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

Thiago de Gouveia Nunes
Surpervior: Prof. Doutor Marcel P. Jackowski

18 de novembro de 2012

# Sumário

# 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 Técnologias

## 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, da NVIDIA, lançada em 1999. O hardware seguia um pipeline de 2 fases, uma 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. Nas placas NVIDIA as threads são agrupadas em blocos, e esses blocos são escalonados para cada SM. Depois, todas as threads dentro de um bloco são divididas em pequenos grupos. Esses grupos chamados de **warp**, e cada warp é executado paralelamente dentro do mesmo SM para qual o bloco foi escalonado. Existe um limite para a quantidade de threads escalonadas para execução dentro de um SM, que é definida pelos recursos que cada thread consome. Por exemplo, não há como executar 10 threads que consomem 10 registradores cada em um SM com 90 registradores.

Outra parte importante do hardware é a memória, que é limitada em relação à da CPU. GPUs tem, em média, 1GB de memória, enquanto CPUs tem 4GB. O acesso a um mesmo bloco de memória é concorrente, mas ao utilizar os caches e leitura/escritas em conjunto 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, e elas podem requisitar em conjunto um mesmo bloco da memória principal, se necessário.

No caso da GTX460 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.

Outro fator limitante é a transferência 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 seja utilizado somente pela GPU ). 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 especifico, 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.

Ao estudar como o código é executado nas GPUs NVIDIA descobrimos a existência de uma máquina virtual chamada de Parallel Thread Execution. Todo kernel é primeiro compilado para um arquivo .ptx que é executado na GPU através da máquina PTX. Ela é utilizada para garantir a retrocompatibilidade de kernels em placas mais antigas.

## 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 inicio de 2007. Atualmente só existe um compilador para CUDA, o nvcc, e ele só da 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 seguira as mesmas instruções, o compilador pode mudar um conjunto de instruções para

outro menos eficiênte, 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 indice 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 independende, ou seja, eles podem executar tanto paralelamente quanto sequencialmente. Com isso, é possivel 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 memoria é separada lógicamente em 4 locais:

- Registradores - Toda variável de uma thread fica em registradores.
- Memória Local - Memória acessivel 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 compratilhada. 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 compatilhada 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 acessivel por qualquer bloco em execução em um *device*. A memoria global não é resetada após a execução de um kernel, então chamadas subsequentes de um mesmo kernel simplesmente leêm 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 possiveis qualificadores são:

- `__device__` Fica na memória global.

- `__constant__` Fica na area 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 Cabapility 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 varios procesadores, dentre eles GPUs e CPUs. OpenCL da 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 possivel, e mantendo auto desempenho, com uma abstração protátil. O OpencL também é uma API para controle de aplicações paralelas em sistemas com processadores heterogéneos. 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 especifico fazem dele a linguagem mais usada para GPGPU fora de GPUs NVIDIA.

O framework do OpenCL pode ser explicado usando 4 modelos hierarquicos, 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 atravez de *comandos* que o *host* manda para o *device*. Os PEs podem executar tanto no modelo de SIMD (Instrução Unica, Multiplos Dados) ou SPMD (Procesuso Unico, Multiplos Dados). No SIMD, todas as threads executão 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 varios 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 indices de até 3 dimensões, onde cada ponto dentro desse conjunto de indices é um work-item. Como visto acima, cada work-item executa o mesmo código, mas com dados diferentes e, existinto pulos condicionais no código, o caminho de execução pode váriar.

Esse conjunto de indices 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 indices, variando de acordo com o tamanho do NDRange. Os indices 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 indices 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:

- Binário que será transformado nas funções de um ou mais kernels;
- O número de kernels dentro desse binário;
- O log da compilação, caso necessário;
- Uma referência para o *context* e os *devices* que ele está associado.

O binário de um *Program Object* pode ser compilado em tempo de execução por uma função do OpenCL.

Com um *conext* 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;

```
clEnqueueNDRangeKernel(queue, kernel, 2, NULL,
    work_dim, local_dim, 0, NULL, &event);
```

- Comandos de transferência de memória;

```
clEnqueueWriteBuffer(queue, columnSize, CL_TRUE,
    0, sizeof(int), &sizeC, 0, NULL, &event);
```

- Comandos de sincronização.

```
clFinish(queue);
```

Esses comandos podem ser executados sequêncialmente, onde um comando na *command-queue* espera todos os anteriores a ele executarem para executar, ou de forma não sequêncial, 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 execuçaõ 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 meória 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 iteração 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 memŕia 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 sequêncial na memória. Não existe diferença entre o mêtodo 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 mêtodos 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 models de programação suportados pelo OpenCL:

1. Modelo de Dados - Esse é o modelo mais comum usado pelo OpenCL, onde os indices do espaço de indices que cada work-item recebe definem um mapa one-to-one para os dados quie 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 supõem 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 implicita na execução sequêncial 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 das abstrações

Como as duas linguagens foram desenvolvidas com base num hardware em comum, as suas abstrações são bem parecidas. Cada uma delas tem uma abstração para as threads executando o kernel ( *work-item* para o OpenCL e *CUDA threads* para o CUDA).

Toda thread, em ambas as linguagens, tem um ID único que a identifica em relação a todas as threads em execução (o ID global) e um ID que a identifica unicamente dentro de um bloco (o ID local). O ID global é uma combinação do ID local com o ID do bloco. É comum usar o ID das threads para identificar quais os dados que ela irá receber. No exemplo desse trabalho, o ID global das threads é usado para determinar qual posição das matrizes ela irá usar nas suas operações.

Para representar a separação das threads nos blocos que serão excalonados para os SM, as duas linguagens implementam uma organização lógica para separar as threads em blocos (*work-group* no OpenCL e *block* no CUDA).

Os blocos são agrupados em um conjunto maior que engloba todas as threads de um kernel. No OpenCL, esse conjunto se chama *NDRange* e no CUDA *Grid*. O OpenCL cria um NDRange por execução do kernel e as dimensões do NDRange e dos work-groups dentro dele são iguais. O espaço de indices das threads de um NDRange pode começar tanto de zero quanto de um número definido pelo usuário, facilitanto operações em posições de memória deslocadas dentro do espaço de memória do problema.

Já no CUDA, os Grids podem ter sua dimensão diferente da dimensão dos blocks. O espaço de indices das threads é limitado a começar do zero. A execução de um kernel é representada por um único grid. Notou-se que o compilador do CUDA devolve um erro ao compilar um kernel que não respeita o tamanho máximo de threads num bloco, enquanto o OpenCL compila, mas o resultado da execução do kernel é sempre inesperado.

Sobre a memória, as duas linguagens deixam a criação e alocação da memória para o *host*. Cada uma delas define uma maneira diferente de tratar a memória. No CUDA a memória do device é tratada como um simples ponteiro. Já o OpenCL cria objetos de memória que serão mapeados para a memória do *device*. As operações de leitura e escrita nesses objetos são feitos através de uma fila de execução e de diretivas auxiliares para a inicialização e alocação.

A memória pode ser direcionada para qualquer um dos 4 espaços do device, usando modificadores especiais na declaração da variável dentro do kernel.

## 3.2 Comparação de eficiencia

### 3.2.1 Como fazer a comparação?

Bem, como fazer a comparação entre essas duas linguagens? A ideia é criar dois tipos de kernels nas duas linguagens, cada tipo para comparar duas caracteristicas importante das linguagens:

- O desempenho ao acessar a memória;
- A capacidade de utilizar o processamento da GPU.

### 3.2.2 Montagem dos kernels

Para testar o desempenho ao acessar a memória, um kernel que faz a cópia de uma matriz de floats foi usado. O código desse kernel tanto em OpenCL:

```
__kernel void MatrixCopy (__global float* a,
                          __global float* b,
                          __global int* rowSize,
                          __global int* columnSize) {
  unsigned int row = get_global_id(0);
  unsigned int column = get_global_id(1);
  b[row+column*(*rowSize)] = a[row+column*(*rowSize)];
}
```

Como em CUDA:

```
__global__ void MatrixCopy (float* MatrixA,
                            float* MatrixB,
                            int rowSize,
                            int columnSize) {
  int row = blockIdx.x*blockDim.x+threadIdx.x;
  int column = blockIdx.y*blockDim.y+threadIdx.y;
  MatrixB[row+column*columnSize] = MatrixA[row+column*columnSize];
}
```

As primeiras linhas de cada kernel determinam qual posição da matriz será copiada usando o ID global da thread. A última linha faz a cópia da matriz A para a matriz B.

Já para testar a capacidade do processamento das linguagens usamos um kernel que faz a multiplicação de duas matrizes de floats e guarda o resultado numa terceira. Em OpenCL:

```
__kernel void matrixmulti(__global float* MatrixA,
                          __global float* MatrixB,
                          __global float* MatrixC,
                          __global int* N) {
  unsigned i = get_global_id(0);
  unsigned j = get_global_id(1);
  unsigned k;
  MatrixC[i*(*N)+j] = 0;
  for( k = 0; k < (*N); k++ )
    MatrixC[i*(*N)+j] += MatrixA[i*(*N)+k]*MatrixB[j+k*(*N)];
}
```

E em CUDA:

```
__global__ void MatrixCopy (float* MatrixA,
                            float* MatrixB,
                            float* MatrixC,
                            int N) {
  int row = blockIdx.x*blockDim.x+threadIdx.x;
  int column = blockIdx.y*blockDim.y+threadIdx.y;
  int k;
  MatrixC[column*N+row] = 0;
  for (k = 0; k < N; k++ )
    MatrixC[column*N+row] += MatrixA[column*N+k]*MatrixB[k*N+row];
}
```

Novamente, as primeiras linha fazem a distribuição da posição de memória para cada thread, enquanto as duas últimas linhas fazem a multiplicação em si.

## 3.3 Os arquivos .ptx

Ao executar um kernel numa GPU NVIDIA ele não é executado diretamente no hardware, na verdade ele passa pela máquina virtual PTX, como dito anteriormente. Ao descobrir esse fato, decidimos comparar os .ptx resultantes da compilação dos nossos kernels para a máquina PTX. Os arquivos ptx usam uma linguagem parecida com o **Assembly**, com comandos especiais para as operações únicas de uma GPU, como operações vetoriais.

Para gerar o PTX de um kernel basta acresentar a flag de compilação *–ptx* para o *nvcc*.

Os comandos PTX podem conter modificadores, por exemplo:

```
ld.param.u64 %rl1, [_Z10MatrixCopyPfS_ii_param_0];
```

em que o comando *ld*, usado para preencher algum endereço de memória, usa o modificador *.param* para carregar um parâmetro do kernel, enquanto no exemplo abaixo,

```
ld.global.f32 %f1, [%rl6];
```

o mesmo comando usando em conjunto com o modificador *.global* irá buscar na memória global o que deve ser carregado para o registrador. Os modificadores *.u64* e *.f32* definem o tipo a ser tratado, no caso *.u* é um **Unsigned Int** e *.f* um **Float**, e o número define o tamanho do endereço.

Nas GPUs NVIDIA, onde várias threads comparlilham o mesmo ponteiro de instrução, cada thread pode modificar os seus registradores para que eles funcionem como registradores booleanos usando a diretiva *.reg .pred*. Com isso, esses registradores podem receber valores de operadores lógicos definidos dentro do PTX. Eles podem ser usandos em conjunto com o símbolo @, que no PTX define se uma instrução será ou não executada baseando-se no valor do operador lógico adjunto a ela. Por exemplo:

```
@%p1 bra BB0_3;
```

A instrução *bra* só será executada se o registrador *p1* conter **TRUE**.

A instrução *bra* define uma separação na estrutura de execução de um warp. Ao encontrar um branch, todas as threads que seguirem esse branch ganham um novo apontador de instruções e continuam a sua execução em paralelo, criando um novo segmento de execução. Por exemplo, se de 16 threads 5 entram em um **if** e o restante passa por ele, temos 2 segmentos de execução. Todas as threads de um mesmo segmento são executadas concorrentemente, enquanto as threads de outro segmento esperam sua vez para executar. Então no nosso exemplo, no primeiro ciclo dos processadores de um SM 5 threads serão executadas, e no próximo ciclo 11 threads, e no seguinte as 5 iniciais, e assim por diante.

Agora que já cobrimos as peculiaridades importantes do PTX, vamos usar como exemplo para estudar a execução de um PTX o kernel de cópia de memória feito em CUDA.

```
.version 3.0
.target sm_20
.address_size 64

.file 1 "/tmp/tmpxft_00001b52_00000000-9_memory.cpp3.i"
.file 2 "memory.cu"

.entry _Z10MatrixCopyPfS_ii(
  .param .u64 _Z10MatrixCopyPfS_ii_param_0,
  .param .u64 _Z10MatrixCopyPfS_ii_param_1,
  .param .u32 _Z10MatrixCopyPfS_ii_param_2,
  .param .u32 _Z10MatrixCopyPfS_ii_param_3
)
{
  .reg .f32   %f<2>;
  .reg .s32   %r<13>;
  .reg .s64   %rl<8>;


  ld.param.u64  %rl1, [_Z10MatrixCopyPfS_ii_param_0];
  ld.param.u64  %rl2, [_Z10MatrixCopyPfS_ii_param_1];
  ld.param.u32  %r1, [_Z10MatrixCopyPfS_ii_param_3];
  cvta.to.global.u64  %rl3, %rl2;
  mov.u32   %r2, %ntid.x;
  mov.u32   %r3, %ctaid.x;
  mov.u32   %r4, %tid.x;
  mad.lo.s32  %r5, %r2, %r3, %r4;
  mov.u32   %r6, %ntid.y;
  mov.u32   %r7, %ctaid.y;
  mov.u32   %r8, %tid.y;
  mad.lo.s32  %r9, %r6, %r7, %r8;
  mad.lo.s32  %r10, %r5, %r1, %r9;
  cvta.to.global.u64  %rl4, %rl1;
  mul.wide.s32  %rl5, %r10, 4;
  add.s64   %rl6, %rl4, %rl5;
  add.s64   %rl7, %rl3, %rl5;
  ld.global.f32   %f1, [%rl6];
  st.global.f32   [%rl7], %f1;
  ret;
}
```

As primeiras linhas,

```
.version 3.0
.target sm_20
.address_size 64
```

definem o ambiente que deve ser preparado na GPU para a execução do kernel. A primeira linha define a versão da máquina PTX, a segunda qual a versão da API de comunicação com a GPU deve ser usada e a última o tamanho do endereçamento a ser usado.
As próximas linhas,

```
.file 1 "/tmp/tmpxft_00001b52_00000000-9_memory.cpp3.i"
.file 2 "memory.cu"
```

associam um inteiro aos arquivos que podem ser usados no kernel. Esses arquivos são acessados usando esse indice, caso necessário. Nesse caso o primeiro *.file* associa a 1 o binário do kernel e a 2 o código fonte.

Agora, ao kernel em si. A próxima linha,

```
.entry _Z10MatrixCopyPfS_ii(
  .param .u64 _Z10MatrixCopyPfS_ii_param_0,
  .param .u64 _Z10MatrixCopyPfS_ii_param_1,
  .param .u32 _Z10MatrixCopyPfS_ii_param_2,
  .param .u32 _Z10MatrixCopyPfS_ii_param_3
)
```

define tanto o ponto de entrada da execução quanto os parâmetros recebidos pelo kernel. A diretiva *.entry* define o ponto de inicio da execução do kernel. Os *.param* definem um meio do kernel acessar os parâmetros passados pelo *Host*, além de configurar o tamanho do endereçamento deles. O último parâmetro da diretiva *.param* é a tag que será usada pelo comando *ld.param* para carregar os parâmetros em registradores.

Já em execução, a primeira coisa que uma thread faz é alocar os registradores que ela irá usar, com a diretiva *.reg*,

```
.reg .f32   %f<2>;
.reg .s32   %r<13>;
.reg .s64   %rl<8>;
```

Os parâmetros dessa diretiva definem o tipo do registrador ( *%f* para **Float** ) e o número de registradores ( *<n>* para *n* registradores ).

A próxima etapa carrega os parâmetros em registradores,

```
ld.param.u64  %rl1, [_Z10MatrixCopyPfS_ii_param_0];
ld.param.u64  %rl2, [_Z10MatrixCopyPfS_ii_param_1];
ld.param.u32  %r1,  [_Z10MatrixCopyPfS_ii_param_3];
```

É importante lembrar que os dois primeiros parâmetros são ponteiros. Ao carregar um ponteiro num registrador, o PTX não sabe se esse endereço faz referência a memória local, global, constante ou a dividida entre as threads, então a próxima instrução é usada para transformar um ponteiro genérico em um ponteiro global:

```
cvta.to.global.u64  %rl3, %rl2;
```

Com os parâmetros necessários carregados e devidamente ajustados, o próximo passo do kernel é calcular o índice da thread. A GPU tem 3 registradores específicos que guardam o índice local de uma thread. As próximas instruções mostram como calculcar o índice global de uma thread num kernel de 2 dimensões:

```
mov.u32   %r2, %ntid.x;
mov.u32   %r3, %ctaid.x;
mov.u32   %r4, %tid.x;
mad.lo.s32  %r5, %r2, %r3, %r4;
mov.u32   %r6, %ntid.y;
mov.u32   %r7, %ctaid.y;
```

```
mov.u32    %r8, %tid.y;
mad.lo.s32  %r9, %r6, %r7, %r8;
mad.lo.s32  %r10, %r5, %r1, %r9;
```

A instrução *mov* preenche um registrador com dados de uma posição não-genérica de memória. A instrução *mad* multiplica o segundo argumento pelo terceiro, soma o quarto à multiplicação e guarda o resultado total em um registrador. Os dois primeiros *mad* calculam os índices da thread, cada um numa dimensão diferente. O terceiro *mad* usa os dois índices e o parâmetro que contém o tamanho da matriz para calcular em qual posição da matriz a thread atual irá operar.

O que resta é copiar os dados de uma matriz para a outra.

```
cvta.to.global.u64  %rl4, %rl1;
mul.wide.s32  %rl5, %r10, 4;
add.s64    %rl6, %rl4, %rl5;
add.s64    %rl7, %rl3, %rl5;
ld.global.f32    %f1, [%rl6];
st.global.f32    [%rl7], %f1;
ret;
```

O último parâmetro usado, o ponteiro para a matriz que será copiada, é transformado pela instrução *cvta*. Os dois *add* adicionam os índices da thread ao ponteiro das matrizes, criando um offset que referência a posição que aquela thread deve usar para a cópia. O *ld* carrega o valor dessa posição num registrador que depois é copiado para a matriz destino usando a instrução *st*, e por fim o kernel finaliza. Não foi encontrado nada na documentação do PTX nem nenhum motivo aparente no kernel que explique a multiplicação da posição da matriz que será operada por quatro, mas essa operação é constante nos dois tipos de kernel e tanto no OpenCL quanto no CUDA.

# 4 Resultados

## 4.1 Comparação de eficiencia

Cada kernel foi executado 3000 vezes em sequência. A GPU tem um mecânismo que faz um cache do código do kernel e também da memória que ele utiliza, então podemos desconsiderar a comunicação da CPU com a GPU neste caso, deixando os nossos resultados mais próximos do tempo de execução dos kernels.

É importante levar em conta que a GPU não estava rodando somente o kernel, já que os drivers necessários para a execução do mesmo estão atrelados ao X Window System, então o kernel sofreu interrupções na GPU, para que a interface gráfica do Ubuntu fosse renderizada.

### 4.1.1 Kernel memory-bound

Os resultados dos kernels memory-bound:

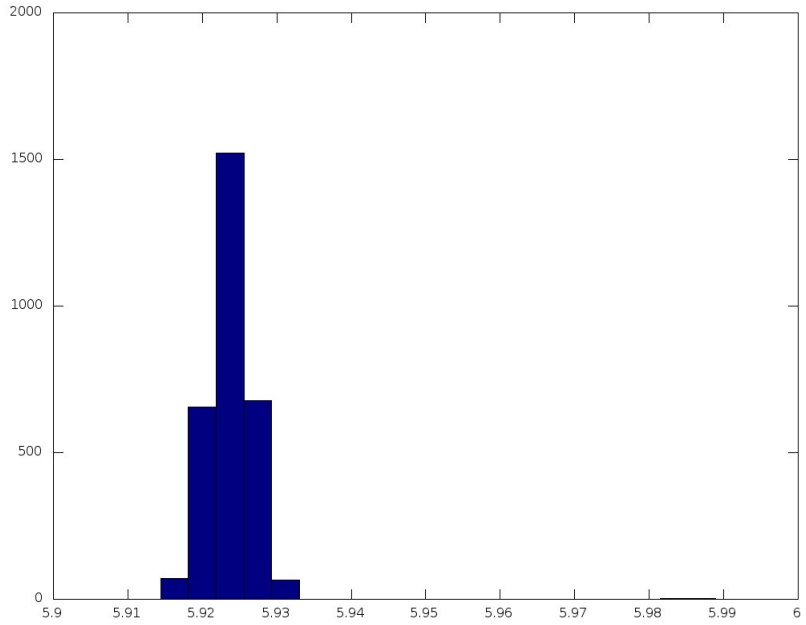


Figura 1: Kernel Memory-Bound CUDA

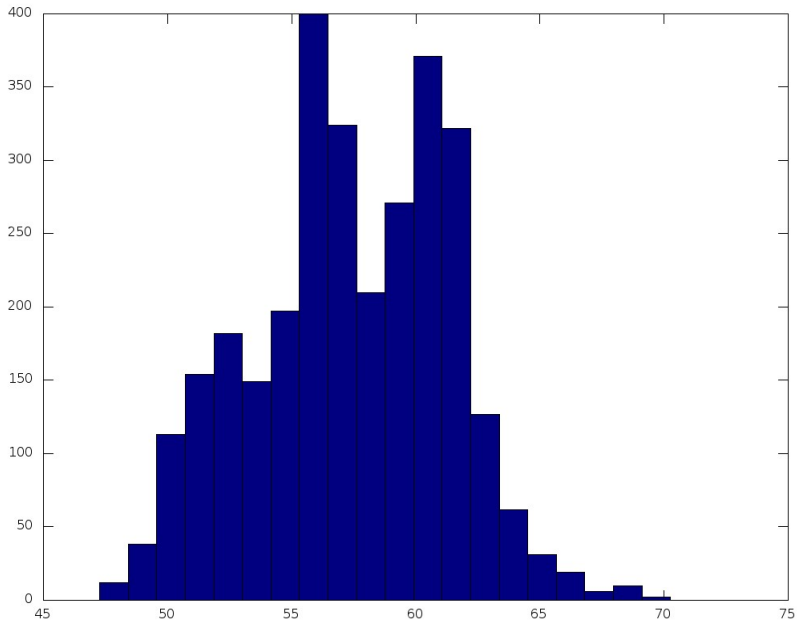


Figura 2: Kernel Memory-Bound OpenCL

#### 4.1.2 Kernel processing-bound

## 4.2 Comparação dos .ptx

# 5 Conclusões

# 6 Bibliografia