Programação em GPU

Supercomputação

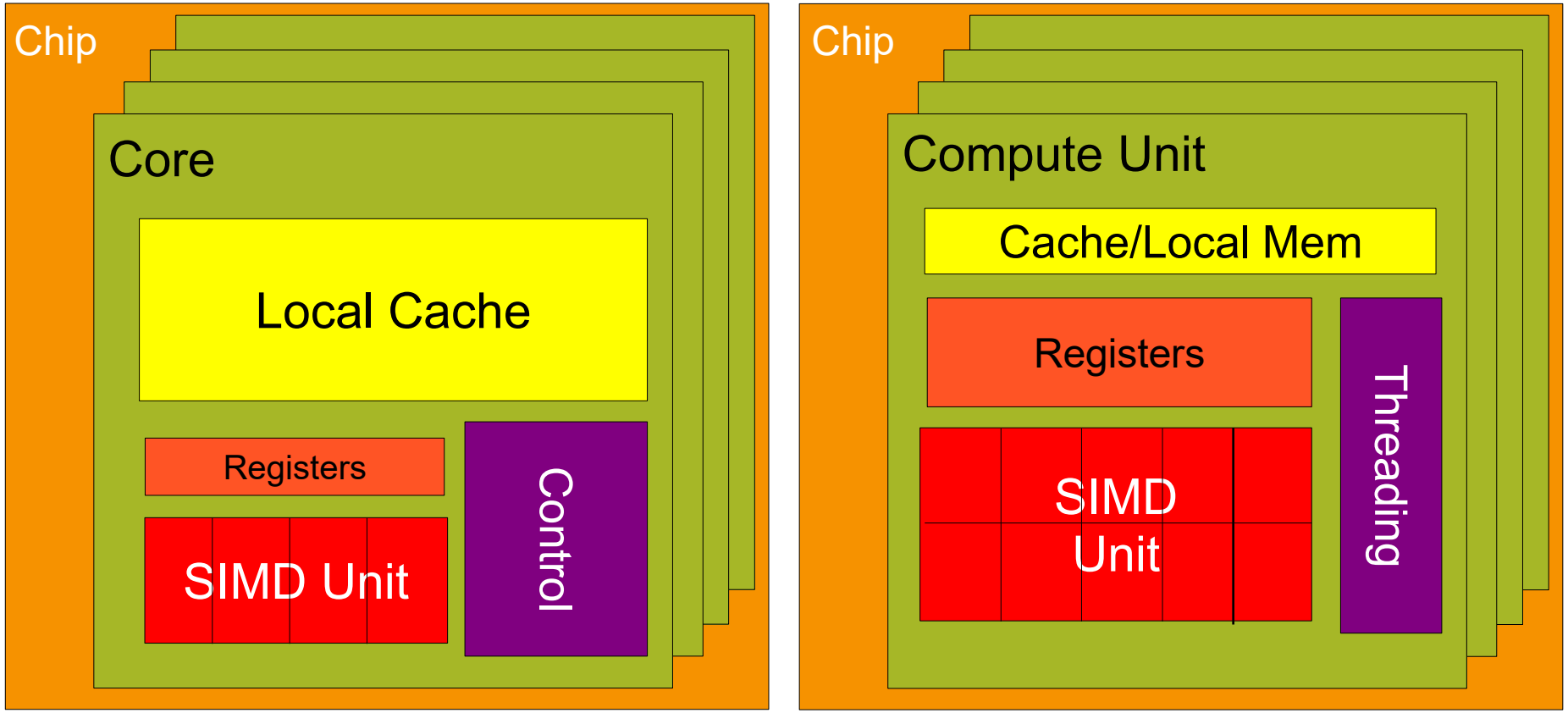
Objetivos

- Entender a diferença entre CPU e GPU
- Devemos unir os pontos fortes para tirar o melhor desempenho possível!

CPU e GPU são diferentes

CPU
Latency Oriented Cores

GPU
Throughput Oriented Cores



CPUs: Focada em latência

ULAs generalistas

Reduz a latência das operações

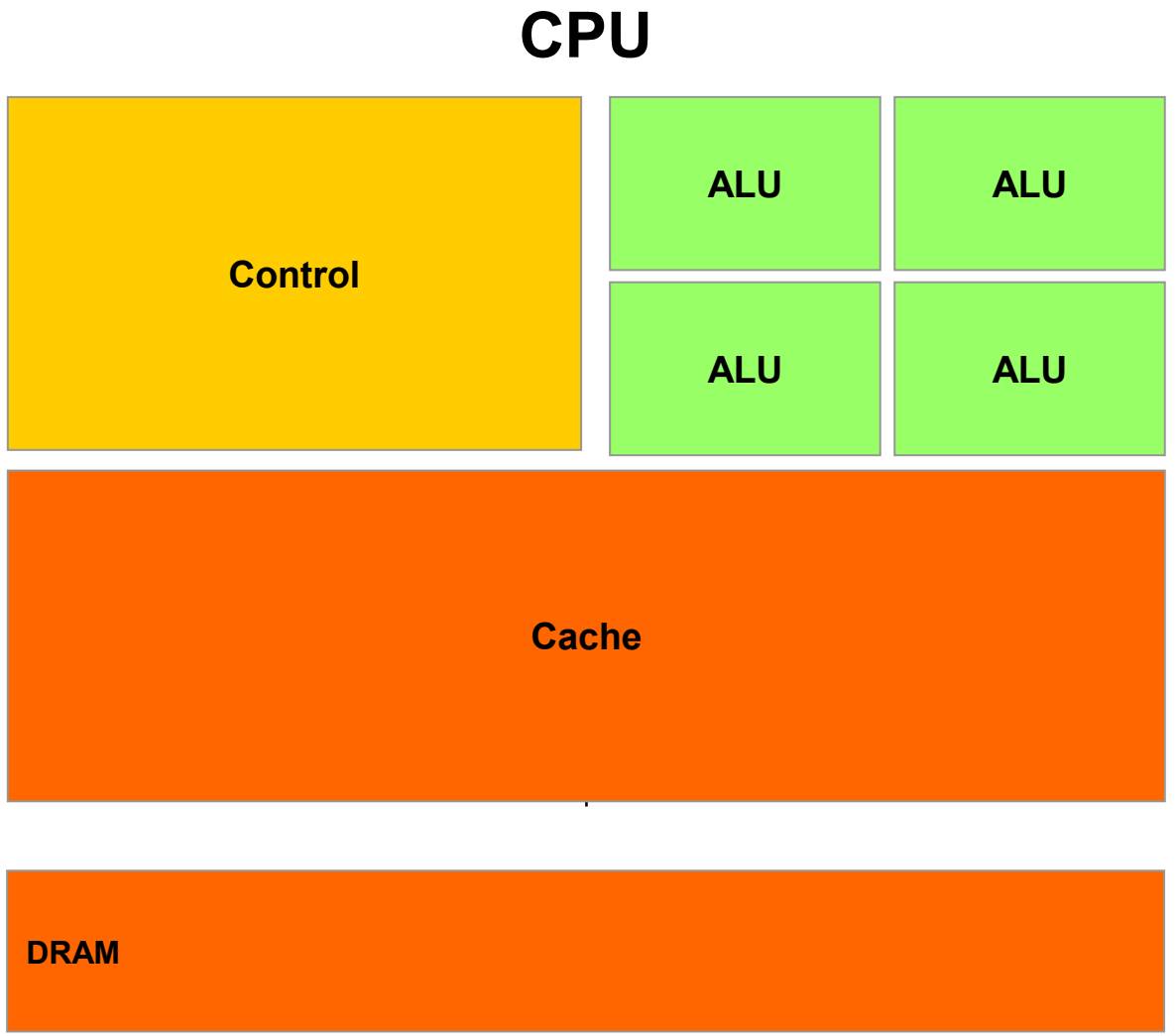
Caches grandes

Convertem acessos à memória de longa latência em acessos ao cache de curta latência

Controle sofisticado

Previsão de desvio para reduzir a latência de desvios

Encaminhamento de dados para reduzir a latência de dados



GPUs: Focada em Throughput

Caches pequenas

Para aumentar a taxa de transferência de memória

Controle simples

Sem previsão de desvio

Sem encaminhamento de dados

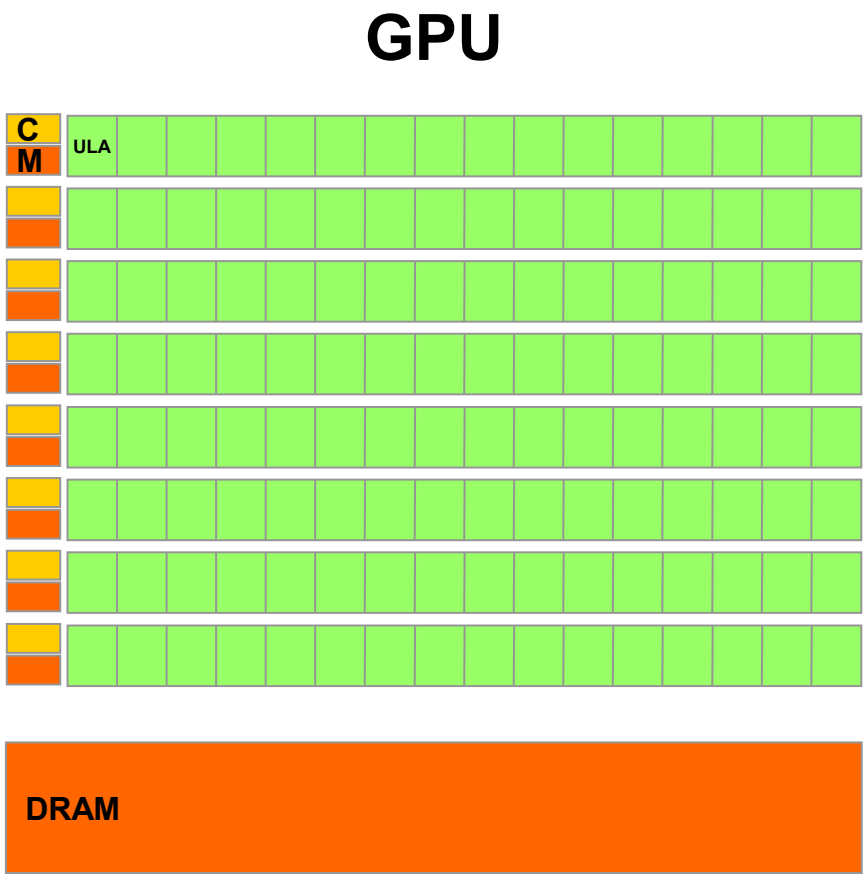
ULAs especialistas

Muitas unidades, com alta latência, mas fortemente pipelinezadas para alto throughput

Requer muitas threads para valer a pena

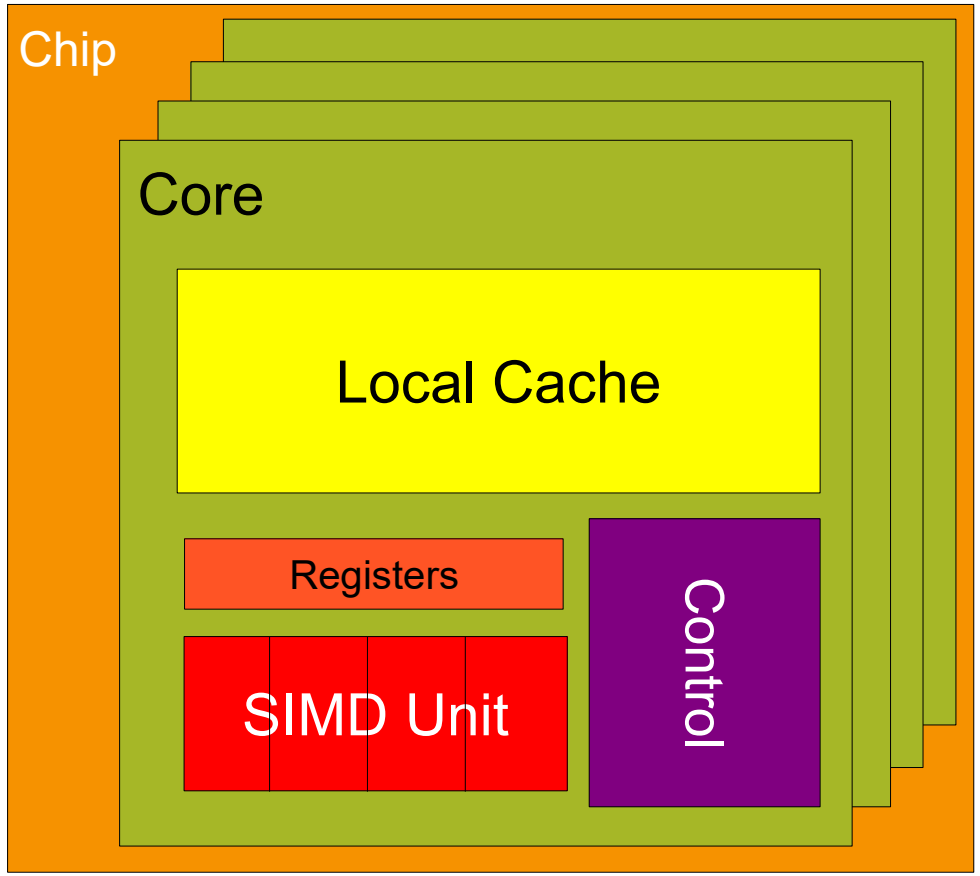
Lógica de threading

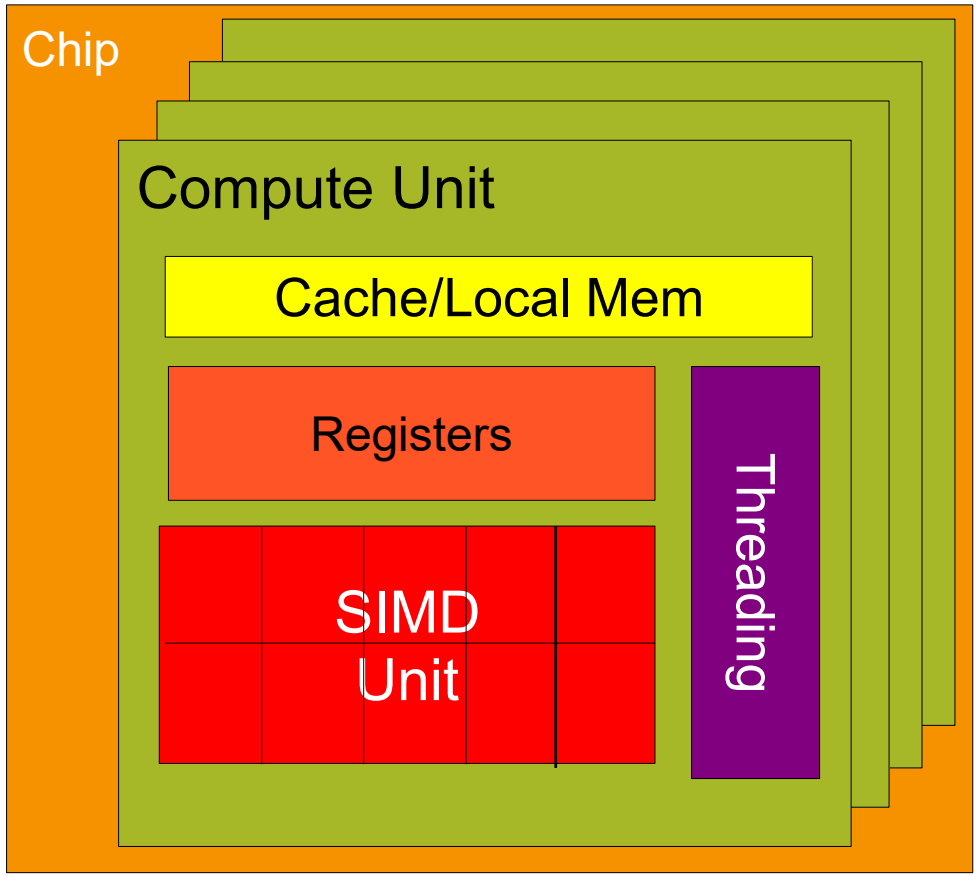
Estado das threads



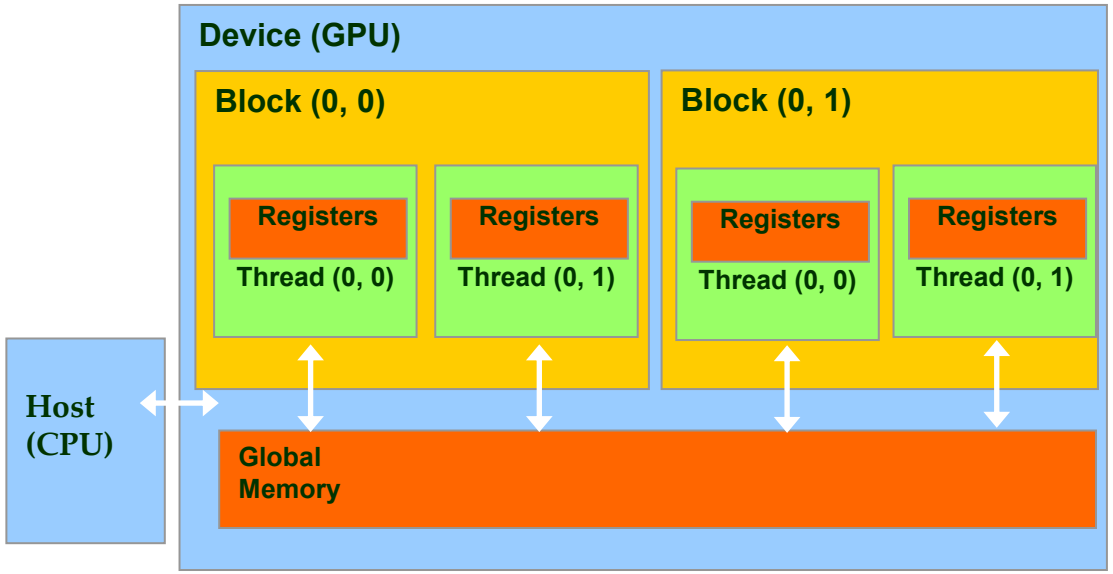
CPU
Latency Oriented Cores

GPU
Throughput Oriented Cores



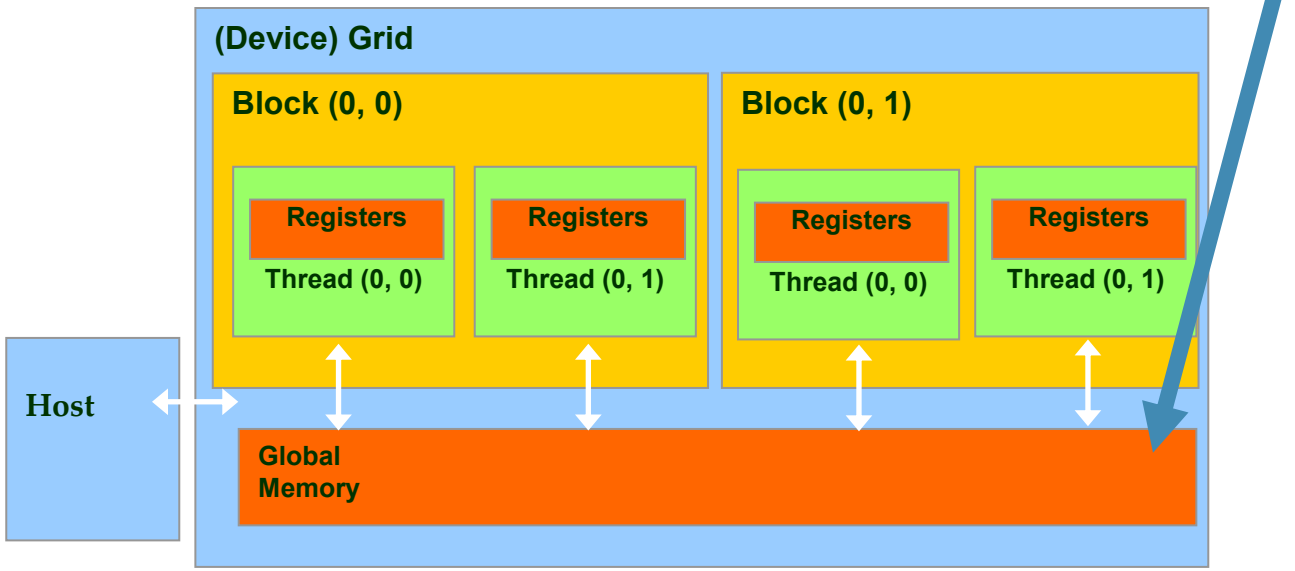


Gerenciamento de memória Host ←→Device



- Device:
  R/W memória local → **registradores**
  R/W memória compartilhada → **memória global**
- Host:
  Transfere dados ←→ **memória global**

CUDA para gerenciamento de memória



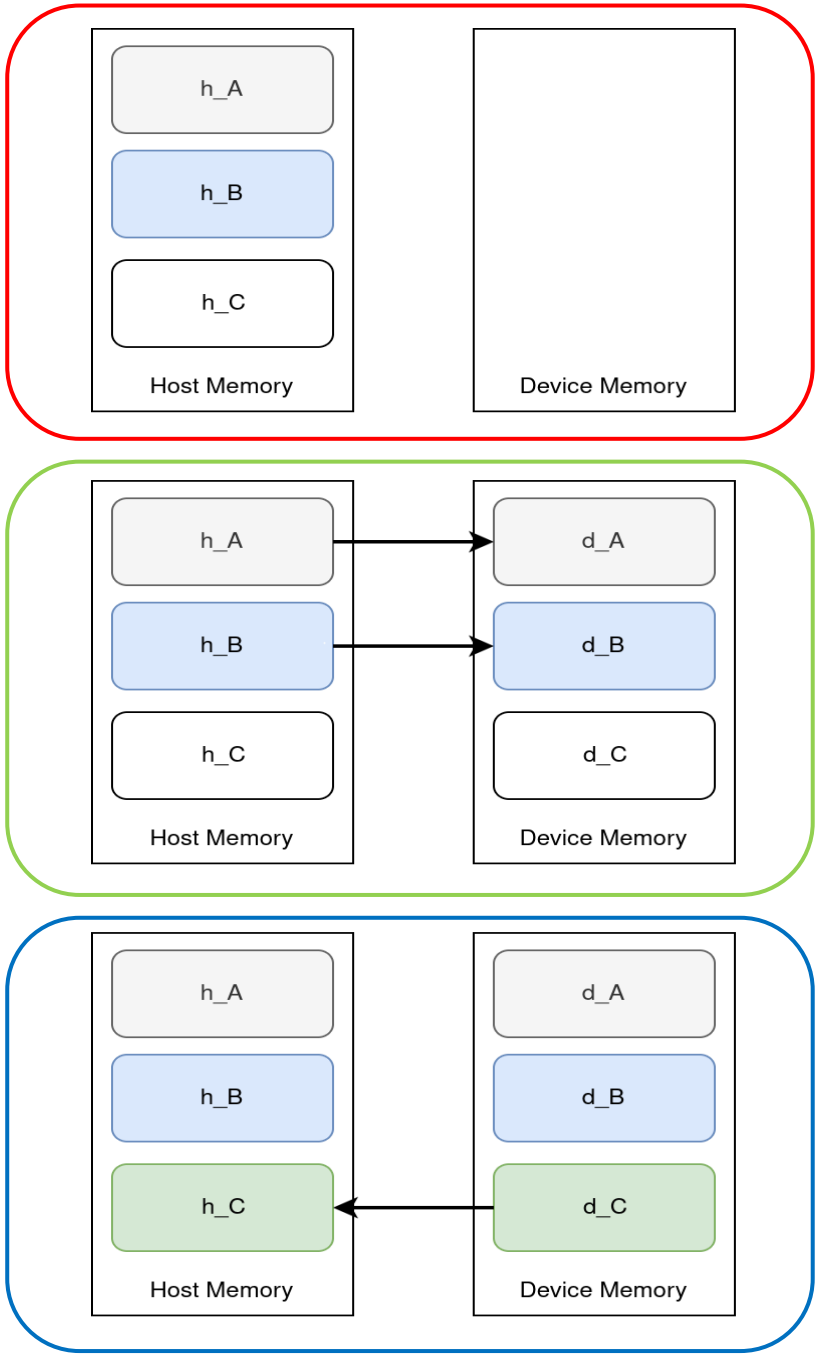
cudaMalloc()
Aloca um objeto na memória global do dispositivo
Dois parâmetros:
- Endereço de um ponteiro para o objeto alocado,
- Tamanho do objeto alocado em bytes

cudaFree()
Libera um objeto da memória global do dispositivo
Um parâmetro:
- Ponteiro para o objeto a ser liberado

cudaMemcpy()
Transfere dados entre memórias
Requer quatro parâmetros:
- Ponteiro de destino
- Ponteiro de origem
- Número de bytes a serem copiados
- Tipo/direção da transferência

> A transferência para o device é síncrona em relação ao host.

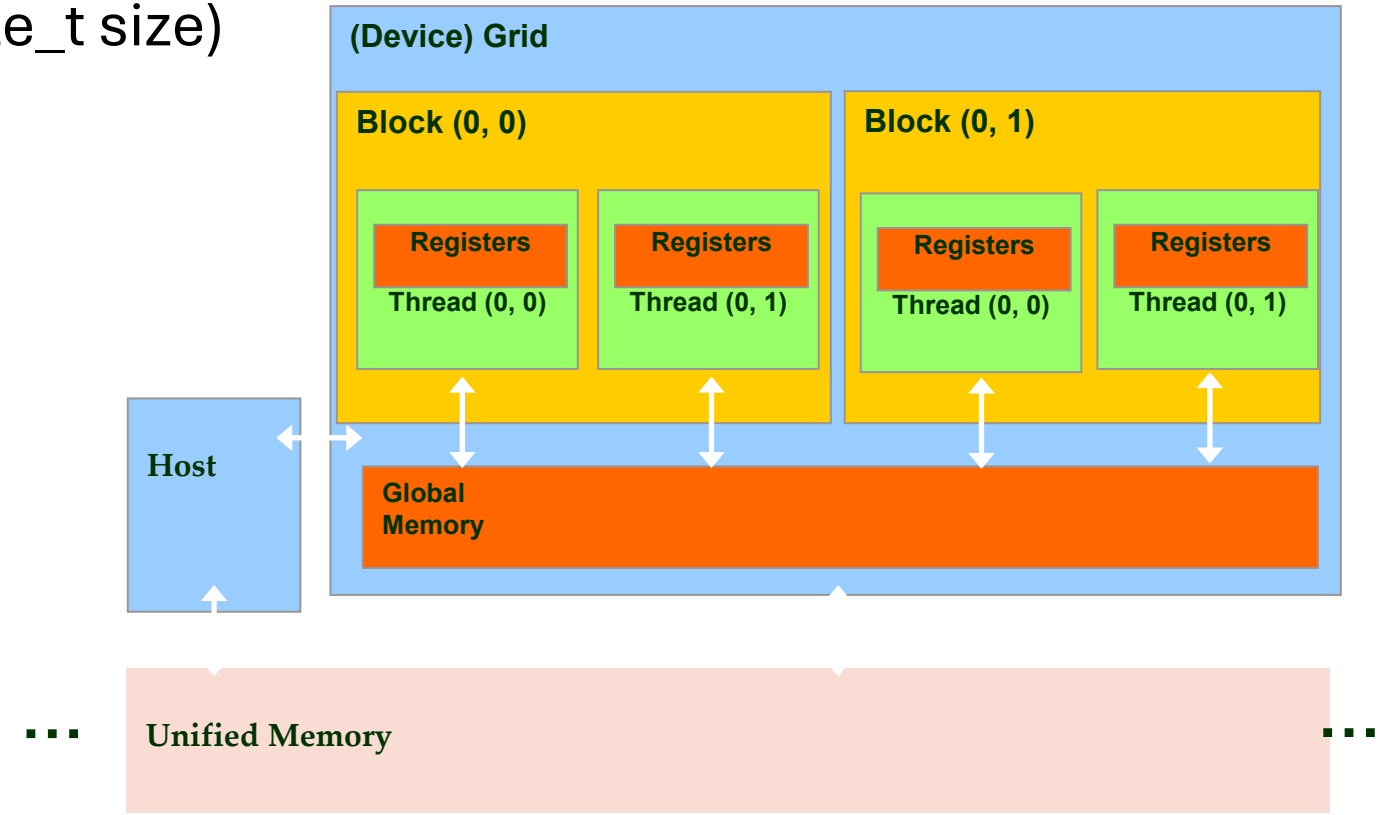
```
... Allocate h_A, h_B, h_C ...

void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
   int size = n * sizeof(float); float *d_A, *d_B, *d_C;

   cudaMalloc((void **) &d_A, size);
   cudaMalloc((void **) &d_B, size);
   cudaMalloc((void **) &d_C, size);

   cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
   cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Kernel code – veremos nos próximos Capitulos...

   cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
   cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}

... Free h_A, h_B, h_C ...
```



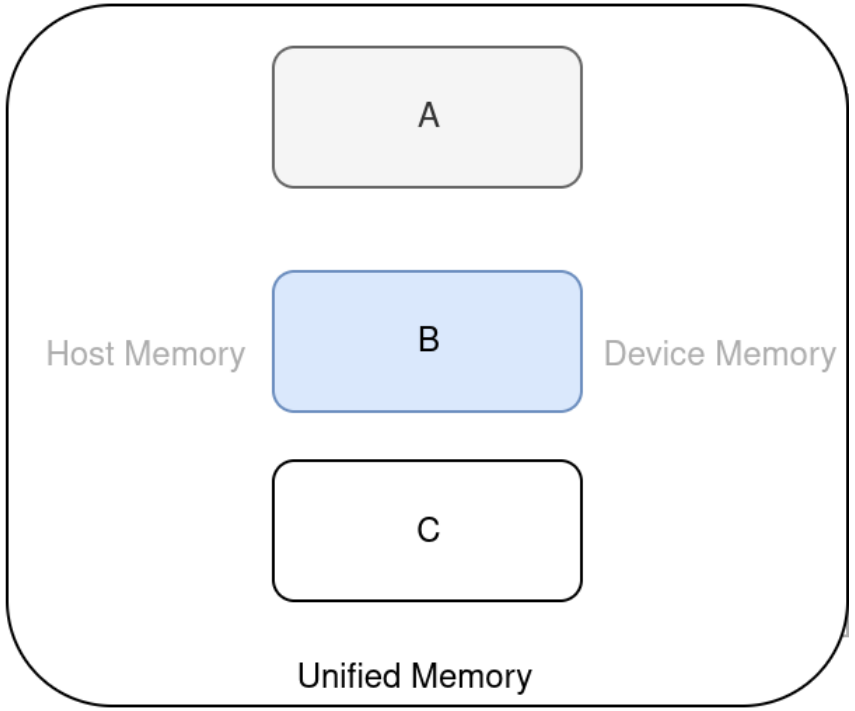
Unified Memory

cudaMallocManaged(void** ptr, size_t size)

- Mesmo espaço de memória CPUs/GPU
- Mantém uma única copia dos dados
- CUDA gerencia o dado
- Migração CPU ←→ GPU sob demanda
- Compativel com cudaMalloc(), cudaFree()
- cudaMemAdvise(), cudaMemPrefetchAsync(),
- cudaMemcpyAsync()



```
float *A, *B, *C
cudaMallocManaged(&A, n * sizeof(float));
cudaMallocManaged(&B, n * sizeof(float));
cudaMallocManaged(&C, n * sizeof(float));

// Initialize A, B

void vecAdd(float *A, float *B, float *C, int n)
{
 // Kernel code – veremos nos próximos Capitulos...
}

cudaFree(A);
cudaFree(B);
cudaFree(C);
```

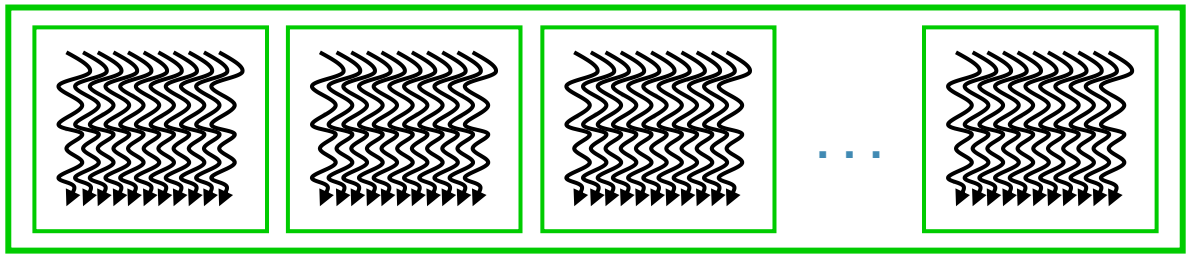


CUDA padrão

Serial Code (host)

Parallel Kernel (device)

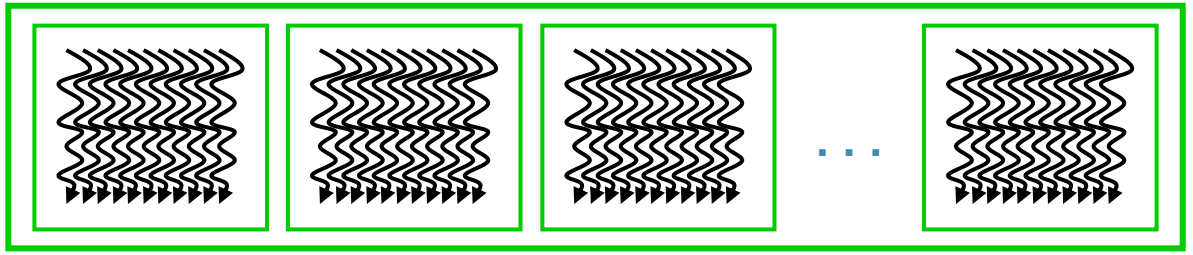
KernelA<<< nBlk, nTid >>>(args);



Serial Code (host)

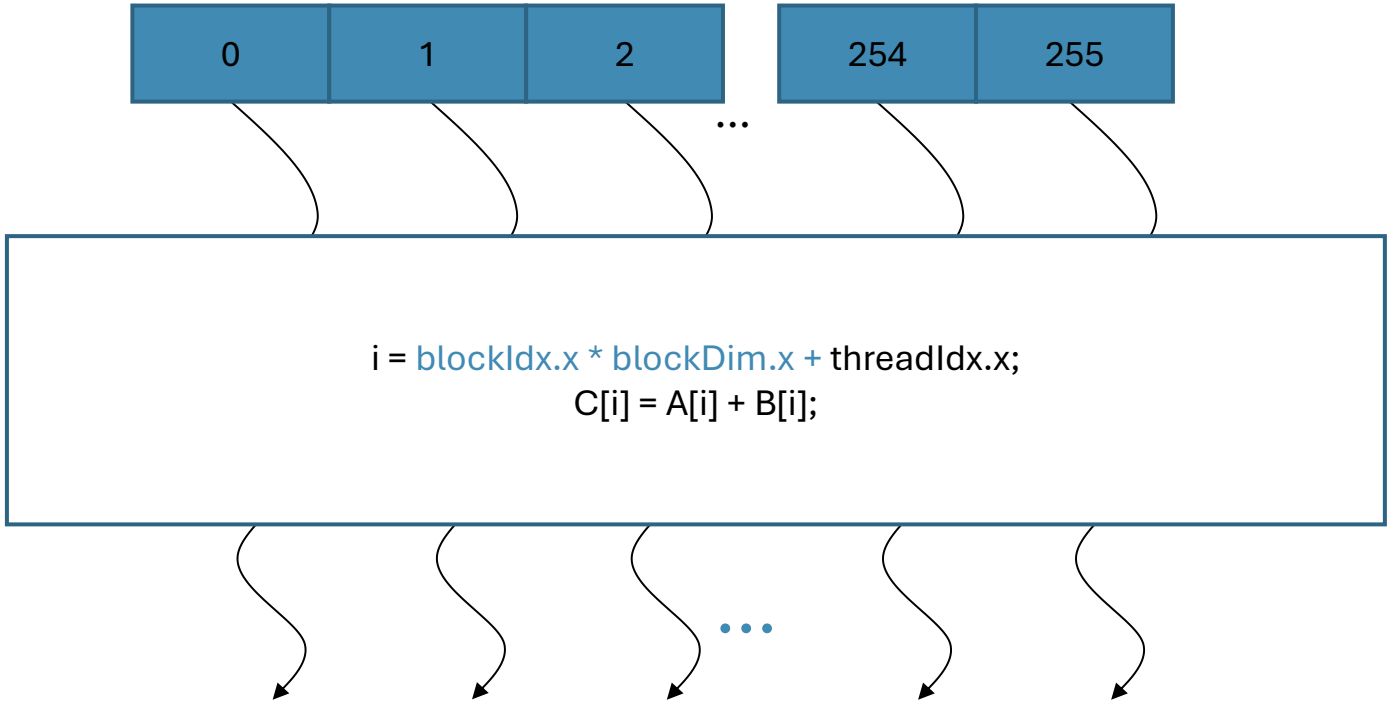
Parallel Kernel (device)

KernelB<<< nBlk, nTid >>>(args);

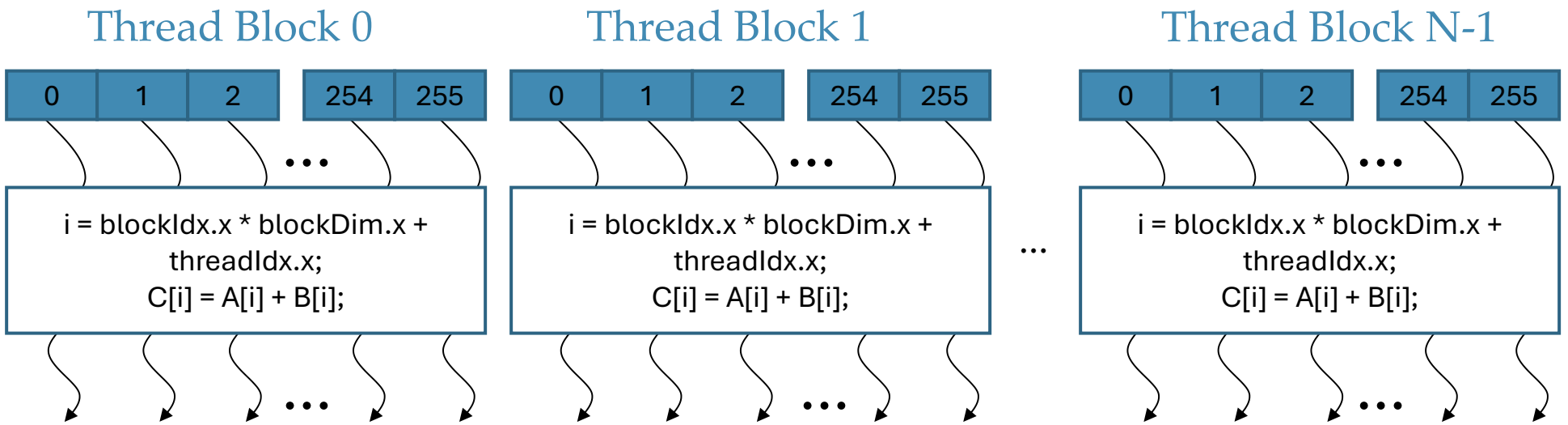


Arrays de Threads Paralelas

- Um kernel CUDA é executado por uma array de threads
  - Todas as threads em um bloco executam o mesmo kernel code (Single Program Multiple Data - SIMD)
  - Cada thread tem Id's únicos que possibilitam personalizações



Thread Blocks



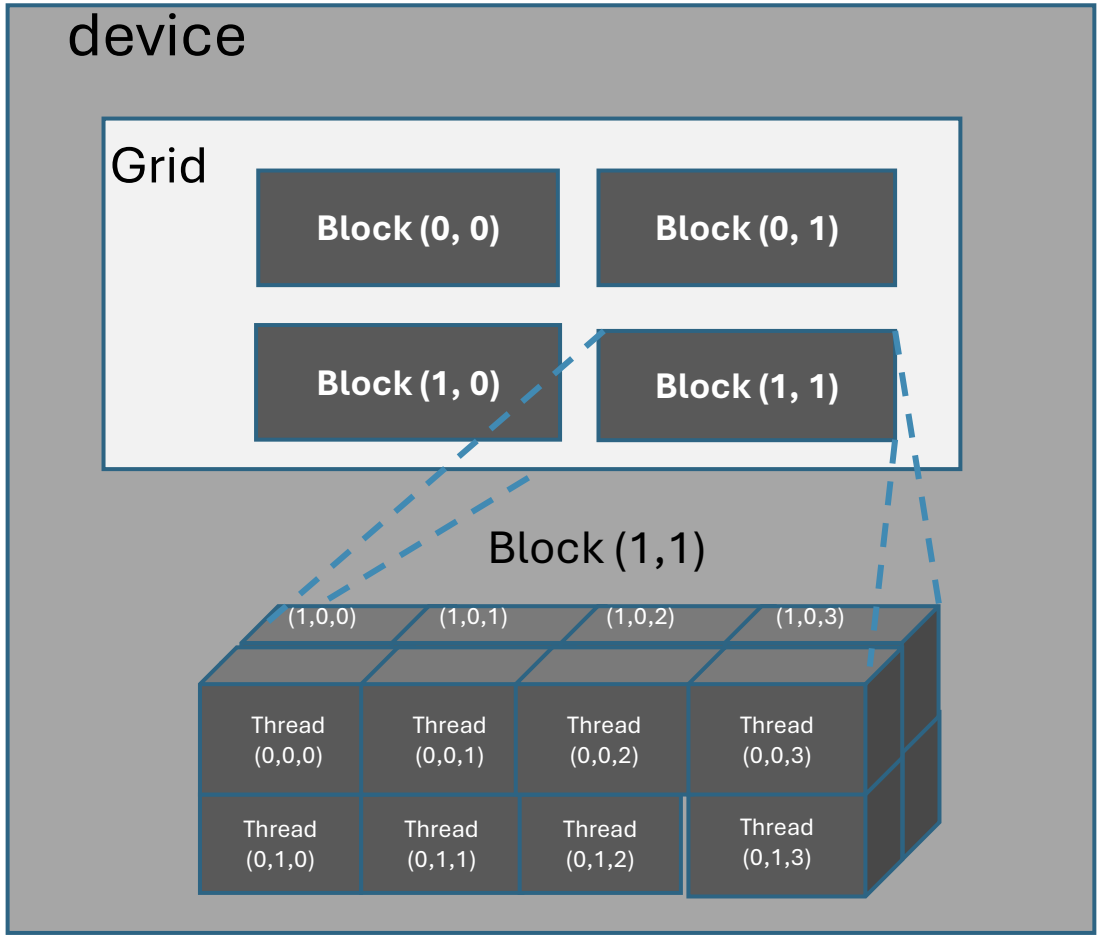
As Threads de um bloco se comunicam via: **shared memory, operações atomicas e barreiras**

Threads em blocos diferentes **NÃO** se enxergam

blockIdx e threadIdx

- Cada thread usa indices para definir qual interval de dado trabalhar
  - blockIdx: 1D, 2D, ou 3D
  - threadIdx: 1D, 2D, ou 3D

- Simplifica o controle de dados na Memória quando trabalha com dados multidimensionais
  - Image processing
  - ...



Hora do Hello World!

```
#include <cstdio>

__global__ void
mykernel(void) {
}

int main(void) {
  mykernel<<<1,1>>>();
  printf("Hello World!\n");
  return 0;
}
```