Insper

24 - Introdução a CUDA C

SuperComputação - 2018/2

Igor Montagner, Luciano Soares

Parte 0 - `thrust` ↔ CUDA C

Nas últimas aulas alocamos memória na GPU e transferimos dados usando `thrust::device_vector`. Todo kernel em CUDA C recebe ponteiros diretamente para os dados. Podemos obter estes ponteiros com as seguintes chamadas.

```
thrust::device_vector<int> data_on_gpu;
int *raw_data_on_gpu = thrust::raw_pointer_cast(data_on_gpu.data());
```

Desta maneira podemos continuar usando a `thrust` para gerenciar nossos dados e chamar kernels em CUDA C quando for necessário. Utilize esta estratégia nas próximas partes do handout.

Parte 1 - Processamento de vetores

Um kernel em *CUDA C* é uma função escrita em *C* com algumas características especiais:

1. é declarada com o modificador `__global__` na frente, o que indica que ela será rodada na GPU mas pode ser invocada a partir de código rodando na CPU
2. apesar de ser projetada para agir sobre um vetor (ou matriz ou array 3D), trata somente um elemento por invocação, determinando qual elemento é computado usar uma API específica.
3. é invocada usando uma sintaxe especial para funções que rodem na GPU.

O código abaixo, que faz a soma de dois vetores, exemplifica estas três características.

```
__global__ void add(double *a, double *b, double *c, int N) {
    int i=blockIdx.x * blockDim.x + threadIdx.x;
    if (i<N) {
        c[i] = a[i] + b[i];
    }
}

....

// dentro da função main
int blocksize = 256;
thrust::device_vector<double> A_d(A), B_d(B), C_d(n);
add<<<ceil(n/blocksize),blocksize>>>(
                                thrust::raw_pointer_cast(A_d.data()),
                                thrust::raw_pointer_cast(B_d.data()),
                                thrust::raw_pointer_cast(C_d.data()),
                                n
                                );
thrust::host_vector<double> C(C_d);
```

Exercício: analise o código acima (arquivo *exemplo-add.cu*) e responda. Para facilitar as contas adote `n=1500`.

1. Em quantos blocos será dividida a adição de dois vetores considerando `blocksize=256`?

2. Dada a chamada da função feita no `main`, quanto vale `blockDim.x` na função `kernel_add` supondo que `n=1500`? Quais são os valores máximos e mínimos para `blockIdx.x` e `threadIdx.x`?

3. Por que é necessário checar se o índice `idx` é menor que o tamanho do vetor `n`? O que aconteceria se este checagem não fosse feita?

Exercício: com base no exemplo acima, faça um kernel em *CUDA C* para calcular a variância de um vetor (gigante). Você pode deverá

1. alocar um vetor para os dados e um vetor para guardar os resultados parciais
2. usar uma operação `reduce` para computar a média.
3. cria um kernel em *CUDA C* para computar $\frac{(x_i - \overline{x})^2}{N}$
4. usar uma operação `reduce` para fazer a soma final.

Como dado de entrada você pode utilizar o arquivo `stocks-google.txt` usado nas últimas atividades.

Parte 2 - Processamento de matrizes

Ao processar matrizes e todo tipo de dado 2D pode ser conveniente dividir os dados em um grid **bidimensional** como o abaixo. Cada bloco possui largura e altura, uma posição na direção `x` e uma posição na direção `y`. Cada thread dentro do bloco possui uma posição `x` entre 0 e a largura do bloco e uma posição `y` entre 0 e a altura do bloco.

Para usar um grid bidimensional são necessárias modificações na chamada da função. Como grids em CUDA C podem ter no máximo três dimensões é necessário passar um objeto do tipo `dim3` para o número de blocos criados e outro do tipo `dim3` para o tamanho de cada bloco. Veja o exemplo abaixo,

```
__global__ void add_one(int *input, int height, int width) {
    int i=blockIdx.x*blockDim.x+threadIdx.x;
    int j=blockIdx.y*blockDim.y+threadIdx.y;

    if (i < height && j < width) {
        input[i * width + j] += 1;
    }
}


// dentro do main
dim3 dimGrid(ceil(nrows/16.0), ceil(ncols/16.0), 1);
dim3 dimBlock(16, 16, 1);

add_one<<<dimGrid,dimBlock>>>(image_raw_pointer, nrows, ncols);
```

Exercício: usando a API de leitura de imagens no arquivos *image.cu/h*, leia uma imagem do disco e faça um filtro de borramento nela. Para cada pixel da imagem você deverá

1. calcular a média dos valores dos pixels em sua vizinhança imediata (totalizando 9 valores - 8 vizinhos mais o próprio pixel)
2. atribuir a uma imagem de saída este valor.
3. salvar a imagem com o nome original mais "-blur".

Exercício: um filtro de bordas muito simples é o operador de Laplace. Faça um novo kernel em CUDA C que calcula o laplaciano da imagem e o salva em uma nova imagem com nome original mais "-edges".

Exercício: Imagens com cores são representadas por arrays com dimensão $N \times M \times C$ onde N é o número de linhas, M o de colunas e $C = 3$. Ou seja, para cada pixel são armazenadas três intensidades (RGB -> vermelho, verde e azul). Faça um programa que lê uma imagem colorida `in` e a converte em uma imagem níveis de cinza `out` segundo a seguinte regra.

$$out(i, j) = 0.6 \times in(i, j, 1) + 0.3 \times in(i, j, 0) + 0.1 \times in(i, j, 2)$$

Parte 3 - Profiling de código GPU

Nos últimos projetos usamos as funções do cabeçalho `<chrono>` para medir o tempo gasto pelo nosso programa. Quando se trata de código rodando em GPU estas medições não são mais confiáveis pois a execução de código CUDA não é sequencial. Precisamos, então, de um timer que esteja integrado ao mecanismo de execução de código da GPU. Veja abaixo um exemplo de uso da estrutura `cudaEvent_t`, que permite registrar a ocorrência de eventos na GPU e calcular quanto tempo passou entre pares de ocorrências.

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start);
// código que desejamos medir o tempo
cudaEventRecord(stop);

// outras operações podem ir aqui

cudaEventSynchronize(stop); // espera até o evento stop ser executado

float elapsed_time;
cudaEventElapsedTime(&elapsed_time, start, stop);

cudaDestroy(&start);
cudaDestroy(&stop;
```

Finalmente, podemos usar o programa `nvprof` para

Exercício: rode o `exemplo-add` usando `nvprof`. Do tempo total de execução quanto foi gasto com cópias Host -> GPU? E GPU ->host? Quanto tempo foi gasto na execução do kernel `add`?