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

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

12 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 460.

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 paralelos genericos executarem na GPU.

O processamento da placa usada para os testes desse trabalho, a GeForce GTX460 que usa a arquitetura Fermi. Essa arquitetura separa o fluxo de execução baseando-se no tipo de operações que serão executadas nela. Existe um fluxo para operações gráficas e outro para operações genericas. 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). Cada SM tem 48 processadores. 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 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.

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 memoria 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 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. 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, ou em alguns casos, algumas instruções não existem em versões mais antigas do hardware.

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 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 parametros 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 númeor 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 supoem 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.

2.3.2 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.

2.3.3 Modelo de Plataforma

SOBRE INICIALIZAÇÃO DO CUDA
ALOCAÇÃO DE MEMORIA
INICIALIZAÇÃO DAS THREADS, KERNEL

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