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

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

7 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

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 pixels como para vértices, foi lançada para o console XBox 360. Para unificar os 2 processos do pipeline num único processador foi necessário generalizar esse processador, abrindo as portas para programas mais genericos rodarem na GPU.

A GPU que foi usada para os testes, uma GeForce GTX260, usa a arquitetura Tesla da NVIDIA, que segue a seguinte estrutura: Ela é composta de multiprocessadores, cada um com 32 processadores. Multiprocessadores são construidos para executar centenas de threads concorrentemente. Os multiprocessadores dividem todas as suas threads em pacotes de 32 em 32, nomeados de *warp*. Todas as threads de um *warp* começam executando a mesma instrução, mas cada uma tem seu contador de instrução e assim cada uma pode seguir um caminho diferente. Os processadores não tem previsão de fluxo como as CPUs. Cada *warp* é ordenado pelo *warp scheduler* dentro de um mesmo multiprocessador. É crucial saber que se um grupo de threads se separa do fluxo de execução padrão, por exemplo se metade entra num if e o resto no else desse if, o multiprocessador irá dividir o tempo entre os fluxos. Isso quer dizer que em um ciclo de execução, somente as threads que entraram no if serão executadas, e no próximo ciclo somente as do else.

Isso impõe algumas complicações na escrita de programas. Temos um desempenho maior em programas que distribuem a memória de modo que threads rodando no mesmo multiprocessador utilizem posições que estão no mesmo bloco da memória, evitando que o gerenciador de memória tenha que fazer leituras na memória principal.

Um dos problemas com GPGPU é a memória da GPU. Ela é bem mais limitada que a acessivel pela CPU. GPUs tem, em média, 1GB de memória, enquanto CPUs tem 4GB. Outro fator é a transferencia da memória principal

do computador para a memória principal da GPU. A transmissão é feita por um barramento PCI Express na maioria dos casos, com velocidades de até 16GB/s dado que o barramento esteja sendo usado somente para isso, o que é pouco provável. Essa transmissão é a parte mais lenta de todo o processo de execução na GPU. Em alguns casos é mais viável fazer algumas operações de baixar eficiência numa GPU do que retornar os dados computados numa GPU para a CPU e passá-los de volta para a GPU para mais operações e retornar esses dados para a CPU.

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.

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 segmentados para caches diferentes. 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.

Conhecer tanto a arquitetura como a capacidade de processamento e memória do hardware em que vamos programar é importante para conseguir o máximo de desempenho possível. É importante também modificar o problema em questão para adequá-lo a esse paradigma.

Para obter o máximo desempenho do programa, é preciso analisar como a linguagem escolhida abstrai o hardware da GPU.

2.3 CUDA

Compute Unified Device Architecture (CUDA) é uma arquitetura de programação para GPUs criada pela NVidia. A versão 1.0 foi disponibilizada no inicio de 2007. Atualmente só existe um compilador para CUDA, o nvcc, e ele só da suporte para GPUs NVidia.

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 hardware suporte e converte o código para usar as instruções corretas. Novamente, 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.

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.

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