Insper

23 - Introdução a GPGPU II

SuperComp 2018/2

Igor Montagner, Luciano Soares

Na última aula fizemos exercícios com `cuda::thrust` para alocar vetores na GPU, transferir dados e executar algumas operações simples usando `transform` e `reduce`. Nesta aula iremos trabalhar com operações customizadas escritas em C++.

Parte 0 - 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.

Parte 1 - operações entre imagens

Toda operação que fizemos até agora foi elemento a elemento. Nesta seção vemos um exemplo de como usar tuplas e `zip_iterator` para processar elementos vizinhos. É essencial conseguir criar transformações customizadas, então se você está em dúvida reveja o handout da aula passada.

Uma tupla em `thrust` pode ter até 10 elementos. Seu uso é baseado (novamente) em templates:

```
thrust::tuple<double, double> pair;
thrust::get<0>(pair) // primeiro elemento
thrust::get<1>(pair) // segundo elemento
```

Ao usar a função `thrust::zip_operator` com iteradores criamos um novo iterador que retorna tuplas! Como os tipos ficam rapidamente gigantesctos, é muito comum usar typedefs. Veja abaixo.

```
typedef thrust::device_vector<int>::iterator IntIter;
typedef thrust::tuple<IntIter, IntIter> Int2IteratorTuple
typedef thrust::zip_iterator<Int2IteratorTuple>  Int2Iterator

thrust::device vector v1, v2;

//inicializa v1 e v2

Int2Iterator iter = thrust::zip_operator(make_tuple(v1.begin(), v2.begin()));
```

Se fizermos uma transformação customizada ela receberá como argumento o tipo `thrust::tuple<int, int>` no lugar deste iterador:

```
typedef thrust::tuple<int, int> Int2;

struct custom_op {
    __host__ __device__
    int operator() (Int2 a) {
        return thrust::get<0>(a) + thrust::get<1>(a);
    }
}

...

//exemplo de operação unária
thrust::transform(iter.begin(), iter.end(), dest.begin() custom_op());
```

Usando iteradores em vetor usando deslocamentos e tuplas podemos criar transformações que analisam elementos vizinhos! Veja um exemplo no arquivo `stocks-up.cu` disponível no github.

Exercício: abra o código `imagem-thrust.cu`. Ele já contém uma leitura de imagens *PGM* para vetores na GPU. Use tuplas e `zip_operator` (se necessário) para implementar as seguintes operações de imagens:

1. Quantização (limitação do número de cores)
2. Borramento
3. Filtro de bordas usando laplace

Note que já implementamos estas operações anteriormente usando kernels *CUDA C*. Desta vez o desafio é usar `thrust` e verificar se foi mais fácil/prático do que anteriormente.

Desafio: faça seu programa funcionar tanto para imagens em níveis de cinza como para imagens coloridas.