GPU Teaching Kit

Accelerated Computing

Introdução à Programação em CUDA

Matheus S. Serpa – msserpa@inf.ufrgs.br

*Baseado no material fornecido pelo NVIDIA Educator Program

O que é CUDA

CUDA é uma plataforma de computação paralela, criada para que desenvolvedores possam usar de forma mais precisa e livre o alto potencial de processamento paralelo proporcionado por GPUs da nVIDIA

Extensão da linguagem C, que permite o uso de GPUs:

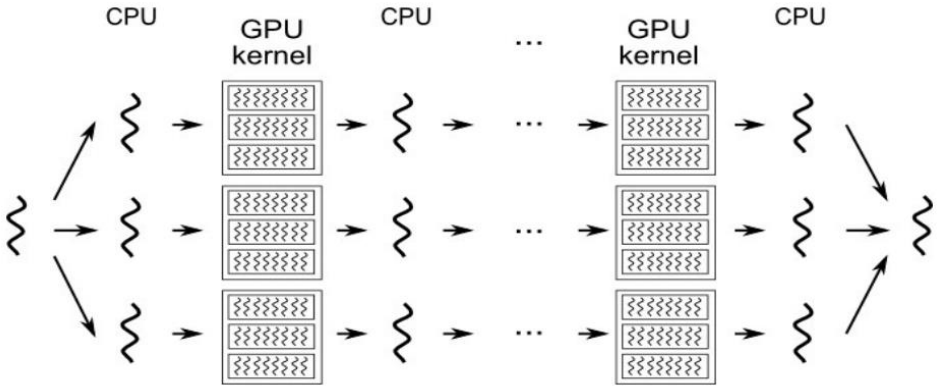
Definição de *kernels* que são executados na GPU

API com funções, que permitem o gerenciamento da memória da GPU e outros tipos de controle

Modelo de Execução

Execução do programa controlada pela CPU que pode lançar kernels, que são trechos de código executados em paralelo por múltiplas threads na GPU

A execução de programas CUDA é compostas por ciclos CPU -> GPU -> CPU -> GPU … CPU

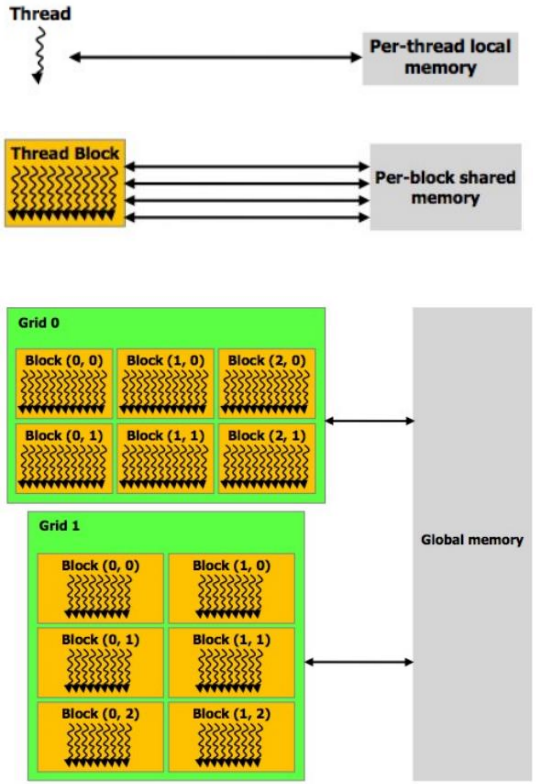


Hierarquia de Threads e Memória

Cada execução do kernel é composta por:

Grade → blocos → threads

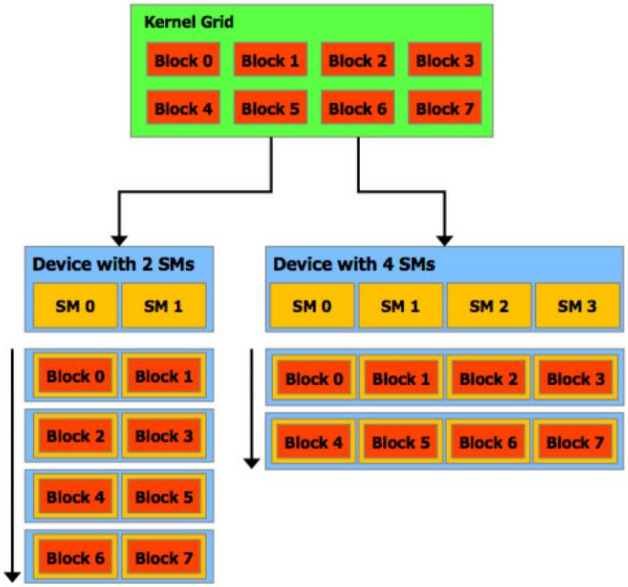
- Registradores por *thread*
- Memória compartilhada por bloco
- Memória global acessível a todas as threads



Execução de Aplicações

Cada bloco é alocado a um multiprocessador da GPU, que pode executar um ou mais blocos simultaneamente

Cada multiprocessador executa 32 threads no modelo SIMT *Single Intruction, Multiple Thread*

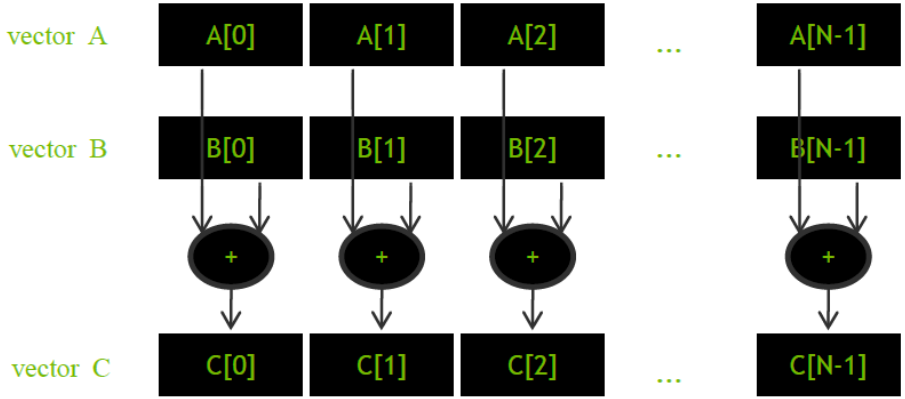


Exercício 1

Neste primeiro exercício iremos aprender como criar um kernel simples, que realiza a soma de 2 vetores.

Veremos as principais operações usadas em CUDA:

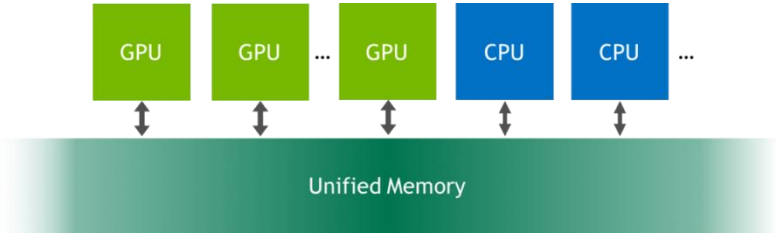
Alocação e liberação de memória, transferência de dados e lançamento do kernel



Alocação e Transferência de Memória

Opção 1) Utilizando a memória unificada

Forma mais fácil. Mas na maioria das vezes não é a mais eficiente.



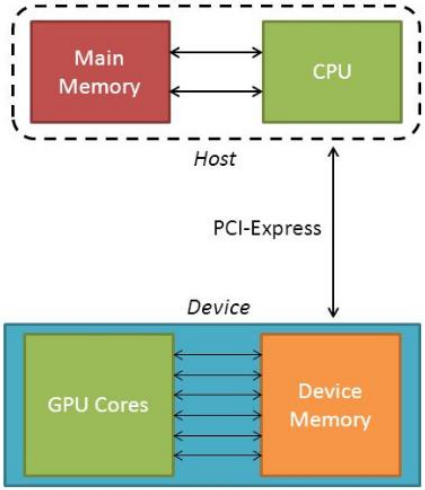

```
1. #include <cuda_runtime.h>
2.
3. int *A;
4. cudaMallocManaged(&A, 1024 * sizeof(int));
5.
6. double *B;
7. cudaMallocManaged(&B, 4096 * sizeof(double));

8. // A partir de agora usa na CPU e na GPU sem preocupação.
```

Alocação e Transferência de Memória

Opção 2) Transferindo os dados manualmente

Forma mais “complexa”. Mas se o programador for inteligente, pode ser mais eficiente.

```
1. #include <cuda_runtime.h>
2.
3. int *host_A, *device_A;
4.
5. host_A = (int *) malloc(1024 * sizeof(int));
6. cudaMalloc(&device_A, 1024 * sizeof(int));
7.
8. cudaMemcpy(device_A, host_A, 1024 * sizeof(int),
    cudaMemcpyHostToDevice);
9. // Após copiar do Host para o Device, utiliza no
   kernel e ao final, caso necessário, copia para o
   host
10.cudaMemcpy(host_A, device_A, 1024 * sizeof(int),
    cudaMemcpyDeviceToHost);
```



Declaração e Lançamento do Kernel

__global__ indica que a função é um Kernel CUDA

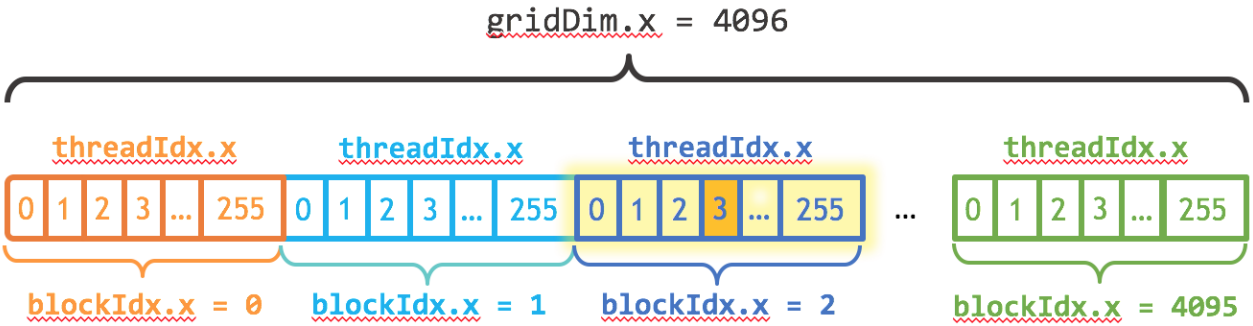
Na chamada do kernel o número de blocos e o *threads* por bloco deve ser passado

```
1. __global__ void vecAdd(int *A, int *B, int *C, int N){
2.     ...
3. }
4.
5. int main(int argc, char **argv){
6.     ...
7.     int blockSize = 32;
8.     int numBlocks = (N + blockSize - 1) / blockSize;
9.     printf("%d blocks of %d threads\n", numBlocks, blockSize);
10.
11.    vecAdd<<<numBlocks, blockSize>>>(A, B, C, N);
12.    cudaDeviceSynchronize(); // Barreira para CPU não avançar
13.    ...
14.}
```

Hierarquia de Threads em CUDA

É possível definir em 1D, 2D ou 3D.

Variáveis disponíveis dentro do kernel: **threadIdx.x, threadIdx.y, threadIdx.z, blockDim.x, blockDim.y, blockDim.z, blockIdx.x, blockIdx.y, blockIdx.z, gridDim.x, gridDim.y, gridDim.z**




```
index = blockIdx.x  * blockDim.x + threadIdx.x

index =    (2)      *   (256)    +     (3)      = 515
```

Desalocação de Memória

Não menos importante...

```
1. #include <cuda_runtime.h>
2.
3. int *A, *B, *C;
4. cudaMallocManaged(&A, N * sizeof(int));
5. cudaMallocManaged(&B, N * sizeof(int));
6. cudaMallocManaged(&C, N * sizeof(int));
7.
8. ...
9.
10.cudaFree(A);
11.cudaFree(B);
12.cudaFree(C);
```

Acesso aos servidores com GPU

1. Crie o arquivo chave.key
2. Copie o conteúdo de: https://pastebin.com/LSbKMLfh
3. chmod 600 chave.key
4. ssh -i chave.key workshop@gppd-hpc.inf.ufrgs.br
5. cp -r exercícios/ nome-sobrenome/
6. cd nome-sobrenome/
7. Trabalhe apenas no seu diretório!

Exercício 1

Crie um kernel, que realiza a soma de 2 vetores.

Solução - Exercício 1

```
1.  __global__ void vecAdd(int *A, int *B, int *C, int N){
2.      int i;
3.      for(i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x)
4.          C[i] = A[i] + B[i];
5.  }
6.
7.  int main(int argc, char **argv){
8.      ...
9.      cudaMallocManaged(&A, N * sizeof(int));
10.     cudaMallocManaged(&B, N * sizeof(int));
11.     cudaMallocManaged(&C, N * sizeof(int));
12.     ...
13.
14.     int blockSize = 256;
15.     int numBlocks = (N + blockSize - 1) / blockSize;
16.
17.     vecAdd<<<numBlocks,blockSize>>>(A, B, C, N);
18.     cudaDeviceSynchronize();
19.     ...
20.     cudaFree(A);
21.     cudaFree(B);
22.     cudaFree(C);
23. }
```

Exercício 2

Crie um kernel, que realiza a multiplicação de 2 matrizes.

Para tanto, utilize blocos e threads 2D.

```
1. dim3 blockSize(32, 32);

2. dim3 numBlocks((N + blockSize.x - 1) / blockSize.x,
     (N + blockSize.y - 1) / blockSize.y);
```

Solução - Exercício 2

```
1. __global__ void matrixMult(float *A, float *B, float *C, int N){
2.     int i, j, k;
3.     float sum;
4.     for(i = blockIdx.y * blockDim.y + threadIdx.y; i < N;
   i += blockDim.y * gridDim.y)
5.         for(j = blockIdx.x * blockDim.x + threadIdx.x; j < N;
   j += blockDim.x * gridDim.x){
6.             sum = 0;
7.             for(k = 0; k < N; k++)
8.                 sum += A[i * N + k] * B[k * N + j];
9.             C[i * N + j] = sum;
10.        }
11.}
```

O que não vimos?

Shared memory, operações atômicas, sincronização entre blocos, multi-gpu, etc

Onde aprender mais?

https://developer.nvidia.com/cuda-education-training

Dúvidas?

msserpa@inf.ufrgs.br

Bônus - Exercício 2 - Otimizado

```
1.  #define TILE 32
2.  __global__ void matrixMult(float *A, float *B, float *C, int N){
3.      __shared__ float As[TILE][TILE];
4.      __shared__ float Bs[TILE][TILE];
5.      int ii, i, j, k;
6.      float sum;
7.      for(i = blockIdx.y * blockDim.y + threadIdx.y; i < N; i += blockDim.y * gridDim.y)
8.          for(j = blockIdx.x * blockDim.x + threadIdx.x; j < N; j += blockDim.x * gridDim.x){
9.              sum = 0;
10.             for(ii = 0; ii < (N - 1) / TILE + 1; ii++){
11.                 if(ii * TILE + threadIdx.x < N)
12.                     As[threadIdx.y][threadIdx.x] = A[i * N + ii * TILE + threadIdx.x];
13.                 else
14.                     As[threadIdx.y][threadIdx.x] = 0;
15.                 if(ii * TILE + threadIdx.y < N)
16.                     Bs[threadIdx.y][threadIdx.x] = B[(ii * TILE + threadIdx.y) * N + j];
17.                 else
18.                     Bs[threadIdx.y][threadIdx.x] = 0;
19.                 __syncthreads();
20.                 for(k = 0; k < TILE; k++)
21.                     sum += As[threadIdx.y][k] * Bs[k][threadIdx.x];
22.                 __syncthreads();
23.                 C[i * N + j] = sum;
24.             }
25.         }
26. }
```

GPU Teaching Kit

Accelerated Computing