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

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

1 de setembro 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).

## 1.2 Objetivos

O objetivo do estudo é comparar a eficiência de programas escritos nessas duas linguagens rodando em uma placa NVidia GeForce GTX 260.

## 1.3 Problemas a serem resolvidos

Para realizar essa comparação de eficiência, devemos entender como as linguagens funcionam, suas semelhanças e diferenças, e criar um método que seja justo para comparar programas semelhantes escritos nas duas 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

Os supercomputadores foram criados para rodar as aplicações que executavam esses objetivos. 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 seus processadores.

## 2.2 GPGPU

As GPUs nasceram da necessidade de acelerar a taxa com que as imagens eram exibidas no monitor. Por isso, sua arquitetura é diferente de uma CPU e temos que levar isso em conta ao escrever programas para elas. O paradigma usado no processamento das GPUs é conhecido como Stream processing.

### 2.2.1 Stream processing

Esse paradigma cria um ambiente que simplifica tanto o software quanto o hardware, mas limita a quantidade de problemas paralelizáveis que odem ser resolvídos nesse ambiente.

Stream processing se baseia em vários processadores, cada um com um pequeno cache de acesso próprio, rodando um mesmo conjunto de instruções ( kernels ). Cada processador irá aplicar o mesmo kernel em um conjunto diferente de dados, e todos eles executarão seus kernels em paralelo. No final, todo o resultado é combinado, assim resolvendo o problema inicial.

Vamos usar de exemplo a GPU usada para os testes desse trabalho, uma GeForce GTX 260. Ela contém 192 processadores, separados em blocos de 8 processadores, cada um com um cache de 16Kb. Cada bloco tem um cache próprio, e a placa tem 896Mb de memória. Conhecer a arquitetura do hardware em que vamos programar é importante para conseguir o máximo de desempenho possível.

É fácil ver como esse paradigma é eficiente para resolver integrais ou multiplicações de matrizes, onde não há dependencia de dados entre instâncias dos kernels e os dados podem ser fácilmente distribuidos pelos caches. Mas esse paradigma pode não ser o mais eficiente para resolver um problema clássico de programação paralela, o dos Filósofos Famintos, onde todos os processos dependem de mais dois processos.

O problema dessa depêndencia é a compartilhação de memória entre os dependentes. Ao modificar um pedaço de memória compartilhada, todos os processos que utilizam essa memória devem parar sua execução e fazer a releitura do cache, que deve ser feita de maneira sequencial.

## 2.3 CUDA

### 2.3.1 Modelo de Memória

### 2.3.2 Modelo de Execução

### 2.3.3 Modelo de Plataforma

### 2.3.4 Modelo de Programação

## 2.4 OpenCL

### 2.4.1 Modelo de Memória

### 2.4.2 Modelo de Execução

### 2.4.3 Modelo de Plataforma

### 2.4.4 Modelo de Programação

# 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

# 7