Insper

15 - aplicação de GPGPU usando cuda::thrust

Igor Montagner Luciano Soares

Introdução

Vimos nas últimas aulas como criar kernels em *CUDA C* e como gerenciar em detalhes a execução na GPU. Porém, frequentemente precisamos de uma API mais alto nível para trabalhar com vetores e matrizes e criar do zero um kernel para tudo é inviável. Além disto, o investimento de tempo é muito grande e seria interessante uma ferramenta de prototipação de código em GPU, permitindo que verifiquemos se a implemententação pode trazer benefícios.

O `cuda::thrust` é uma biblioteca C++ já inclusa no *CUDA SDK* que permite a prototipação de código GPU sem a complexidade de criar um kernel e tratar com a GPU em baixo nível. Claro que isto tem um custo: código em *CUDA C* é potencialmente mais rápido se calibrarmos bem todos os parâmetros de execução. Porém, existem algumas operações padrão que não faz sentido nenhum reprogramar toda vez e que gostaríamos de aproveitar, como operações ponto a ponto e reduções.

Vamos trabalhar neste handout com esta biblioteca e focar em transformar código *CUDA C* complexo em versões mais legíveis usando (em parte) `cuda::thrust`.

Tipos host e device

Uma grande vantagem de utilizar `thrust` é o gerenciamento de memória presente na biblioteca. Um ponto confuso quando trabalhamos com `cudaMalloc` é que não existe nada que indique, no tipo da variável, se ela está alocada no host ou na GPU. `thrust` trabalha com dois tipos de vetores: `device_vector` e `host_vector` e checa os tipos passados na compilação. Ou seja, não podemos misturar os dois tipos na mesma chamada e o tipo das variáveis diz explicitamente onde elas estão alocadas. Veja o exemplo abaixo.

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

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

thrust::host_vector<double> vec_gpu (10); // alocado na GPU
```

Outro ponto prático porém perigoso é que `thrust` pode fazer cópias GPU <-> CPU implicitamente. O código a seguir fazer a cópia de dados CPU->GPU na atribuição.

```
vec_gpu = vec_cpu; // copia o conteúdo da CPU para GPU
thrust::device_vector<double> vec2_gpu (vec_cpu); // também transfere para GPU
```

Existem diversas funções que inicializam vetores em `thrust`. Veja os exemplos de código abaixo. Todos os códigos são válidos tanto para vetores `device` como `host`.

```
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 = 3
// v = {13, 13, 2, 3, 4}
```

Iteradores

Nas funções `fill` e `sequence` acima usamos *iteradores* para representar em quais intervalos do vetor uma função deve ser aplicada. 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 `++`, `*` 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 o arquivo `inicializacao-iteracao.cu` desta aula para um exemplo completo com estas funções. Note que a criação do vetor `device` é bem mais lenta que a criação do vetor `host`.

Exercício: modifique o arquivo acima para que o vetor, que contém 10 posição valha `{5, 5, 5, 7, 7, 7, 90, 0, 1, 2}`. Você só pode usar as funções `fill` e `sequence` neste exercício.

Algoritmos em `thrust`

Existem, basicamente, dois tipos de algoritmos em `thrust`: *transformations* e *reductions*. Ambos tipos de algoritmos são totalmente baseados em iteradores e podem ser customizados de diversas maneiras baseado somente na ordem de iteração dos elementos.

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.

Exercício: Usaremos os dados `stocks-google.txt`, que contém os valores de fechamento das ações do google nos últimos 10 anos (obtidos do site da *Nasdaq*). Gostaria de saber o seguinte:

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.

Para isto, faça um programa `stocks.cu` que lê o arquivo acima da entrada padrão e calcula as medidas acima.

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: a partir deste ponto recomendamos medir o tempo de execução e cópia de todas as operações feitas.

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 `exemplo-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. Leia cada sequência em um vetor separado, calcule a diferença entre elas e calcule a diferença média e a variância das diferenças. Este e os próximos exercícios devem ser feitos em um arquivo `stocks2.cu`.

Dica: Podemos criar iteradores "na mão" usando alguns tipos especiais da `thrust`. Por exemplo, se quisermos somar 10 a todos elementos do vetor podemos fazer um transform com o segundo elemento sendo um `thrust::constant_iterator(10)`. Pense em como isto pode ser usado para calcular a variância ;)

Transformações customizadas

Além das operações unárias e binárias disponíveis podemos criar também nossas próprias operações. A sintaxe é bastante estranha, mas ainda é mais fácil que usar kernels do *CUDA C*. A operação abaixo calcula soma o valor de dois vetores, elemento a elemento, e divide por um valor especificado pelo usuário.

```
struct custom_transform
{

    const double param;

    custom_transform(double d): double_param(d) {}

    __host__ __device__
        double operator()(const double& x, const double& y) const {
            return (x + y)/d;
        }
};
```

Exercício: reescreva a variância do exercício acima usando uma transformação customizada. Para ter ainda mais desempenho pesquise como usar `thrust::transform_reduce` para fazer, ao mesmo tempo, a transformação e o somatório do reduce. Meça o tempo de sua implementação e compare com a do exercício anterior, que usa várias chamadas a `transform` e `reduce`.

Exercício: uma informação importante é saber se o valor de uma ação subiu no dia seguinte. Isto seria útil para, por exemplo, fazer um sistema de Machine Learning que decide compra e venda de ações. Porém, gostaria de saber se houve um aumento significativo, ou seja, quero gerar um vetor que possui 1 se o aumento foi maior que 10% e 0 caso contrário. Complemente seu programa para gerar uma

Dica: uma maneira de fazer isto é criar uma transformação customizada.

Desafio: a média móvel é um modelo de análise de séries temporais usado para verificar se a tendência de uma série é de subida ou queda. Ela é calculada usando a média dos últimos n eventos para prever o evento atual. Implemente este modelo nos dados das ações do google usando os últimos 7 dias para prever o dia atual.