice-versa, `cudaMemcpy()` é usada. Para liberar o espaço alocado após a execução basta usar o `cudaFree()`. Todas essas diretivas recebem um ponteiro do *host*, usado para o controle sobre qual posição da memória está sendo operado em cada operação.

O CUDA dá suporte a alocação de vetores em duas ou três dimensões através de: `cudaMallocPitch()` e `cudaMalloc3D()`, respectivamente. É necessário usar as modificações dos comandos `Memcpy` para duas ou três dimensões também, que são: `cudaMemcpy2D()`, `cudaMemcpy3D()`.

### 2.3.2 Modelo de Programação

Um kernel no CUDA é uma função C que será executada paralelamente  $n$  vezes em  $n$  threads diferentes na GPU. Um kernel pode ser definido em qualquer lugar do seu código, usando a declaração `__global__` do lado esquerdo do tipo de retorno do kernel. Para invocar um kernel, o *host* faz a chamada de uma função com a sintaxe parecida com o C, mas usa uma configuração de execução definida pelo CUDA, que usa a sintaxe `<<<...>>>` junto da chamada da função. Os parâmetros da configuração são o número de blocos de threads e o número de threads por blocos. Para somar dois vetores de tamanho  $M$  e guardar o resultado num outro vetor, o código é o seguinte:

```
__global__ void MatrixMulti ( float* a, float* b, float* c) {
    int i = threadIdx.x;
    a[i] = b[i] + c[i];
}

int main () {
    ...
    VecAdd<<<1,M>>>(a, b, c)
    ...
}
```

No kernel acima, a linha `int i = threadIdx.x` atribui a variável `i` o valor do índice da thread atual na primeira dimensão. A estrutura `threadIdx` é um vetor de 3 dimensões, logo as threads podem ser organizadas em 1, 2 ou 3 dimensões dentro de um *device*. As threads são organizadas por blocos. Cada bloco tem dimensões maleáveis, mas as GPUs atuais limitam para 1024 o número máximo de threads por blocos. Cada bloco é lançado para execução em um processador diferente. Blocos são organizados em grids, que tem seu tamanho configurado na chamada o kernel, bem como o tamanho de cada bloco. No nosso exemplo acima, na linha `VecAdd<<<1,M>>>(a,b,c)`, o 1 determina o número de blocos e o  $M$  o número de threads por bloco.

O CUDA supõem que todos os blocos podem ser executados de maneira independente, ou seja, eles podem executar tanto paralelamente quanto sequencialmente. Com isso, é possível que o desempenho do código aumente em

GPUs com mais processadores, sem que o programador tenha que modificar o código.

O CUDA sabe qual instruções ele pode executar dentro de um *device* baseando-se no seu Compute Capability (Capacidade Computacional). A Compute Capability de um *device* são dois números, um que representa a arquitetura do *device*, e outro que representa melhorias numa arquitetura. A arquitetura *Tesla*, a primeira da NVIDIA a dar suporte a GPGPU, tem Compute Capability 1.x, a seguinte, a *Tesla*, tem 2.x e a atual, a *Kepler*, tem 3.x. Dentro de cada arquitetura, podem existir melhorias nas instruções, que são refletidas no número após o ponto, ou seja, uma placa com Compute Capability 2.1 tem instruções que uma 2.0 não tem.

### 2.3.3 Hierarquia de Memória

No CUDA, a memória é separada logicamente em 4 locais:

- Registradores - Toda variável de uma thread fica em registradores.
- Memória Local - Memória acessível por cada thread separadamente, mas de uso pouco provável. Ela só é usada se não existe mais espaço nos registradores ou se o compilador não ter certeza sobre o tamanho de um vetor.
- Memória Compartilhada - Cada bloco de threads tem uma memória compartilhada. A memória compartilhada é separada em pequenos blocos independentes. Se uma requisição de leitura tem  $n$  endereços em  $n$  blocos diferentes, o tempo de leitura desses  $n$  endereços é igual ao tempo de leitura de 1 endereço. Caso duas leituras caiam no mesmo bloco, elas serão serializadas. A memória compartilhada fica em chips dentro dos SMs, logo seu acesso é mais rápido do que o acesso a memória global.
- Memória Global - A memória global é acessível por qualquer bloco em execução em um *device*. A memória global não é resetada após a execução de um kernel, então chamadas subsequentes de um mesmo kernel simplesmente leem os resultados da memória global. Existe um pedaço da memória global reservada para valores constantes do programa.

Por padrão, o compilador do CUDA cuida do gerenciamento da memória, ou seja, ele é o responsável por distribuir os dados entre os locais diferentes de memória. O programador pode dar dicas para o compilador usando qualificadores indicando o local que ele quer que aquele elemento fique na memória. Os possíveis qualificadores são:

- `__device__` Fica na memória global.
- `__constant__` Fica na área constante da memória global.
- `__shared__` Fica na memória compartilhada das threads.
- `__restrict__` Indica para o compilador que todos os ponteiros com esse qualificador apontam para locais diferentes da memória. Isso é importante pois o compilador pode fazer otimizações com o código sabendo dessa informação.

GPUs com Compute Capability maior ou igual a 2.0 podem alocar memória dentro do *device* em tempo de execução.

## 2.4 OpenCL

Open Computing Language (OpenCL) é uma framework aberta para programação genérica para vários processadores, dentre eles GPUs e CPUs. OpenCL dá suporte para sistemas embarcados, sistemas pessoais, corporativos e até HPC. Ele consegue isso criando uma interface de baixo nível, ou seja, o mais próximo do hardware possível, e mantendo alto desempenho, com uma abstração portátil. O OpenCL também é uma API para controle de aplicações paralelas em sistemas com processadores heterogêneos. O OpenCL consegue, numa mesma aplicação, reconhecer vários processadores diferentes dentro de um mesmo computador, e executar códigos distintos entre eles, coordenando os hardwares. Aqui, como no CUDA, a parte do código executado na CPU é chamada de *Host* e o hardware que executa os kernels de *Devices*. É importante lembrar que dado essa generalização do OpenCL, é possível que a CPU onde o código do *host* esteja executando seja usada para rodar um kernel, e essa CPU passa a ser um *device* ao mesmo tempo em que roda o *host*. Tanto o fato do OpenCL ser aberto quanto o fato dele não se restringir a um hardware específico fazem dele a linguagem mais usada para GPGPU fora de GPUs NVIDIA.

O framework do OpenCL pode ser explicado usando 4 modelos hierárquicos, que são:

- Plataforma
- Memória
- Execução
- Programação

### 2.4.1 Modelo de Plataforma

No OpenCL existe um *host* conectado a um ou mais *devices*. Os *devices* são abstrações de uma GPU ou de uma CPU. Cada *device* é composto de uma ou mais Compute Unit (CU), e cada CU é composto de um ou mais Processing Element (PE). Por exemplo, uma CPU com 2 cores seria vista pelo OpenCL como um *device* com uma Compute Unit e 2 Processing Elements. O processamento dentro de um *device* ocorre num PE. O processamento é iniciado através de *comandos* que o *host* manda para o *device*. Os PEs podem executar tanto no modelo de SIMD (Instrução Única, Múltiplos Dados) ou SPMD (Processo Único, Múltiplos Dados). No SIMD, todas as threads executam a mesma operação ao mesmo tempo em dados diferentes e no SPMD cada thread tem um ponteiro de instrução próprio. O responsável por iniciar a execução dos kernels nos PE é o *host*.

O OpenCL tem suporte para vários tipos de *devices* diferentes: GPUs, CPUs, DSP ou Cell/B.E. . Para manter a retrocompatibilidade do código, cada *device* guarda 3 números importantes para o OpenCL:

- A versão da plataforma - Indica qual a versão da API que o *host* pode usar para se comunicar com o OpenCL. Diz respeito ao contexto, objetos de memória, filas de comando e *devices*.



- A versão do *device* - Indica qual a capacidade de um *device*, como possíveis funções implementadas em hardware ou limites de memória.
- A versão da linguagem - Indica o número de features do OpenCL implementadas no *device*.

O *host* usa a versão da linguagem para determinar o que pode ou não ser feito no *device* em momento de compilação. A versão da linguagem nunca é maior que a versão da plataforma, mas pode ser maior que a versão do *device*.

### 2.4.2 Modelo de Execução

Com as plataformas definidas, vamos entender como o OpenCL cuida da execução dos kernels dentro de uma plataforma. Cada instância de um kernel rodando dentro de um Processing Element é chamada de Work-Item. Dentro de um *device* é criado um conjunto de índices de até 3 dimensões, onde cada ponto dentro desse conjunto de índices é um work-item. Como visto acima, cada work-item executa o mesmo código, mas com dados diferentes e, existindo pulsos condicionais no código, o caminho de execução pode variar.

Esse conjunto de índices é chamado de NDRange. Ele é definido por um vetor de tamanho  $N$ ,  $N$  sendo o número de dimensões do NDRange, em que cada componente do vetor determina o tamanho de cada dimensão do NDRange.

Os work-items estão organizados dentro de work-groups. O OpenCL escalona a execução dos work-groups, ou seja, ele envia um work-group para a execução, fazendo com que todos os work-items dentro dele sejam executados, e quando esse terminar sua execução um novo work-group com novos work-items é enviado para execução até que todos os work-items sejam executados. O número de dimensões do NDRange, de work-items por dimensão do NDRange e o número de work-items por dimensão de um work-group devem ser definidos pelo *host* antes da chamada de execução do kernel. O número de work-items é definido pela multiplicação o número de work-items por dimensão do NDRange, e a quantidade de work-items por work-groups é definida pela multiplicação das dimensões de um work-group.

Cada work-item é identificado através de um ID único global ou um ID único local dentro de um work-group. Cada work-group é identificado por um ID global único, logo um work-item pode ser identificado ou pelo seu ID global ou pela combinação do seu ID local e do ID do seu work-group. Esses IDs são tuplas de 1, 2 ou 3 índices, variando de acordo com o tamanho do NDRange. Os índices desses IDs vão de  $M$  até  $M + \delta$ ,  $\delta$  sendo o tamanho da dimensão que a tupla representa e  $M$  o um valor inicial para os índices daquela dimensão definido na criação do NDRange pelo *host*.

Para controlar a execução de vários kernels ao mesmo tempo em *devices* diferentes, o OpenCL define um **Context**. Um *Context* é um conjunto de *Devices*, *Kernels*, *Program Objects* e *Memory Objects*. *Devices* e *Kernels* já foram explicados acima, e *Memory Objects* serão explicados na subseção abaixo. *Program Objects* são objetos que tem as seguintes informações:

- O código fonte ou binário que será transformado em funções de um ou mais kernels.
- O número de kernels dentro desse binário ou fonte.

- O log da compilação se o objeto tiver um código fonte.
- Uma referência para o *context* e os *devices* que ele está associado.

Com um *context* criado e inicializado, o *host* controla a execução dele usando um objeto chamado **Command-Queue**. O *host* adiciona comando a uma *command-queue* que está associada a um *context*, e os comandos são executados dentro dos *devices* do *context*. Os comandos são divididos em 3 tipos:

- Comandos de execução de kernel.
- Comandos de transferência de memória
- Comandos de sincronização

Esses comandos podem ser executados sequencialmente, onde um comando na *command-queue* espera todos os anteriores a ele executarem para executar, ou de forma não sequencial, onde a *command-queue* só define a ordem em que os comandos terão sua execução iniciada, mas não se eles devem esperar um comando anterior para rodarem.

### 2.4.3 Modelo de Memória

As threads em execução num kernel tem acesso a 4 locais distintos de memória:

1. Memória Global - Toda thread em execução num kernel tem acesso de escrita e leitura a essa região da memória.
2. Memória Constante - Toda thread em execução num kernel tem acesso de leitura a essa região da memória. Somente o *host* tem acesso de escrita a essa parte da memória.
3. Memória Local - Todas as threads de um work-group tem acesso a essa região da memória. Dependendo do hardware, ela pode ser colocada numa região próxima da região de execução de um work-group ou na memória principal da GPU.
4. Memória Privada - Região privada de uma thread, somente ela tem acesso a está região.

O *host* tem acesso de escrita e leitura na memória global e constante. O kernel tem acesso de escrita e memória em todas as localidades, a menos da local, onde ele só tem acesso de leitura. O OpenCL aplica uma consistência de memória relaxada, ou seja, não existem garantias que o estado de um bloco de memória acessado por um work-item seja igual para qualquer outro work-item acessando aquele bloco. A única consistência de memória garantida pelo OpenCL é de que dentro de uma barreira de um work-group, tanto a memória global quanto a local será igual para todos os work-itens dentro daquele work-group.

A interação entre o modelo de memória do *host* e do *device* é feita através de uma API que ou copia dados para a GPU ou faz um mapeamento de um setor da memória do *host* para um setor da memória do *device*. A passagem da memória é feita por uma *Command-Queue*. A transferência de dados é feita através de um tipo básico de objetos do OpenCL, os *Memory Objects*. eles podem ser de 2 tipos:

- Tipo *buffer* - Representa tipos primitivos como **int** ou *float*, vetores e estruturas definidas pelo usuário. Eles são acessados pelo kernel através de um ponteiro, e são organizados de maneira sequencial na memória. Não existe diferença entre o método de leitura ou escrita de um *buffer*.
- Tipo *image* - Representa um buffer (não o tipo acima, mas o conceito de buffer na computação) de uma imagem ou de uma textura. Existe uma diferença entre os métodos de escrita e leitura de um *image*. Para ler ou escrever é necessário usar funções próprias do OpenCL. As funções de leitura transformam o tipo *image* num vetor de 4 componentes, e as funções de escrita transformam vetores de 4 componentes em uma componente do tipo *image*.

#### 2.4.4 Modelo de Programação

Existem 2 modelos de programação suportados pelo OpenCL:

1. Modelo de Dados - Esse é o modelo mais comum usado pelo OpenCL, onde os índices do espaço de índices que cada work-item recebe definem um mapa one-to-one para os dados que o kernel recebe do *host*. No OpenCL esse modelo é relaxado, já que os work-items podem estar associados a mais de um bloco de dados.
2. Modelo de Tarefas - Esse modelo supõe que somente um work-item será executado em cada device, e que o programador será o responsável por paralelizar a aplicação usando ou vários kernels ou tipos vetoriais de dados que o *device* implemente.

Sobre a sincronização entre *device* e *host* no OpenCL, ela pode ser feita de 2 maneiras:

1. Pela barreira implícita na execução sequencial da *command-queue*
2. Por eventos do OpenCL. Ao rodar um comando numa *command-queue* é possível adicionar um objeto do OpenCL chamado de evento, e podemos esperar esse evento ser concluído no *host* para continuar a execução.

## 3 Atividades Realizadas

### 3.1 Comparação de eficiencia

#### 3.1.1 Como fazer a comparação?

#### 3.1.2 Montagem dos kernels

### 3.2 Comparação das abstrações

### 3.3 Comparação entre os arquivos .ptx

## 4 Resultados

### 4.1 Comparação de eficiencia

#### 4.1.1 Kernel memory-bound

#### 4.1.2 Kernel processing-bound

### 4.2 Comparação das abstrações

#### 4.2.1 Semelhanças

#### 4.2.2 Diferenças

### 4.3 Comparação dos .ptx

## 5 Conclusões

## 6 Bibliografia