Insper

22 - Introdução a GPGPU

SuperComp 2018/2

Igor Montagner, Luciano Soares

Neste primeiro roteiro sobre GPU iremos instalar as ferramentas necessárias e criar programas simples acelerados pela GPU. Nossos objetivos de aprendizagem são:

1. Compilar programas para GPU
2. Alocar dados na GPU e transferir dados CPU↔GPU usando `cuda::thrust`
3. Acelerar computações simples usando as operações de `transform` e `reduce`

Neste roteiro iremos calcular algumas estatísticas simples usando séries temporais de preços de ações disponíveis nos arquivos `stocks-google.txt` e `stocks2.csv`.

Nesta parte introdutória do curso usaremos a biblioteca `cuda::thrust`. Ela possui um pequeno conjunto de operações otimizadas para GPU e que podem ser customizadas para diversos propósitos. A parte mais avançada do curso usará *CUDA C*, porém manteremos as funções da `thrust` para alocação e transferência de dados.

Parte 0 - instalação e compilação usando `nvcc`

Eu tenho GPU NVIDIA no meu PC

Instruções fáceis: os repositórios oficiais do *Ubuntu* já contém o pacote `nvidia-cuda-toolkit` pronto para instalação via *apt*. A versão disponibilizada não é a mais atual (`9.1.85` vs `10.0`), mas tudo funciona de maneira integrada e não é necessário instalar nada manualmente. Esta versão será suportada pelo curso.

Instruções não tão fáceis: Visitar o site da NVIDIA, baixar o pacote `.deb` e instalar manualmente. Estas instruções não são difíceis, mas como pode ser necessário instalar novos drivers de vídeo isto pode dar algum trabalho.

Eu não tenho GPU NVIDIA no meu PC

Os notebooks Alienware estão disponíveis para empréstimo na 404. Outra opção, disponível somente durante horário de aulas, é usar máquinas na AWS com GPU.

Compilação

Assim como *MPI*, para compilar programas que usem a GPU precisamos invocar um compilador diferente. Este compilador identifica quais porções do código deverão ser compiladas para a GPU. O restante do código, que roda exclusivamente na CPU, é passado diretamente para um compilador *C++* regular e um único executável é gerado contendo o código para CPU e chamadas inseridas pelo `nvcc` para invocar as funções que rodam na GPU.

Exercício: verifique que sua instalação funciona compilando o arquivo abaixo.

```
>$ nvcc -std=c++11 exemplo1-criacao-iteracao.cu -o exemplo1
```

Parte 1 - transferência de dados

Como visto na expositiva, a CPU e a GPU possuem espaços de endereçamento completamente distintos. Ou seja, a CPU não consegue acessar os dados na memória da GPU e vice-versa. A `thrust` disponibiliza somente um

tipo de container (`vector`) e facilita este gerenciamento deixando explícito se ele está alocado na CPU (`host`) ou na GPU (`device`). A cópia CPU↔ GPU é feita implicitamente quando criamos um `device_vector` ou quando usamos a operação de atribuição entre `host_vector` e `device_vector`. Veja o exemplo abaixo:

```
thrust::host_vector<double> vec_cpu(10); // alocado na CPU

vec1[0] = 20;
vec2[1] = 30;

// aloca vetor na GPU e transfere dados CPU->GPU
thrust::device_vector<double> vec_gpu (vec_cpu);

//processa vec_gpu

vec_cpu = vec_gpu; // copia dados GPU -> CPU
```

A `thrust` usa iteradores em todas as suas funções. Pense em um iterador como um ponteiro para os elementos do array. Porém, um iterador é mais esperto: ele guarda também o tipo do vetor original e suporta operações `++` e`*` para qualquer tipo de dado iterado de maneira transparente.

Vetores `thrust` aceitam os métodos `v.begin()` para retornar um iterador para o começo do vetor e `v.end()` para um iterador para o fim. Podemos também somar um valor `n` a um iterador. Isto é equivalente a fazer `n` vezes a operação `++`. Veja abaixo um exemplo de uso das funções `fill` e `sequence` para preencher valores em um vetor de maneira eficiente.

```
thrust::device_vector<int> v(5, 0); // vetor de 5 posições zerado
// v = {0, 0, 0, 0, 0}
thrust::sequence(v.begin(), v.end()); // inicializa com 0, 1, 2, ....
// v = {0, 1, 2, 3, 4}
thrust::fill(v.begin(), v.begin()+2, 13); // dois primeiros elementos = 13
// v = {13, 13, 2, 3, 4}
```

Consulte o arquivo *exemplo1-criacao-iteracao.cu* para um exemplo completo de alocação e transferência de dados e do uso de iteradores.

Exercício: variáveis do tipo `host_vector` funcionam como `std::vector`. Crie um programa que lê os valores de fechamento de ações do google nos últimos 10 anos (obtidos do site da *Nasdaq*) para um `host_vector` e os copia para um `device_vector`.

Exercício: a criação de um `device_vector` é demorada. Meça o tempo que a operação de alocação e cópia demora e imprima na saída de erros.

Parte 2 - transformações e reduções

A `thrust` disponibiliza dois tipos básicos de operações aceleradas em *GPU*:

1. *transformações* tomam como entrada um (ou dois vetores) e fazem operações *elemento a elemento* entre eles, colocando o resultado em novo vetor. Com isto é possível, por exemplo, computar expressões algébricas simples entre vetores ou até mesmo calcular diferenças entre posições vizinhas.
2. *reduções* tomam como entrada um vetor e devolvem um único número como saída. Exemplos incluem calcular a soma ou máximo de um vetor.

Reduções

Assim como no OpenMP, podemos fazer operações de redução que "resumem" um vetor em um único valor numérico. Podemos fazer reduções aritméticas usando a função `thrust::reduce`

```
val = thrust::reduce(iter_comeco, iter_fim, inicial, op);
// iter_comeco: iterador para o começo dos dados
// iter_fim: iterador para o fim dos dados
// inicial: valor inicial
// op: operação a ser feita.
```

Um exemplo de uso de redução para computar o máximo pode ser visto aqui. Outro exemplo, que calcula a soma, pode ser visto aqui. Outras operações aritméticas para redução pode ser encontrada aqui.

Dica: além do `reduce` genérico, `thrust` também contém outras funções de redução que podem ser convenientes. Veja esta lista na documentação.

Exercício: Continuando o exercício anterior, calcule as seguintes medidas. Não se esqueça de passar o `device_vector` para a sua função `reduce`

1. O preço médio das ações nos últimos 10 anos.
2. O preço médio das ações no último ano.
3. O maior e o menor preço da sequência.

Exercício: Todos os algoritmos da `thrust` podem ser rodados também em *OpenMP* passando como primeiro argumento `thrust::host`. Modifique o seu exercício acima para fazer as mesmas chamadas porém usando `OpenMP` e meça o tempo das duas implementações. Separe o tempo de cópia para GPU e o de execução em sua análise.

Importante: O principal critério para paralelização em GPU é o tamanho dos dados: para poucos dados a velocidade de execução das operações em paralelo não vale o tempo de cópia dos dados CPU->GPU->CPU.

Transformações ponto a ponto

Além de operações de redução também podemos fazer operações binárias ponto a ponto tanto entre vetores e operações unárias que transformam um só vetor. O `thrust` dá o nome de `transformation` para este tipo de operação.

```
// para operações entre dois vetores iter1 e iter2. resultado armazenado em out
thrust::transform(iter1_comeco, iter1_fim, iter2_comeco, out_comeco, op);
// iter1_comeco: iterador para o começo de iter1
// iter1_fim: iterador para o fim de iter1
// iter2_comeco: iterador para o começo de iter2
// out_comeco: iterador para o começo de out
// op: operação a ser realizada.
```

Um exemplo concreto pode ser visto abaixo. O código completo está em `exemplo2-transform.cu`

```
thrust::device_vector<double> V1(10, 0);
thrust::device_vector<double> V2(10, 0);
thrust::device_vector<double> V3(10, 0);
thrust::device_vector<double> V4(10, 0);
// inicializa V1 e V2 aqui

//soma V1 e V2
thrust::transform(V1.begin(), V1.end(), V2.begin(), V3.begin(), thrust::plus<double>());

// multiplica V1 por 0.5
thrust::transform(V1.begin(), V1.end(),
                  thrust::constant_iterator<double>(0.5),
                  V4.begin(), thrust::multiplies<double>());
```

Exercício: Vamos agora trabalhar com o arquivo `stocks2.csv`. Ele contém a série histórica de ações da Apple e da Microsoft. Seu objetivo é calcular

1. a diferença média entre os preços das ações AAPL - MSFT
2. a variância das diferenças

Dicas:

1. É necessário executar várias operações de `transform` e `reduce`.
2. Consulte a documentação do `thrust::constant_iterator(10)`. Como você poderia usá-lo para computar a variância.

Exercício: cada chamada da `thrust` resulta em uma chamada de função na GPU, que tem um custo fixo não desprezível. Por isto foi criada a função `thrust::transform_reduce`, que combina ambas operações em uma só chamada e evita parte deste custo. Modifique seu código do exercício anterior para usar esta função.