Programação Paralela – OPRP001

Programação em GPU

Desenvolvido por Prof. Guilherme Koslovski e Prof. Maurício Pillon

Referências

- Material de suporte no Moodle sobre GPU e CUDA
- Cursos da ERAD
  - http://www2.sbc.org.br/erad/doku.php?id=start
- https://developer.nvidia.com/cuda-gpus
- Fonte das figuras: documentação sobre CUDA

Graphics processing unit (GPU)

- Popularizado pela Nvidia
  - GeForce 256
- ATI technologies
  - Visual processing unit (VPU)
  - Radeon 9700
- Operações sobre matrizes e vetores
  - Qual classificação de Flynn?

OpenCL (Open Computing Language)

- CPU, GPU, FPGA
- Apple, AMD, IBM, Qualcomm, Intel e Nvidia
- Portabilidade de código

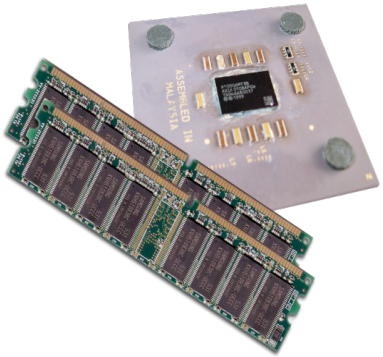
CUDA

- Compute Unified Device Architecture
- Plataforma + API para General Purpose GPU (GPGPU)
- GPU atua como dispositivo auxiliar

- E a memória?

- Kernel → funções compiladas para execução na GPU
- Uma thread executa N kernels

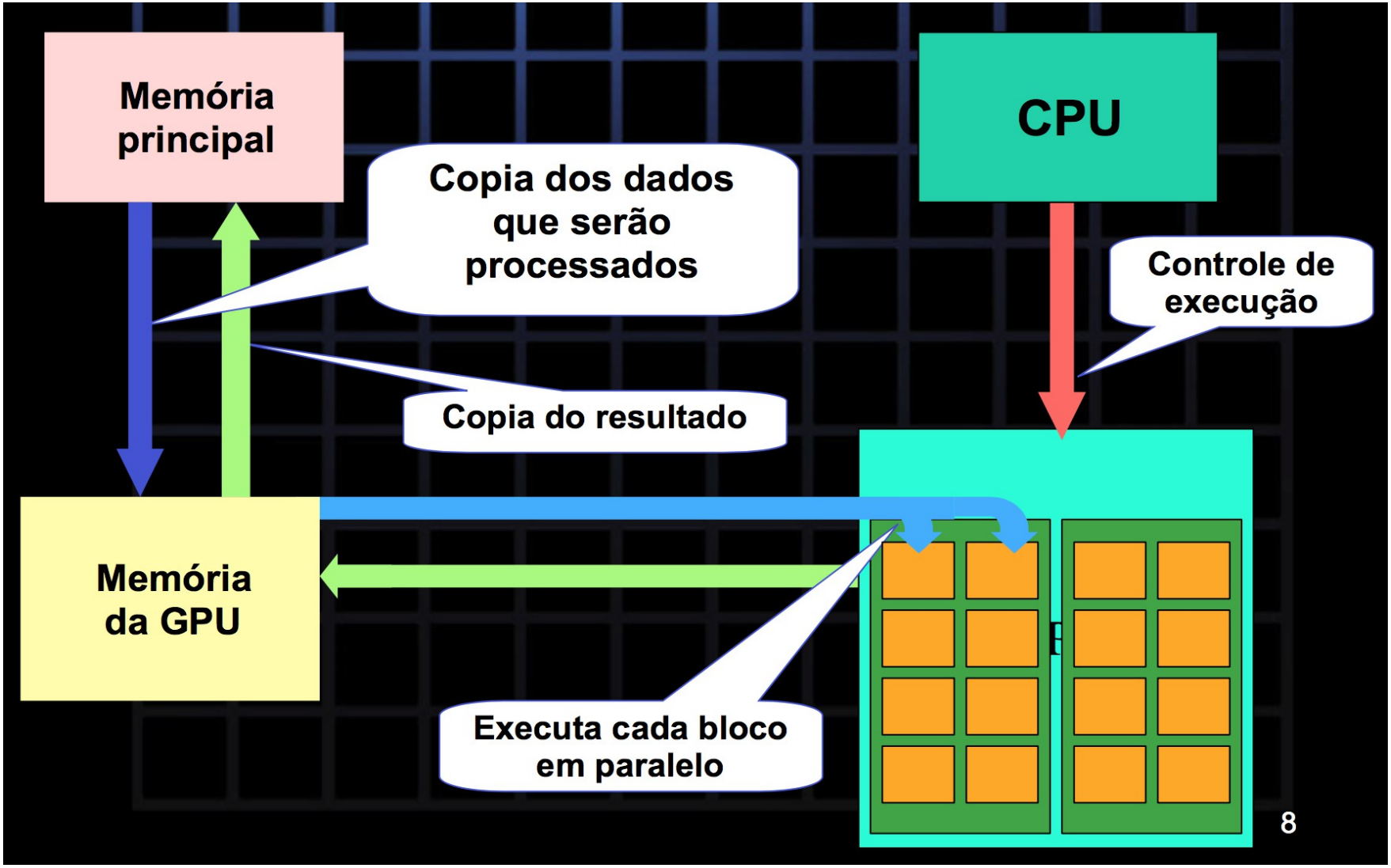
CUDA: terminologia



Host

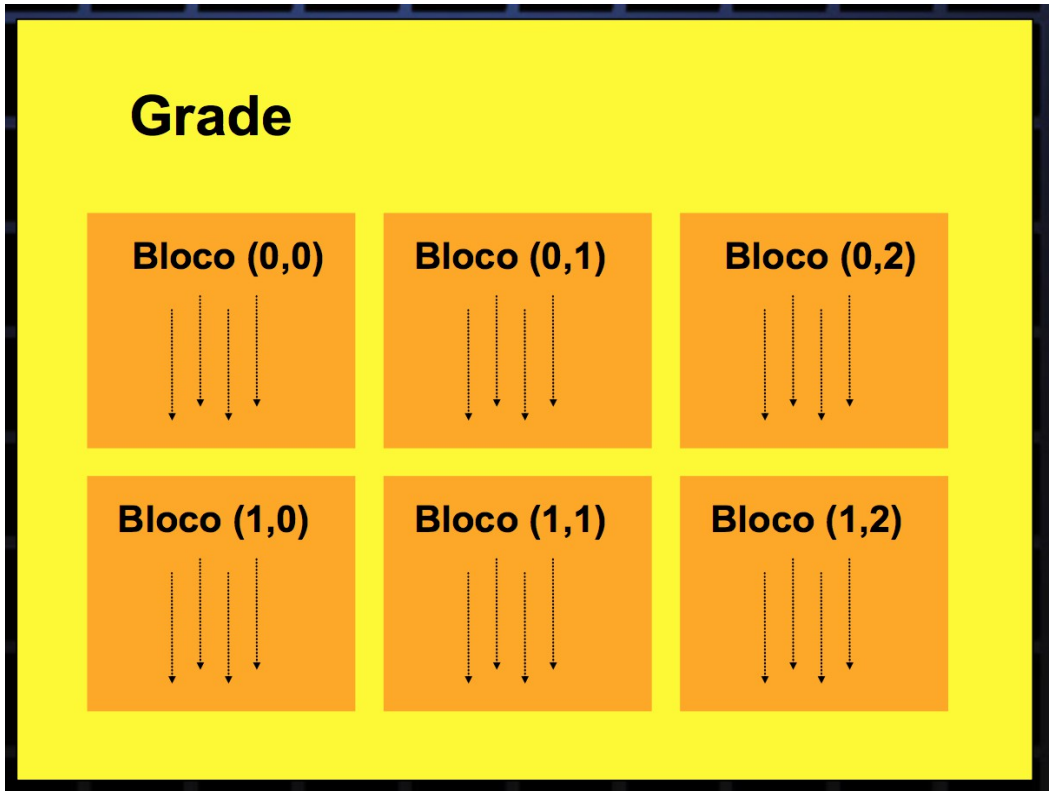
Device

CUDA - arquitetura

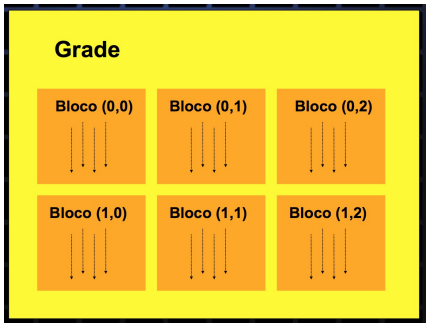


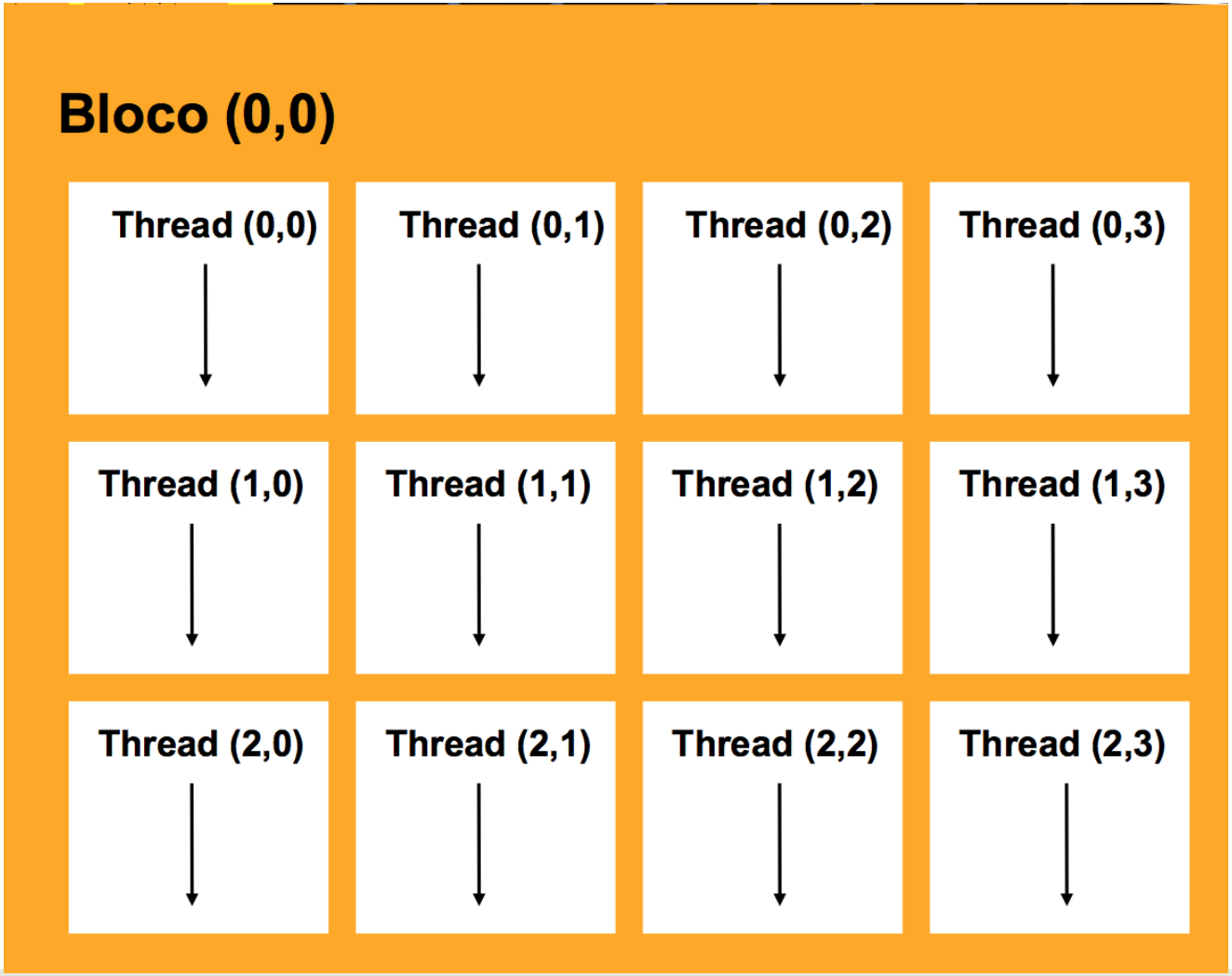
CUDA: grade/bloco/threads

- Grade: conjunto de blocos
- Bloco: conjunto de threads (executadas em paralelo)
- Thread: executa o kernel
- Cada bloco tem um identificador único
- Cada thread tem um identificador único no bloco



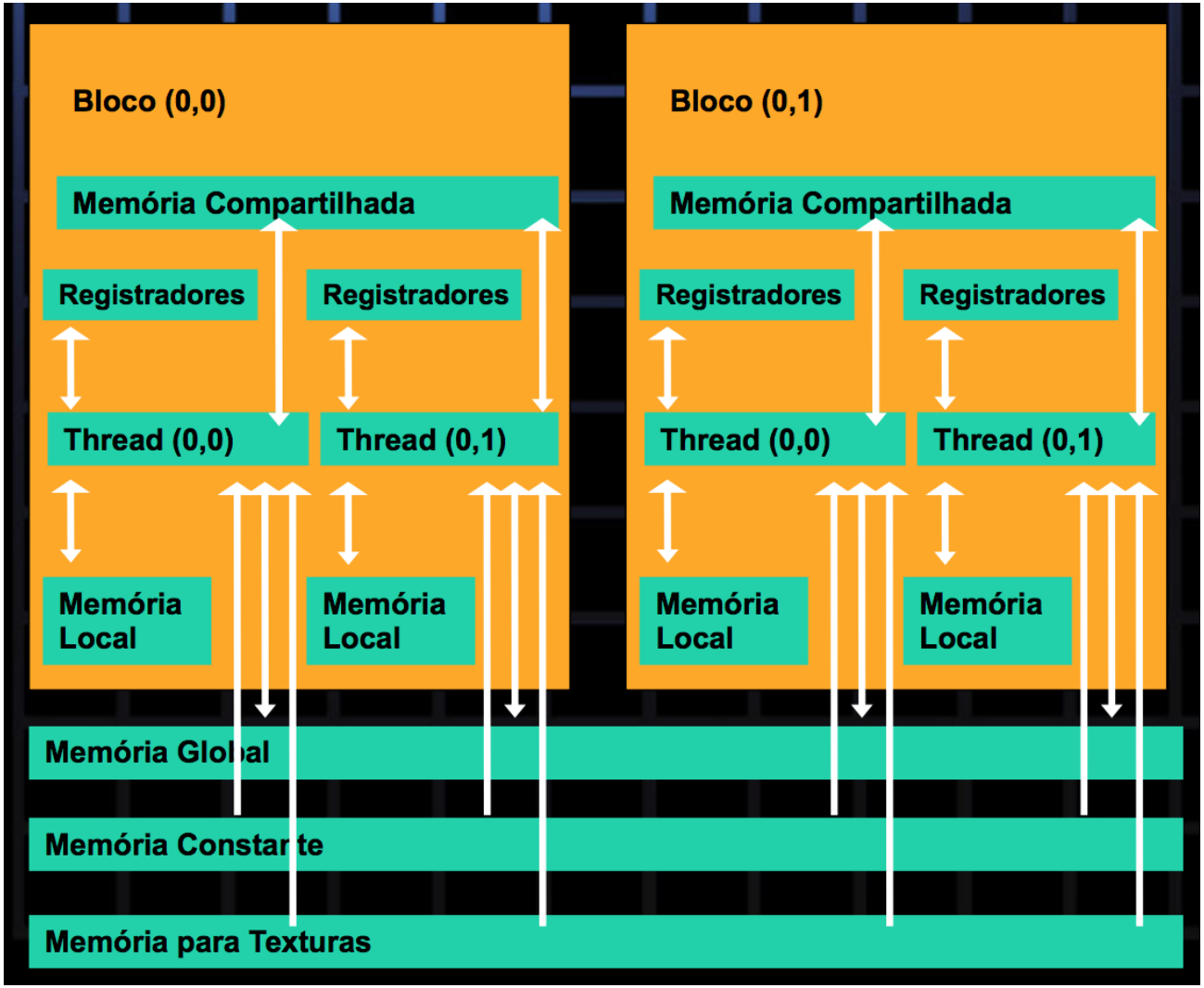
CUDA: grade/bloco/threads





CUDA: memória

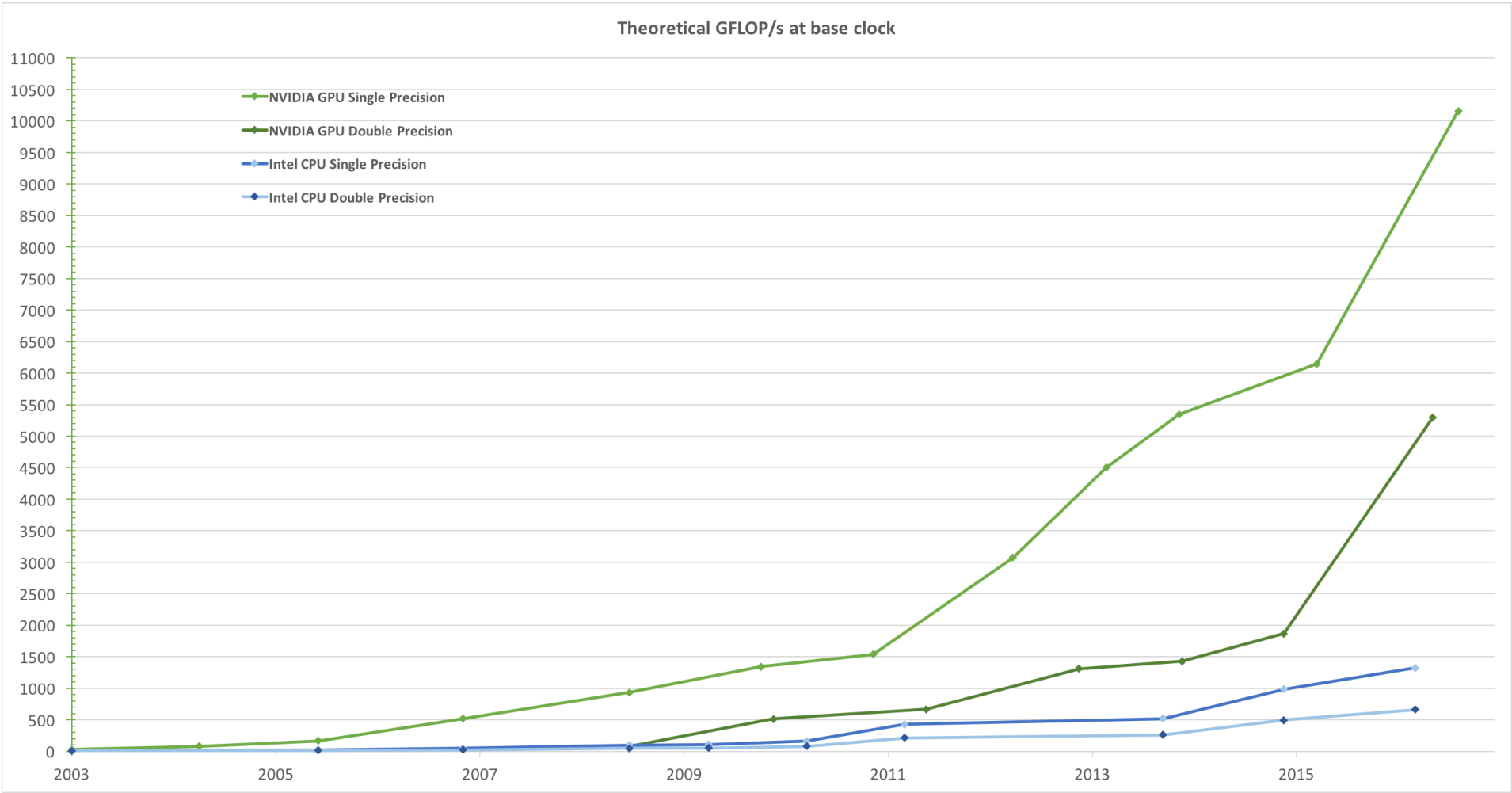
- Bloco
  - Memória compartilhada
- Grade
  - Memória global



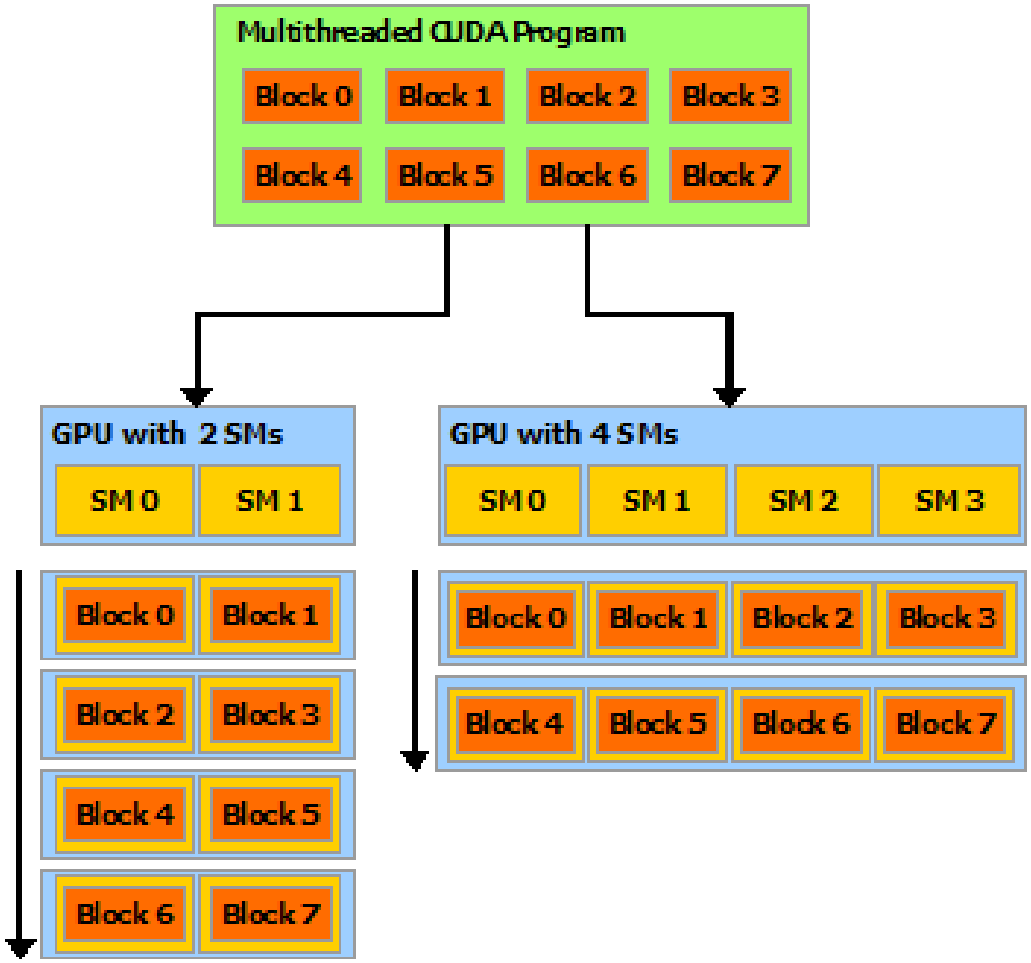
CUDA: hardware

- Diversas especificações!
- GPU é composta por N multiprocessadores (MP)
- MP é composto por M grupos de s*tream processors* (SP, cores)
- Cada core executa uma thread
- Os cores de um grupo (SP) executam a mesma instrução (SIMD).
- Warp → grupo de *threads* que efetivamente executam em paralelo no MP
  - 32 threads com IDs consecutivos
  - Keller: 1 MP → 16 SPs → 2 ciclos para executar a instrução → 32 instruções no pipeline
- Bloco é atribuído a um MP. A cache representa a memória compartilhada pelas threads.

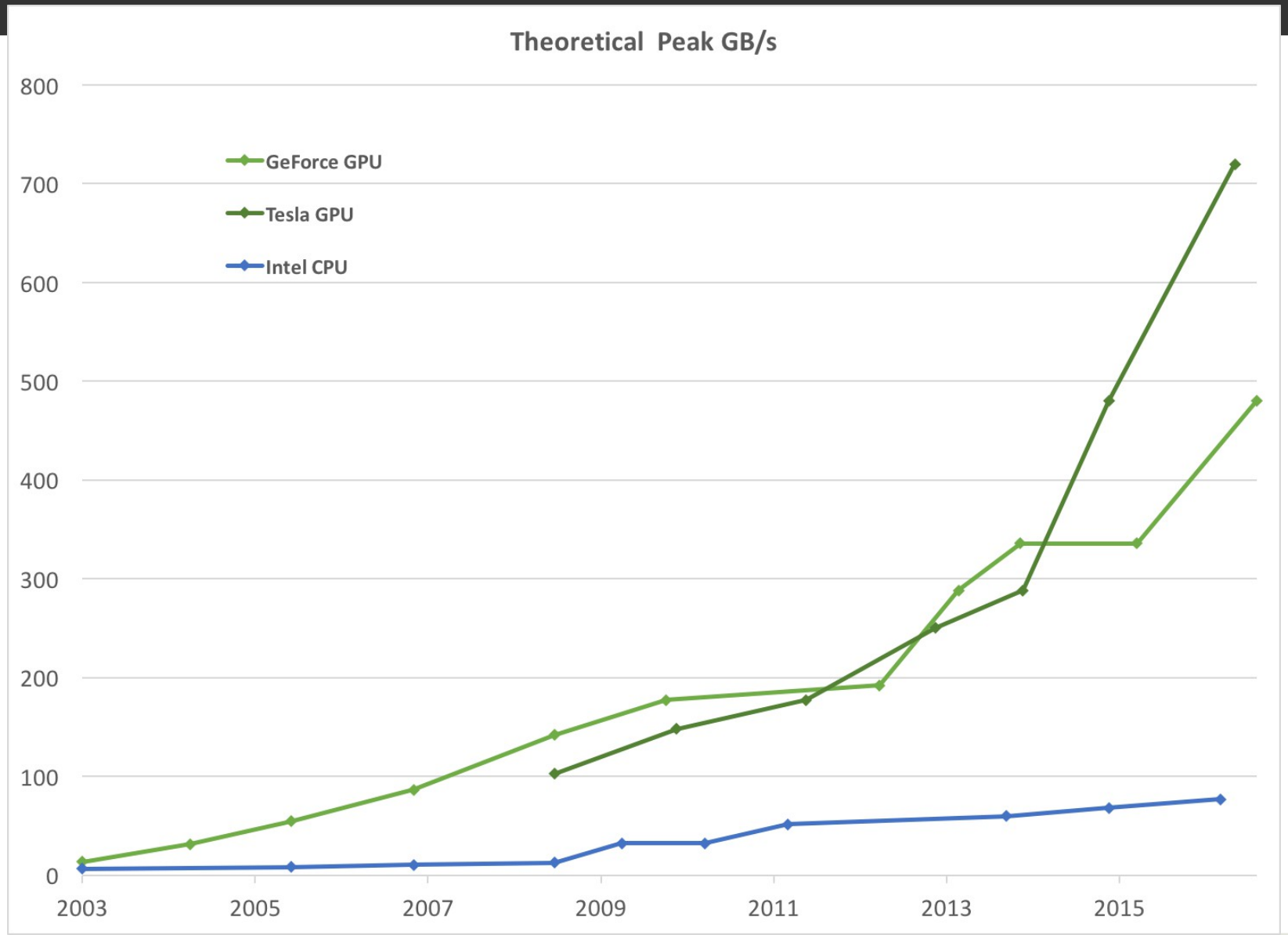
CUDA: hardware



CUDA: hardware



CUDA: hardware



CUDA: API

- Conjunto de extensões C e C++
  - Não suporta exceptions, STL headers

- CUDA kernels:
  - Não podem acessar memória do host;
  - Devem retornar void;
  - Número fixo e pré-definido de parâmetros;
  - Sem chamadas recursivas (preferencialmente);
  - Sem variáveis estáticas

CUDA: API

- **Qualificadores de funções**

  __device__: iniciadas por instruções na GPU. Executadas na GPU.

  __global__: iniciadas por instruções na CPU ou GPU. Executadas na GPU.

  __host__: iniciadas pela CPU. Executadas na CPU.

  __host__ e __device__ podem ser combinados

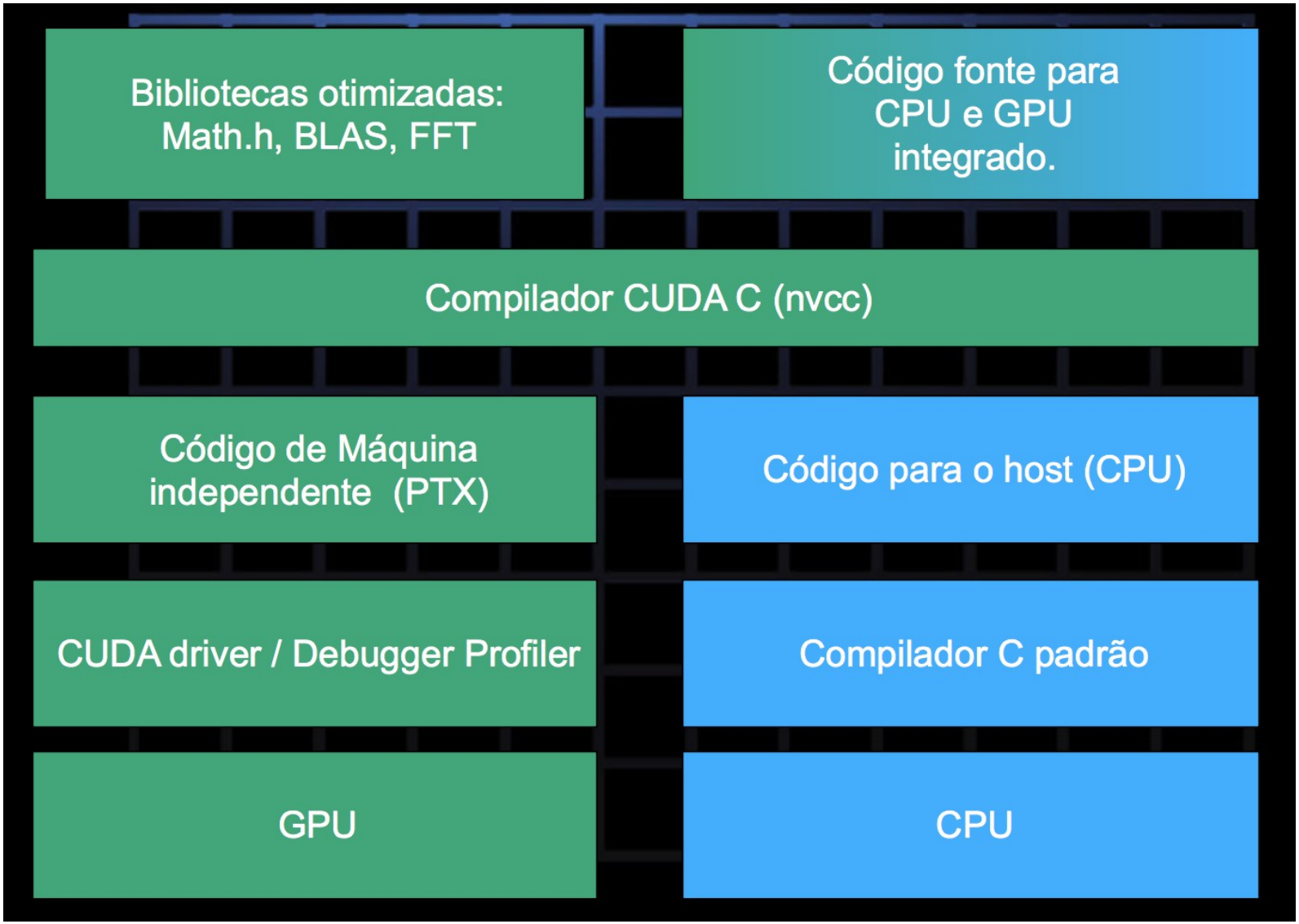
- **Qualificadores de variáveis**

  __device__: reside na GPU

  __constant__: reside na GPU em memória constante. Acessível por todas as threads.

  __shared__: reside na GPU em memória compartilhada. Acessível somente pelas threads do bloco.

CUDA: ambiente



CUDA: Exemplos iniciais

- **Largura de banda e informações sobre a placa**
- **Hello world :)**
- **Compilador nvcc**

CUDA: Variáveis built-in

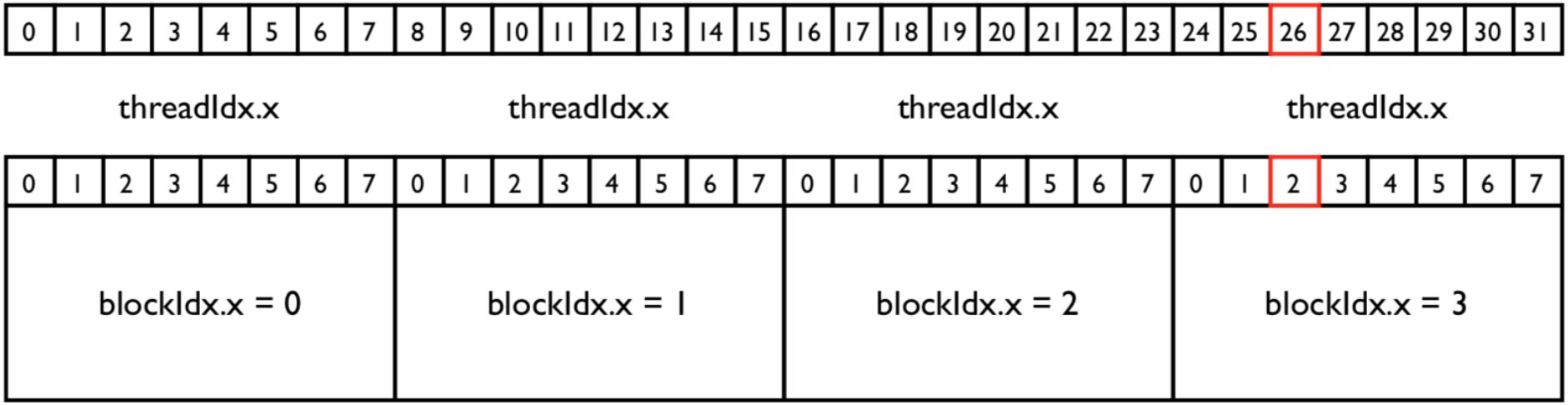
- **dim3: vetor de 3 inteiros**
- **gridDim: tipo dim3 contendo as dimensões do grid**
- **blockIdx: índice do bloco no grid**
- **blockDim: tipo dim3 contendo as dimensões do bloco**
- **threadIdx: identificador da thread no bloco**
- **warpSize: tamanho do warp da arquitetura**

CUDA: Soma de vetores

- Vamos tentar
- 1 bloco com N threads

```
#define N 256
#include <stdio.h>

__global__ void vecAdd (int *a, int *b, int *c);

int main() {
 int a[N], b[N], c[N];
 int *dev_a, *dev_b, *dev_c;

 // initialize a and b with real values (NOT SHOWN)

 size = N * sizeof(int);

 cudaMalloc((void**)&dev_a, size);
 cudaMalloc((void**)&dev_b, size);
 cudaMalloc((void**)&dev_c, size);

 cudaMemcpy(dev_a, a, size,cudaMemcpyHostToDevice);

 cudaMemcpy(dev_b, b, size,cudaMemcpyHostToDevice);

 vecAdd<<<1,N>>>(dev_a,dev_b,dev_c);

 cudaMemcpy(c, dev_c, size,cudaMemcpyDeviceToHost);

 cudaFree(dev_a);
 cudaFree(dev_b);
 cudaFree(dev_c);

 exit (0);
}

__global__ void vecAdd (int *a, int *b, int *c) {
 int i = threadIdx.x;
 c[i] = a[i] + b[i];
}
```

CUDA: Identificando uma thread



- **Global thread id 26**
  - **blockDim.x = 8**
  - **26 = blockIdx.x * blockDim.x + threadIdx.x**

CUDA: Soma de vetores novamente

- Ceil?
- Número máximo de threads por bloco = 1024
- E se a divisão não for exata?

```
#define N 1618
#define T 1024 // max threads per block
#include <stdio.h>

__global__ void vecAdd (int *a, int *b, int *c);

int main() {
 int a[N], b[N], c[N];
 int *dev_a, *dev_b, *dev_c;

 // initialize a and b with real values (NOT SHOWN)

 size = N * sizeof(int);

 cudaMalloc((void**)&dev_a, size);
 cudaMalloc((void**)&dev_b, size);
 cudaMalloc((void**)&dev_c, size);

 cudaMemcpy(dev_a, a, size,cudaMemcpyHostToDevice);

 cudaMemcpy(dev_b, b, size,cudaMemcpyHostToDevice);

 vecAdd<<<(int)ceil(N/T),T>>>(dev_a,dev_b,dev_c);

 cudaMemcpy(c, dev_c, size,cudaMemcpyDeviceToHost);

 cudaFree(dev_a);
 cudaFree(dev_b);
 cudaFree(dev_c);

 exit (0);
}

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

CUDA: multiplicação de matrizes

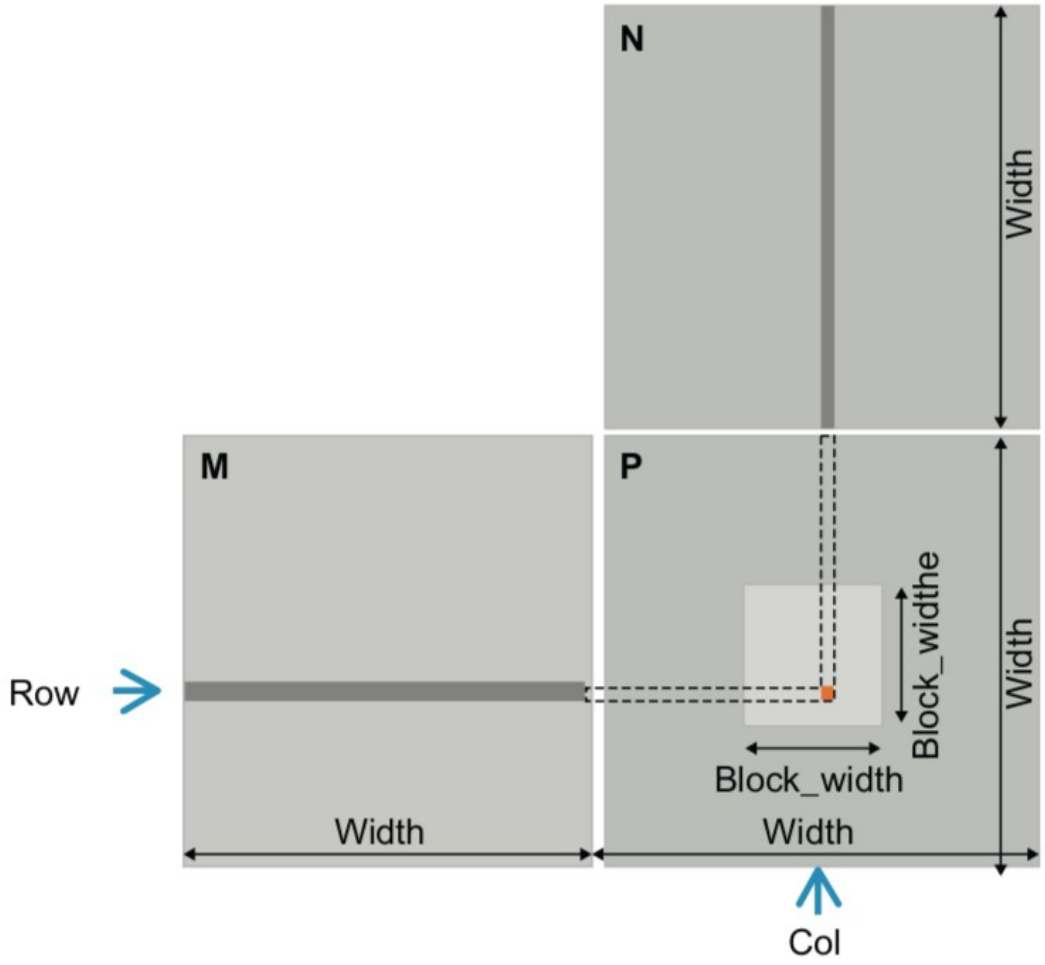
- **Multiplicação de matrizes**

- **Vetores: dim3 → x, y, z**
  - **y e z foram omitidos no exemplo anterior**
- **Matrizes: z será omitido**

  int linha = blockIdx.y * blockDim.y + threadIdx.y

  int coluna = blockIdx.x * blockDim.x + threadIdx.x

CUDA: multiplicação de matrizes



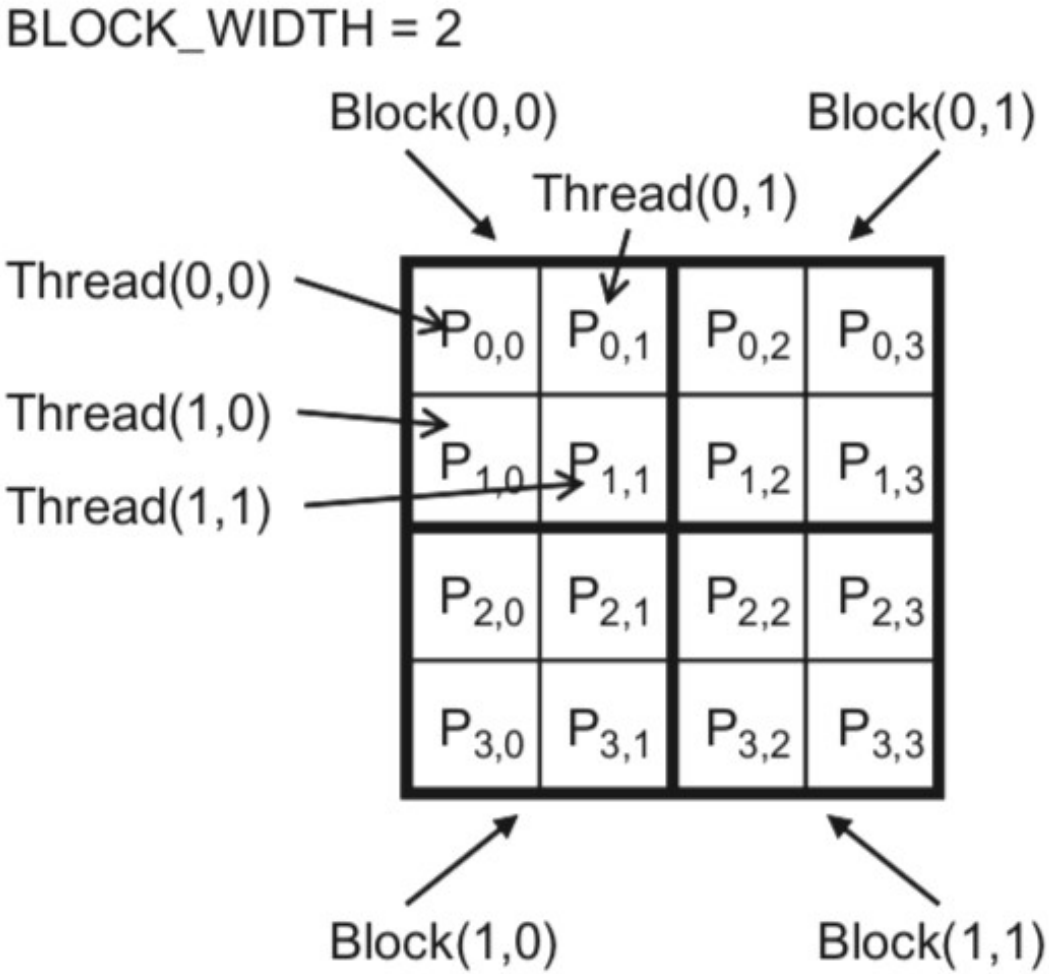
CUDA: multiplicação de matrizes

```
__global__ void MatrixMulKernel(float* M, float* N, float* P,
  int Width) {
  // Calculate the row index of the P element and M
  int Row = blockIdx.y*blockDim.y+threadIdx.y;
  // Calculate the column index of P and N
  int Col = blockIdx.x*blockDim.x+threadIdx.x;
  if ((Row < Width) && (Col < Width)) {
    float Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for (int k = 0; k < Width; ++k) {
      Pvalue += M[Row*Width+k]*N[k*Width+Col];
    }
    P[Row*Width+Col] = Pvalue;
  }

}
```

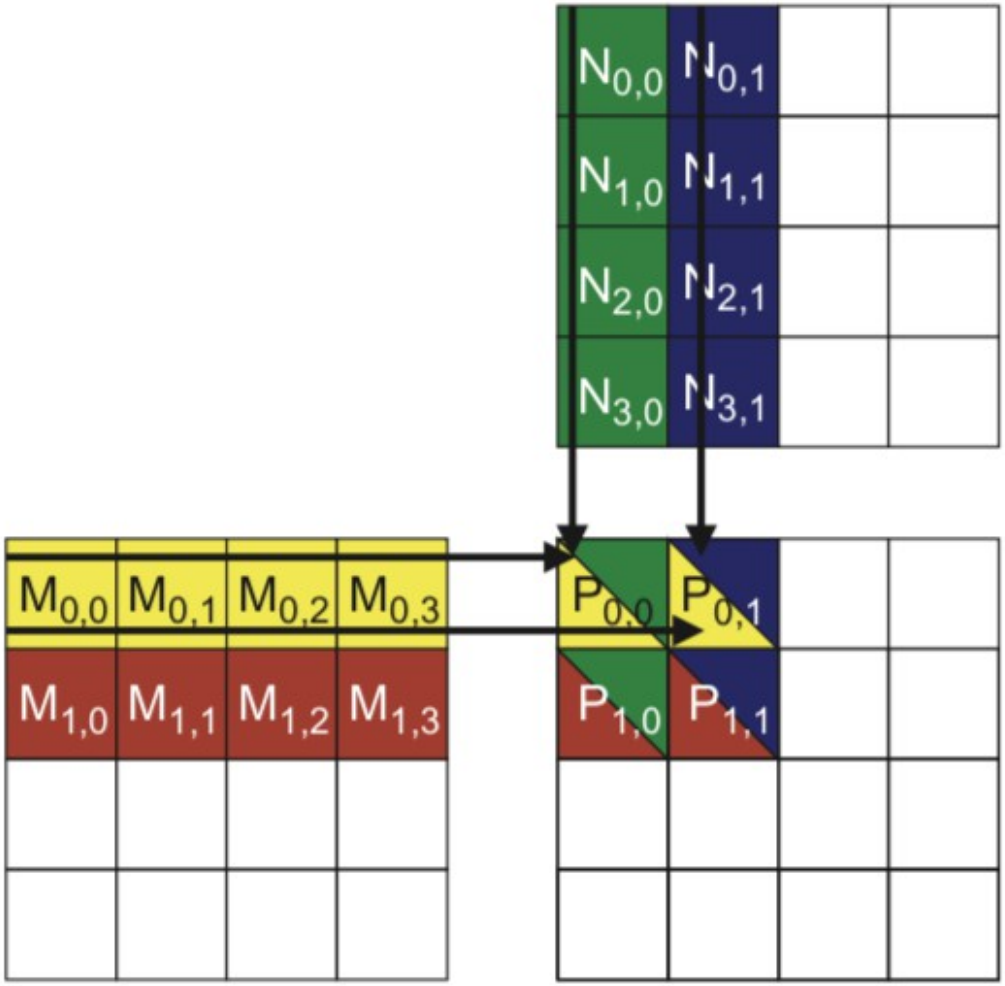
CUDA: multiplicação de matrizes

- **Qual o problema com a implementação atual?**
  - **Memória!**

CUDA: multiplicação de matrizes



CUDA: multiplicação de matrizes



CUDA: multiplicação de matrizes

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P,
    int Width) {

1.  __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
2.  __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

3.  int bx = blockIdx.x;  int by = blockIdx.y;
4.  int tx = threadIdx.x; int ty = threadIdx.y;

    // Identify the row and column of the d_P element to work on
5.  int Row = by * TILE_WIDTH + ty;
6.  int Col = bx * TILE_WIDTH + tx;

7.  float Pvalue = 0;
    // Loop over the d_M and d_N tiles required to compute d_P element
8.  for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {

      // Collaborative loading of d_M and d_N tiles into shared memory
9.    Mds[ty][tx] = d_M[Row*Width + ph*TILE_WIDTH + tx];
10.   Nds[ty][tx] = d_N[(ph*TILE_WIDTH + ty)*Width + Col];
11.   __syncthreads();

12.   for (int k = 0; k < TILE_WIDTH; ++k) {
13.     Pvalue += Mds[ty][k] * Nds[k][tx];
      }
14.   __syncthreads();
    }
15. d_P[Row*Width + Col] = Pvalue;
  }
```