



# Formação Inteligência Artificial

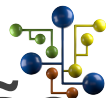


# Programação Paralela em GPU



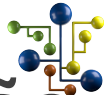
# Programação Paralela em CUDA

## Parte 3



# Programação Paralela em CUDA - Parte 3





# Programação Paralela em CUDA - Parte 3

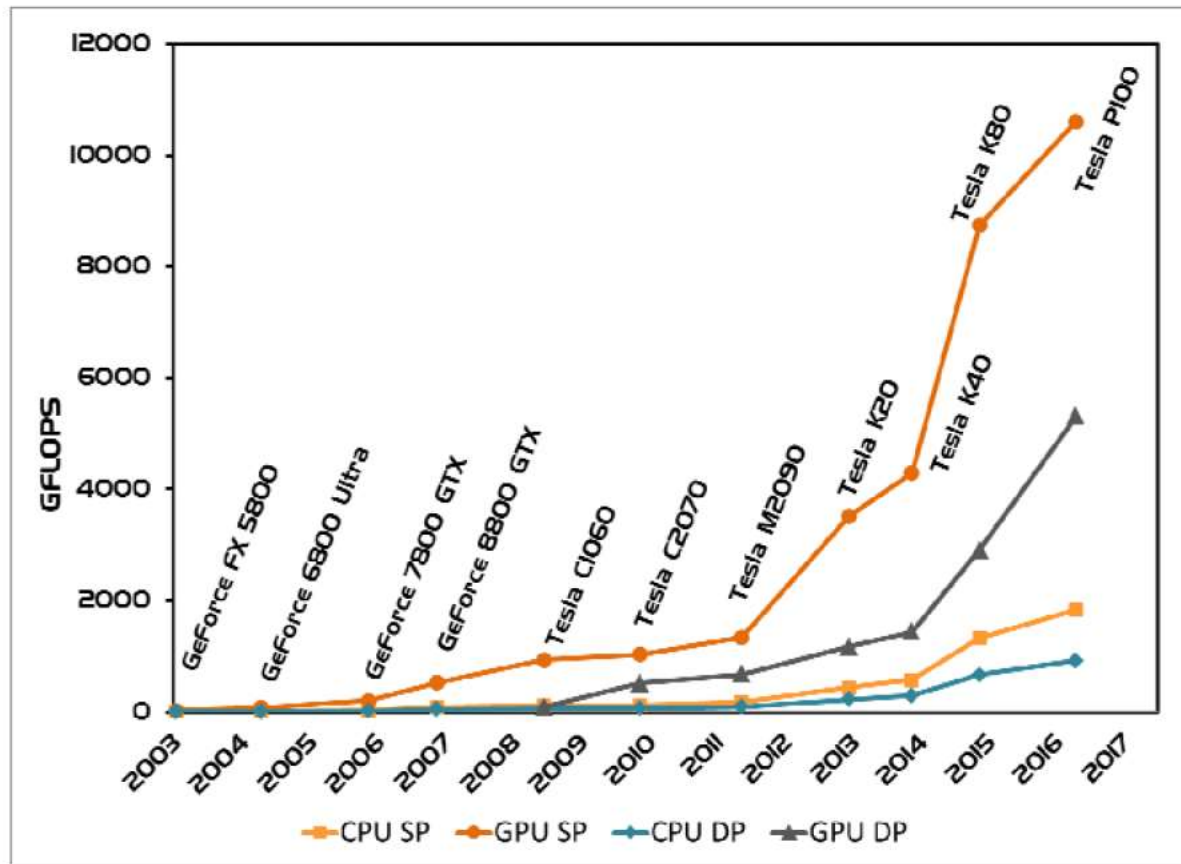
- Estrutura de Programação Paralela em GPU
- Revisão
- Padrões de Programação Paralela
- Gerenciamento de Memória da GPU
- Sincronismo de Threads



# Estrutura de Programação Paralela em GPU



# Estrutura de Programação Paralela em GPU



A evolução das GPUs é muito mais rápida que a evolução das CPUs












# Estrutura de Programação Paralela em GPU

## CPU x GPU

	Intel Xeon E5-2699v4 (Broadwell-EP) 	NVIDIA Tesla K80 (Kepler) 	NVIDIA Tesla M60 (Kepler) 	NVIDIA Tesla P100 (Pascal) 	NVIDIA Jetson TX2 (Pascal) 
Processing Cores	22	4992	4096	3584	8 ARM + 256 Pascal
Clock Frequency	2.2-3.6GHz	0.562-0.875GHz	0.900-1.180GHz	1.328-1.48GHz	0.854 – 1.465GHz
Memory Bandwidth	76.8 GB/s / socket	480GB/s	320GB/s	720GB/s	58.4GB/s
Peak Tflops (single)	1.83 @ 2.6GHz	8.74 @ 0.875GHz	9.68 @ 1.180GHz	10.6 @ 1.48GHz	0.75@1.465GHz
Peak Tflops (double)	0.915 @ 2.6GHz	2.91 @ 0.875GHz	0.30 @ 1.180GHz	5.3 @ 1.48GHz	0.023@1.465GHz
Gflops/Watt (single)	12.62	29.1	32.2	35.3	50
Total Memory	>>24GB	24GB	16GB	16GB	8GB







# Estrutura de Programação Paralela em GPU

## Compute Capability

Architecture	Compute Capability	GPUs	Example Features
Tesla	1.0	GeForce 8800, Tesla C870	Base Functionality
Fermi	2.0	GeForce GTX 480, Tesla C2050	Fast Double Precision, Memory Caches
Kepler	3.0	GeForce GTX 680, Tesla K10	Warp Shuffle Functions
	3.5	GeForce GTX Titan Black, Tesla K40	Dynamic Parallelism
	3.7	Tesla K80	More Registers / Shared Memory
Maxwell	5.0	GeForce GTX 750 Ti, Tegra X1	Power Efficient Architecture
	5.2	Tesla M40, Tesla M60	More Shared Memory
Pascal	6.0	Tesla P100	Half Precision (FP16)
	6.1	Titan Xp	Int8
	6.2	Tegra P1	Int8 + FP16





# Estrutura de Programação Paralela em GPU

Programação  
para a GPU

- CUDA C/C++
- CUDA Fortran
- Matlab
- Mathematica
- Python (NumbaPro, PyCUDA)





# Estrutura de Programação Paralela em GPU

Operações independentes são ótimas candidatas para programação paralela em GPU

```
a = 2 * x;  
b = 2 * y;  
c = 3 * x;
```

Operações independentes  
Perfeito para execução em paralelo

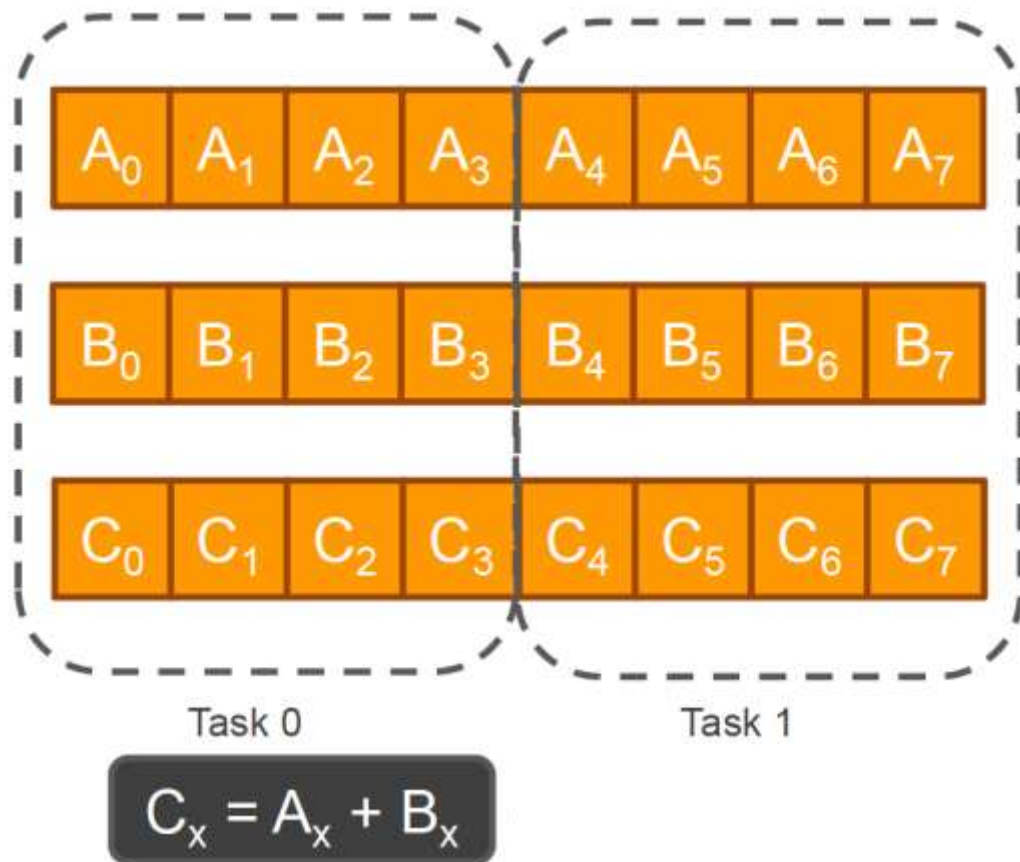
```
a = 2 * x;  
b = 2 * a * a;  
c = b * 9;
```

Operações dependentes  
Ruim para execução em paralelo





# Estrutura de Programação Paralela em GPU

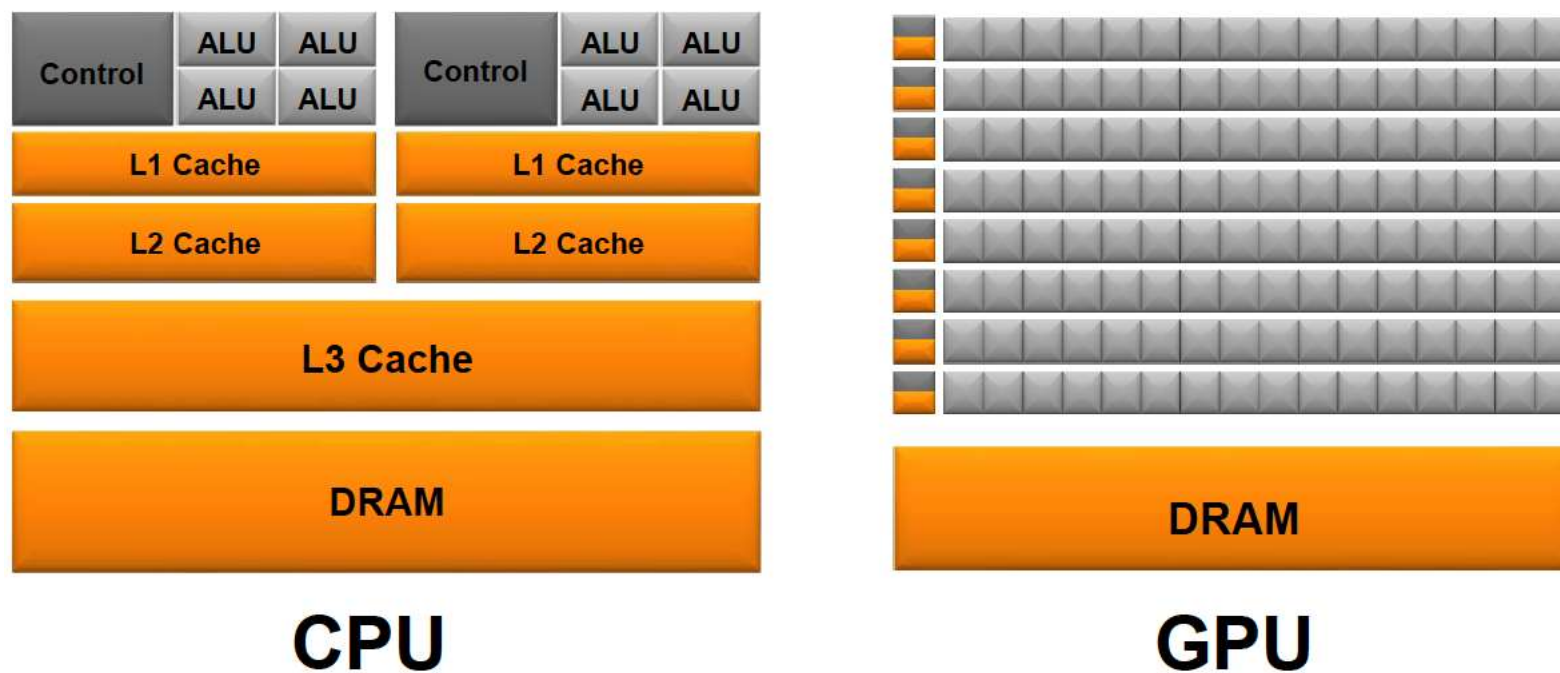


Paralelismo portanto é ótimo para operações com matrizes, tarefas típicas em Deep Learning.





# Estrutura de Programação Paralela em GPU



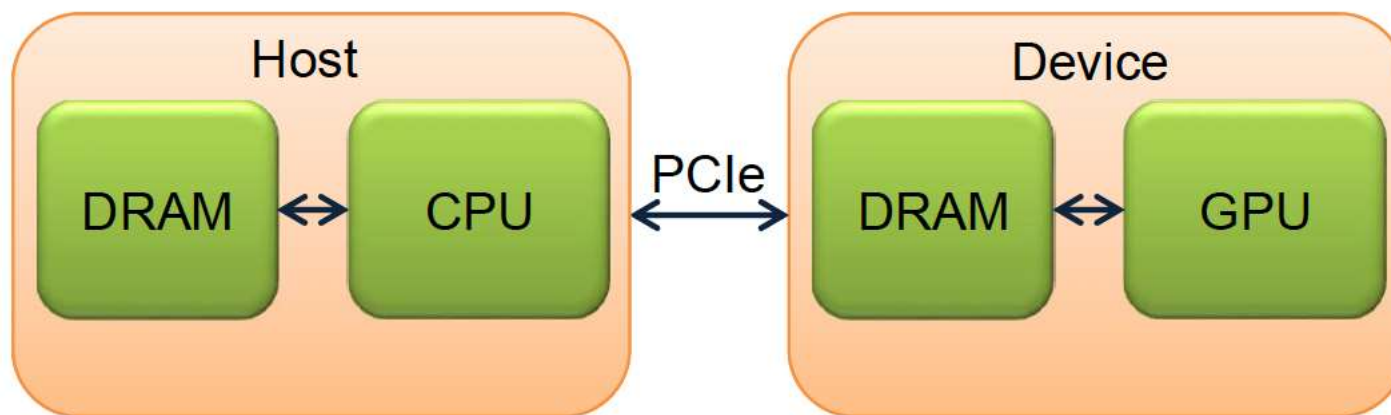
O controle e fluxo simplificado da GPU, facilita o paralelismo





# Estrutura de Programação Paralela em GPU

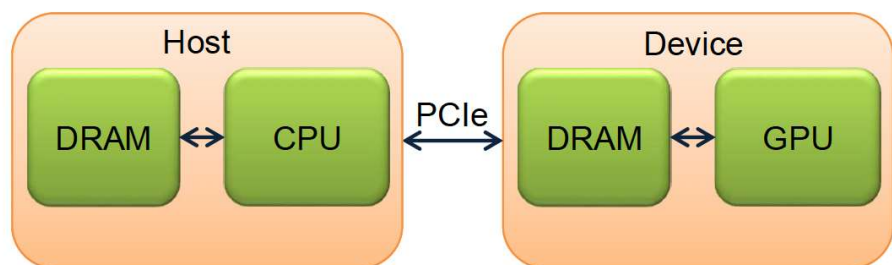
CUDA é um modelo de programação paralelo heterogêneo para ambos host (CPU) e device (GPU)





# Estrutura de Programação Paralela em GPU

CUDA é um modelo de programação paralelo heterogêneo para ambos host (CPU) e device (GPU)



- O código executado no device é chamado de kernel
- Kernels são funções C/C++ com algumas restrições e extensões
- Somente um kernel pode ser executado por vez
- Cada kernel é executado por várias threads

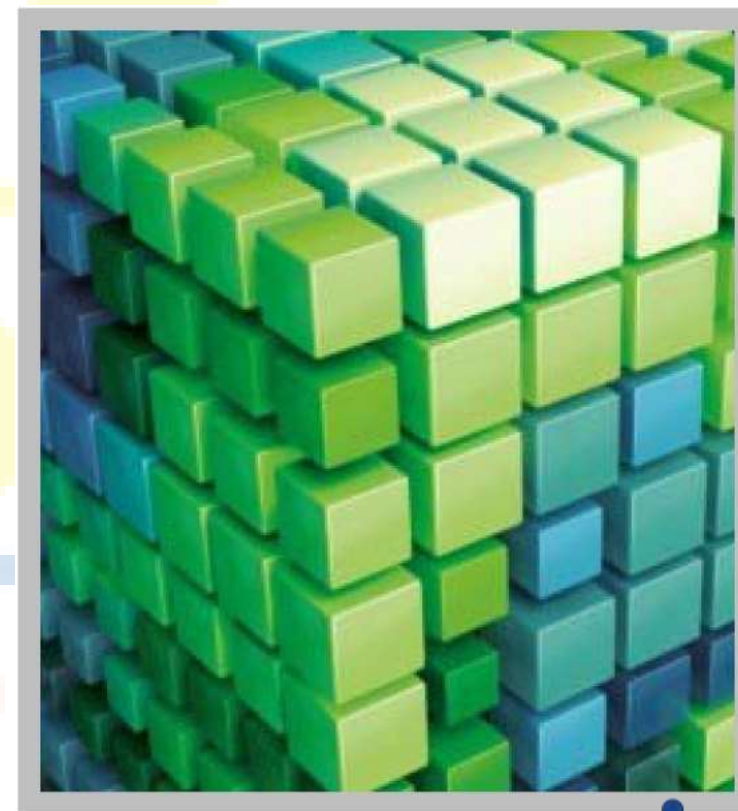




# Estrutura de Programação Paralela em GPU

## CUDA Thread

- Cada thread realiza as mesmas operações em subsets da estrutura de dados
- Threads executam de forma independente
- CUDA Threads executam sempre o mesmo kernel (uma Thread não pode executar 2 kernels ao mesmo tempo)
- Com programas escritos na plataforma CUDA podemos executar milhares de Threads

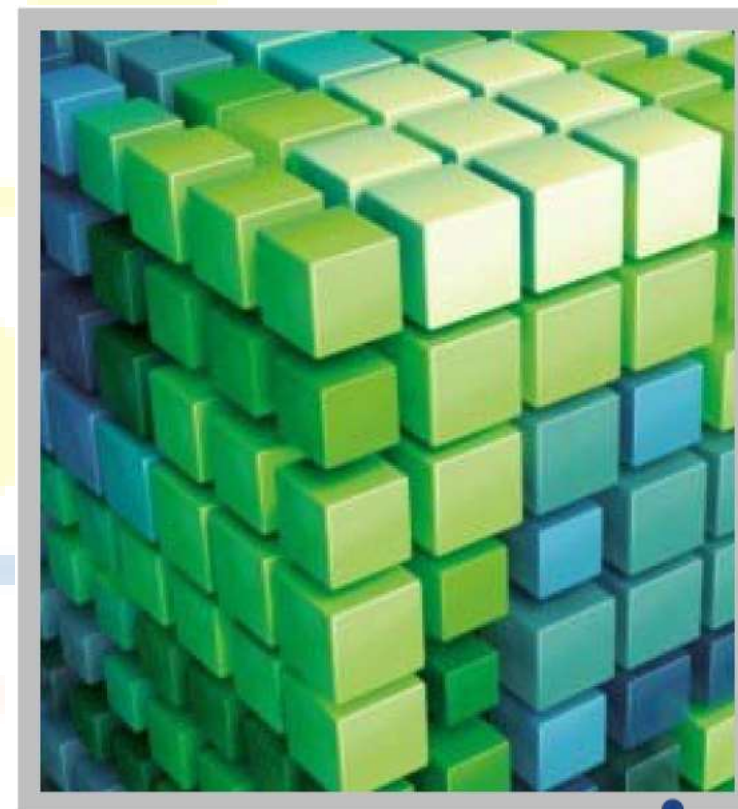




# Estrutura de Programação Paralela em GPU

## CUDA Thread

- As Threads são agrupadas em blocos
- Os blocos são agrupados em uma Grid

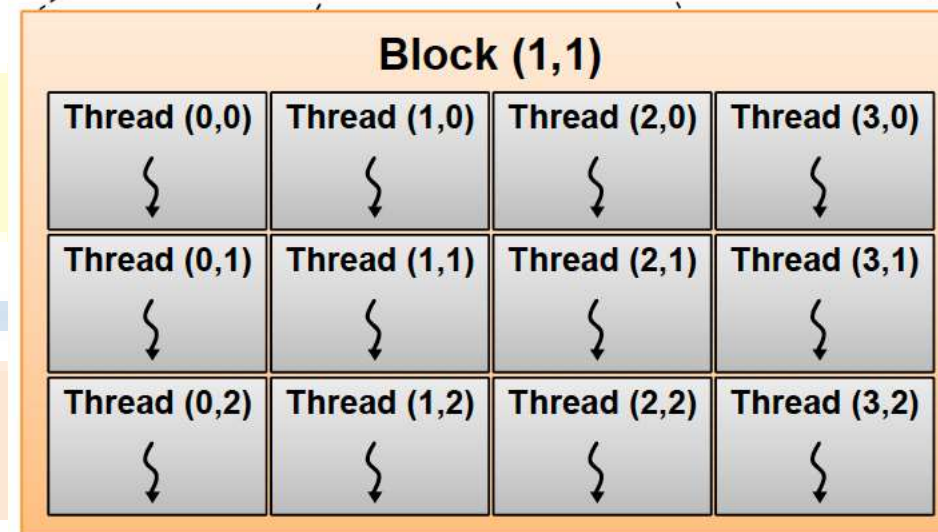
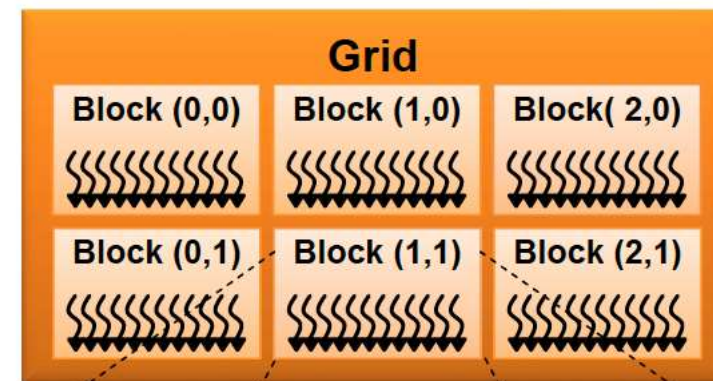




# Estrutura de Programação Paralela em GPU

## CUDA Thread Hierarquia

- Thread Blocks e Grids podem ser 1D, 2D ou 3D
- A dimensão é definida quando começa a execução do kernel
- Thread Blocks e Grids não precisam ter a mesma dimensionalidade. Por exemplo: podemos ter um Grid 1D de Blocos 2D de Threads



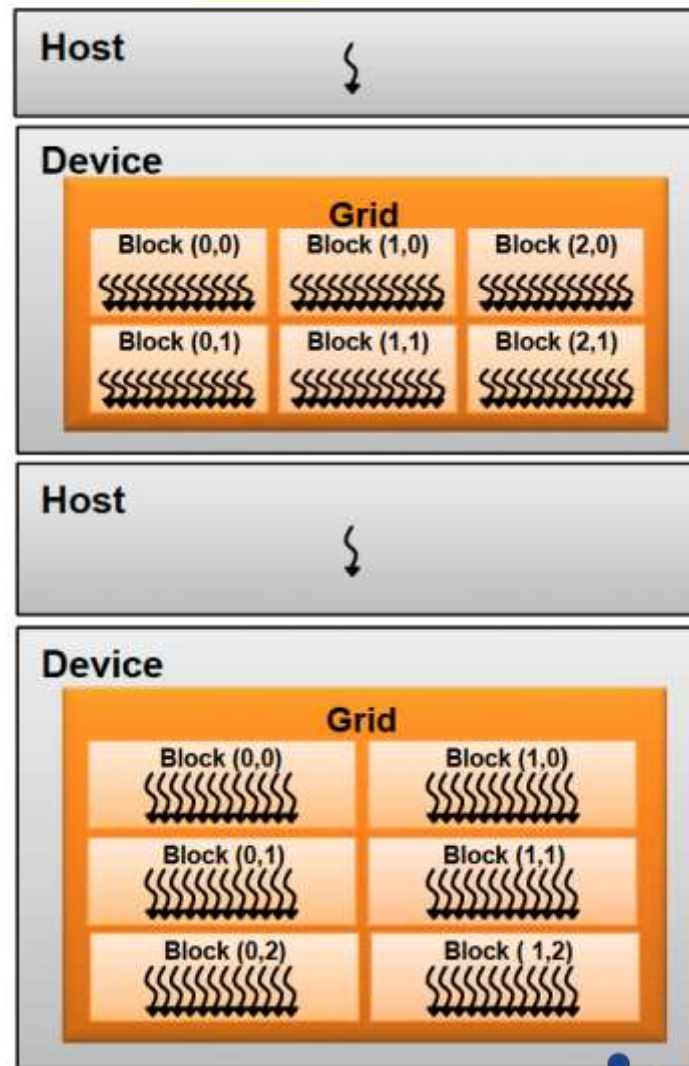




# Estrutura de Programação Paralela em GPU

## Modelo de Programação CUDA

- O host dispara os kernels
- O host executa código sequencial entre as execuções dos kernels no device
- Dados são transferidos entre CPU/GPU (host/device)





# Estrutura de Programação Paralela em GPU

## CUDA APIs

- CUDA usa CUDA C (Runtime API) e low-level Driver API
- Driver API requer gerenciamento explícito de recursos e código para interação direto com o hardware
- CUDA C foi construído sobre o Driver API
- Atenção com a documentação
  - `cuFunctionName()` – Driver API
  - `cudaFunctionName()` – Runtime API





# Estrutura de Programação Paralela em GPU

## CUDA kernel

Iniciar um CUDA Kernel é como iniciar uma função em C, usando uma sintaxe modificada:  
`kernelName <<< dimGrid, dimBlock >>> (...)`

		Compute Capability		
		2.x (Fermi)	3.x (Kepler) 5.x (Maxwell)	6.x (Pascal)
Total Threads per Block		1024	1024	1024
Grid Size	dGrid.x	65535	$2^{31}-1$	$2^{31}-1$
	dGrid.y	65535	65535	65535
	dGrid.z	65535	65535	65535
Block Size	dBlock.x	1024	1024	1024
	dBlock.y	1024	1024	1024
	dBlock.z	64	64	64





# Estrutura de Programação Paralela em GPU

## CUDA kernel

- Um kernel é indicado pelo qualificador `__global__`
- Chamado pelo host, executado no device
- CUDA kernels não tem acesso a memória do host (memória RAM)

```
__global__ void SimpleKernel(float* a, float b)
{
    a[0] = b;
}
```





# Estrutura de Programação Paralela em GPU

## CUDA kernel

- Precisam retornar *void*
- Não tem acesso às funções do host
- Ponteiros para device memory

```
__global__ void SimpleKernel(float* a, float b)
{
    a[0] = b;
}
```







# Estrutura de Programação Paralela em GPU

## CUDA kernel

- Kernels possuem variáveis built-in como: blockIdx, blockDim, threadIdx
- O tamanho de blocos e grids não variam durante a execução

```
__global__ void SimpleKernel(float* a, float b)
{
    a[0] = b;
}
```





# Estrutura de Programação Paralela em GPU

## CUDA kernel - Sintaxe

As variáveis built-in são normalmente usadas para identificar de forma única as threads, mapeando o ID de uma thread ao global array index

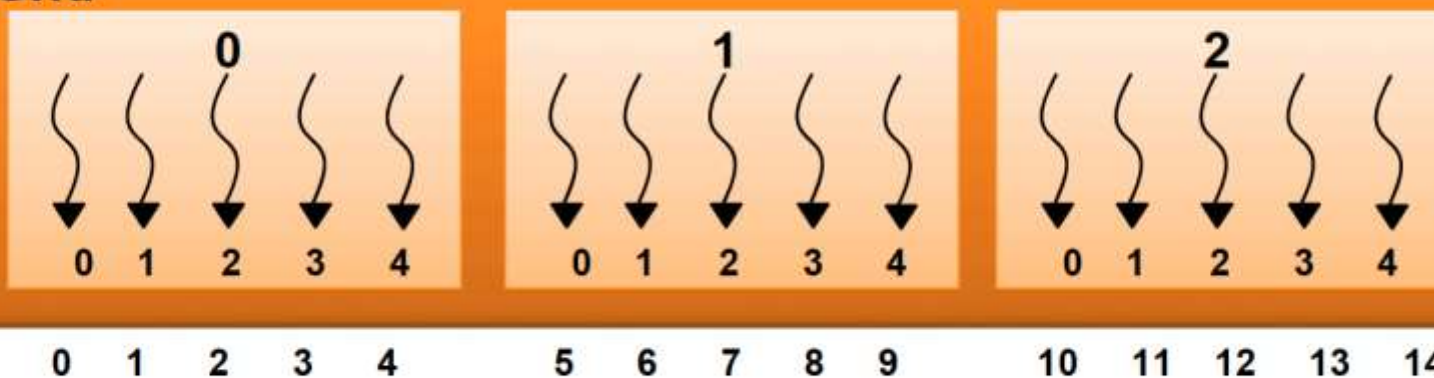
`myKernel<<<3,5>>>(...)`

`blockDim.x = 5`  
`gridDim.x = 3`

`blockIdx.x`

`threadIdx.x`

**Grid**



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

→ **Global Index = 1 \* 5 + 3 = 8**





# Estrutura de Programação Paralela em GPU

## CUDA Suporte a C++

### Suportado

- Classes
- Herança
- Precisamos adicionar qualificadores `__device__` quando executando código apenas na GPU
- Templates
- C++11 features incluindo funções auto e lambda

### Não suportado

- C++ Standard Library
- Run time type information (RTTI)
- Exception handling
- Classes com funções virtuais não são compatíveis binariamente entre o host e o device





# Estrutura de Programação Paralela em GPU

Funções definidas pelo usuário para o device

```
__device__ float myDeviceFunction(int i)
{
    ...
}

__global__ void myKernel(float* a)
{
    int idx = blockIdx.x*blockDim.x+threadIdx.x;
    a[idx] = myDeviceFunction(idx);
}
```

Funções declaradas com ambos qualificadores `__device__` e `__host__` serão compilados para ambos CPU e GPU





# Estrutura de Programação Paralela em GPU

```
void VectorAdd(float* aH, float* bH, float* cH, int N)
{
    float* aD, *bD, *cD;
    int N_BYTES = N * sizeof(float);
    dim3 blockSize, gridSize;

    cudaMalloc((void*)&aD, N_BYTES);
    cudaMalloc((void*)&bD, N_BYTES);
    cudaMalloc((void*)&cD, N_BYTES);

    cudaMemcpy(aD, aH, N_BYTES, cudaMemcpyHostToDevice);
    cudaMemcpy(bD, bH, N_BYTES, cudaMemcpyHostToDevice);

    blockSize.x = 512;
    gridSize.x = N / blockSize.x;
    VectorAddKernel<<<gridSize, blockSize>>>(aD, bD, cD);

    cudaMemcpy(cH, cD, N_BYTES, cudaMemcpyDeviceToHost);
}
```

This code assumes N  
is a multiple of 512

Allocate memory on  
GPU

Transfer input  
arrays to GPU

Launch kernel

Transfer output  
array to CPU







# Estrutura de Programação Paralela em GPU

## Desvantagens Das GPUs

- Arquitetura não é tão flexível quanto das CPUs
- Os algoritmos precisam ser reescritos para rodar em paralelo
- A velocidade da GPU pode ser limitada pela conexão com a CPU, via PCIe
- Memória limitada – 8 a 24 GB



# Padrões de Programação Paralela



# Padrões de Programação Paralela

Como Executar Aplicações em Paralelo?

Computadores  
em Cluster

Múltiplas  
Threads em  
CPU

GPU





# Padrões de Programação Paralela



- **Diferentes Tipos de Memória**
  - Shared vs. Private
  - Velocidade de Acesso
- **Dados em Arrays**
  - CUDA não oferece estruturas de dados para execução em paralelo
  - O compilador CUDA não oferece suporte a "auto-parallelization/vectorization"
  - Sem equivalente a CPU-type SIMD
- **Restrições do Compilador**
  - NVCC não suporta padrão C++11

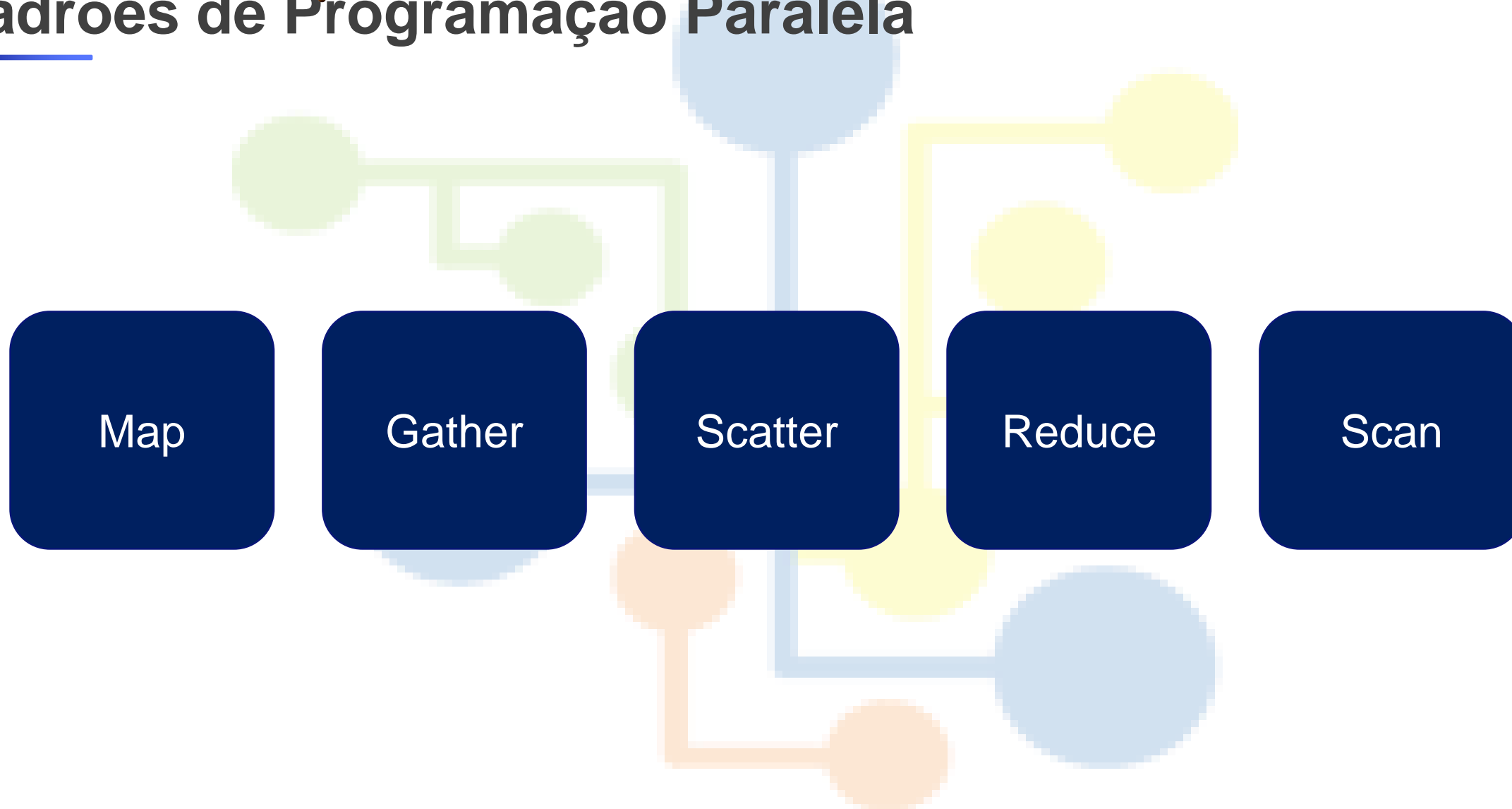


# Padrões de Programação Paralela

## Map, Gather e Scatter



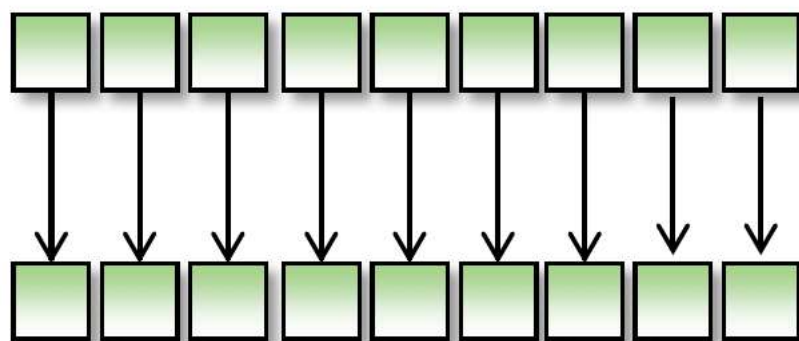
# Padrões de Programação Paralela







# Padrões de Programação Paralela



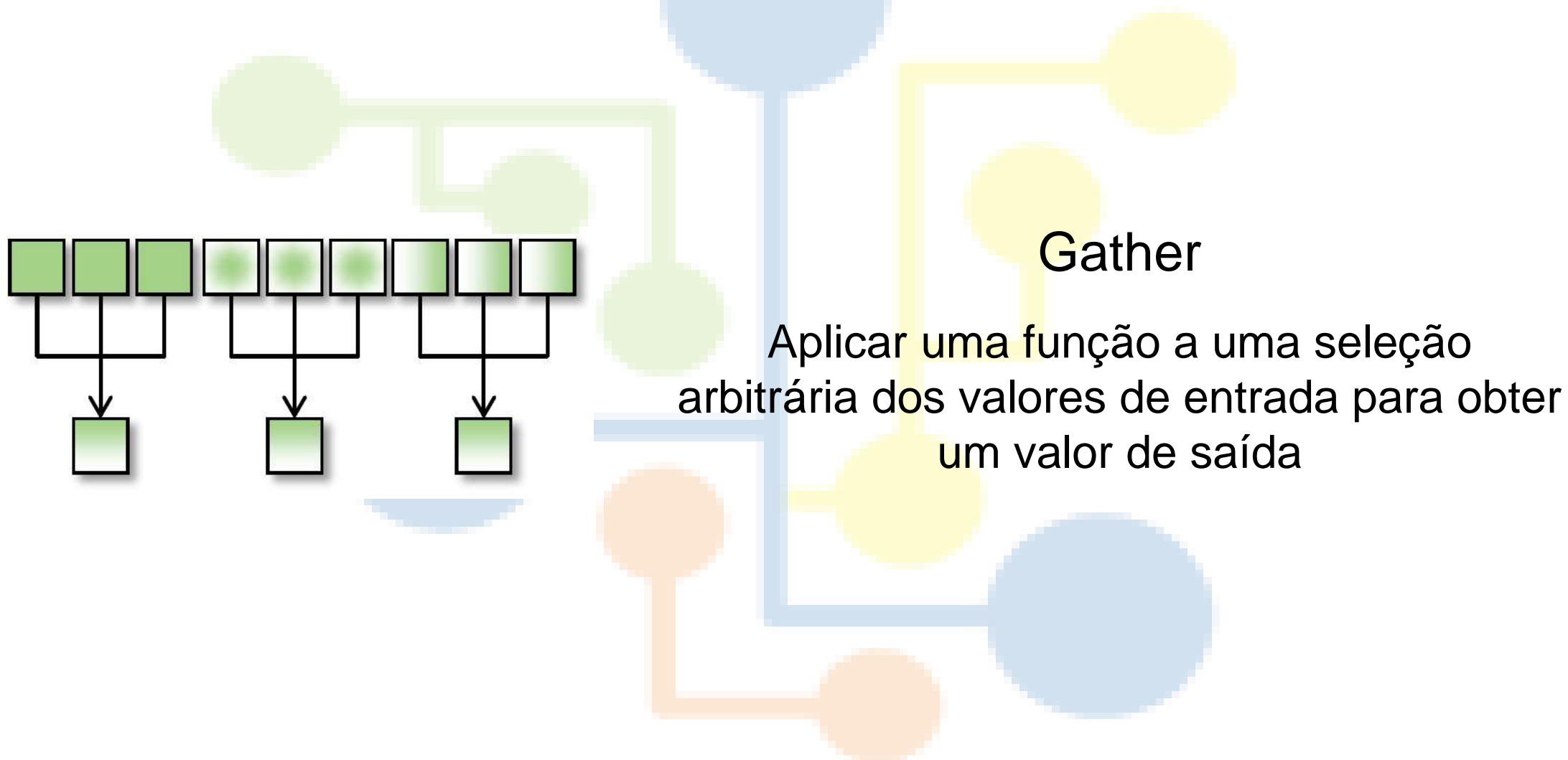
Map

Aplicar uma função a um array e replicar a função sobre cada elemento do array



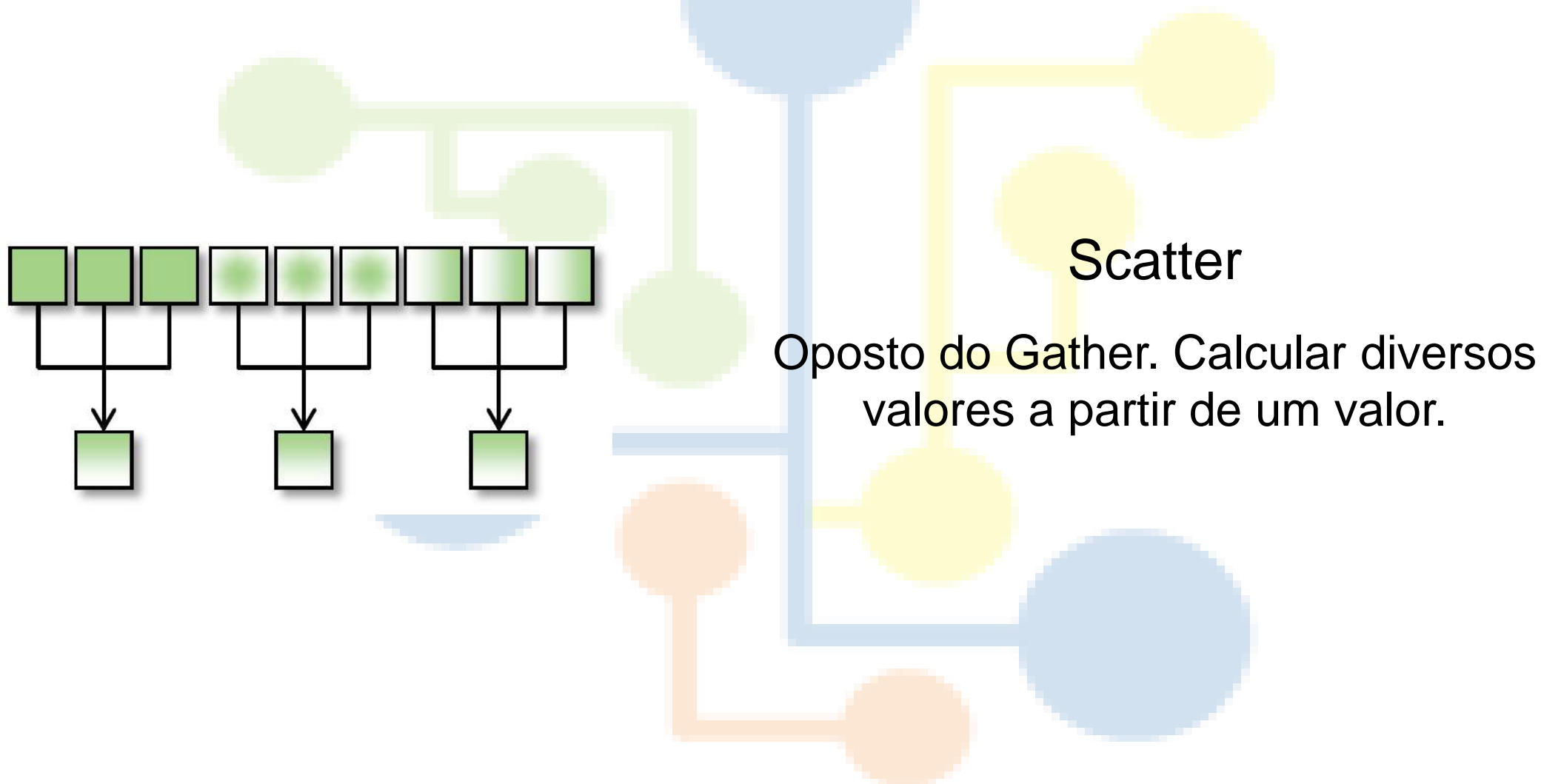


# Padrões de Programação Paralela





# Padrões de Programação Paralela

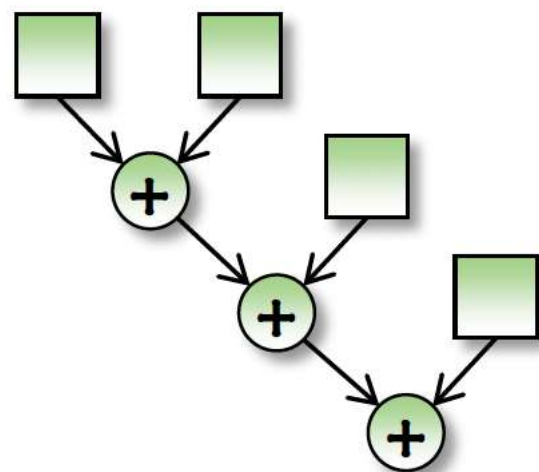


# Padrões de Programação Paralela

## Reduce e Scan

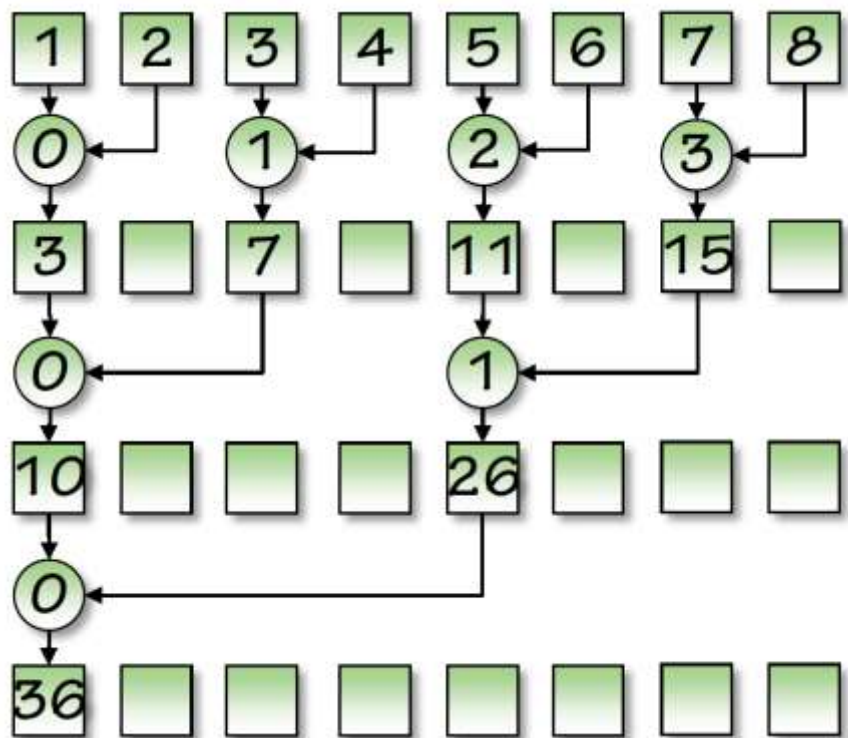


# Padrões de Programação Paralela





# Padrões de Programação Paralela



**Reduce** de forma  
paralela

**Reduce**

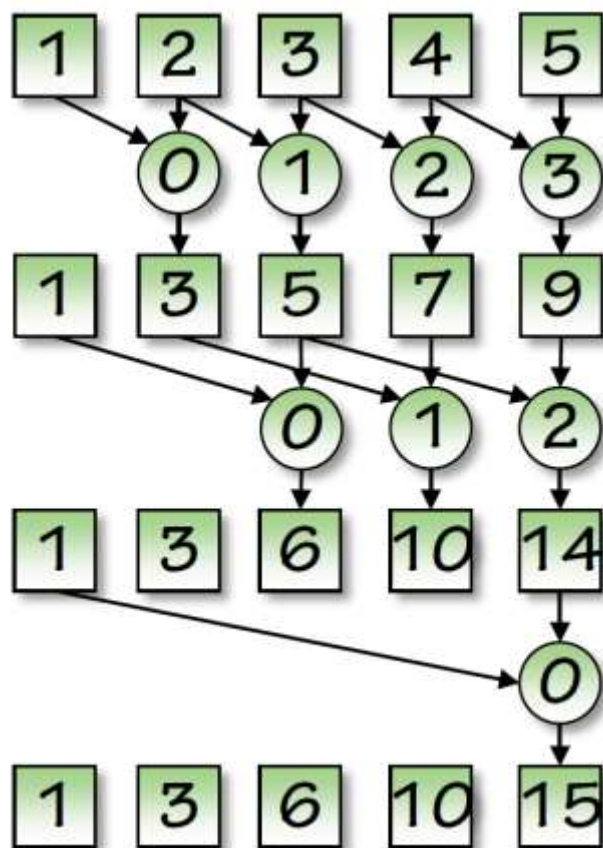
Operação pair-wise para cada elemento do array. Muito usada para soma de elementos de forma paralela.







# Padrões de Programação Paralela



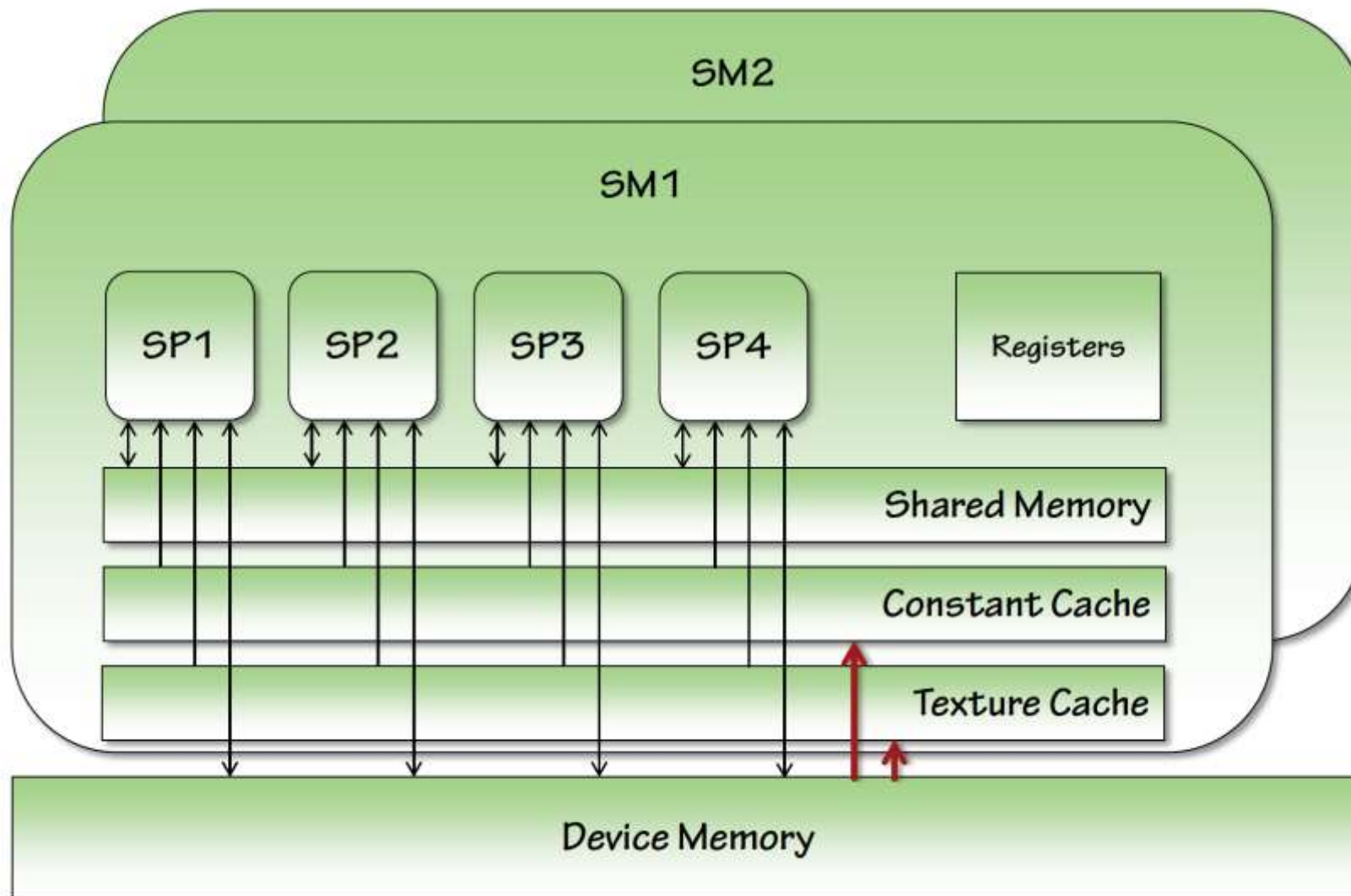
Scan

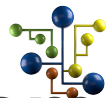
Cada output é calculado como uma função envolvendo todos os inputs dos valores anteriores.

# Tipos de Memória da GPU



# Tipos de Memória da GPU

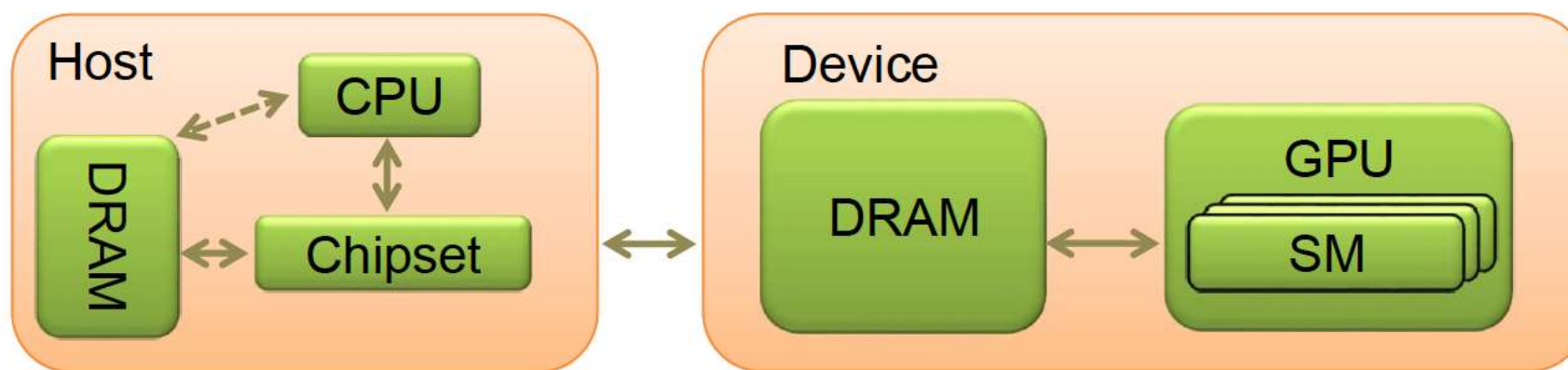




# Tipos de Memória da GPU

Cada GPU é composta de um ou mais Streaming Multiprocessors (SMs)

- Cada SM possui uma coleção de recursos computacionais:
  - Processadores (cores)
  - Registradores
  - Memória especializada





# Tipos de Memória da GPU

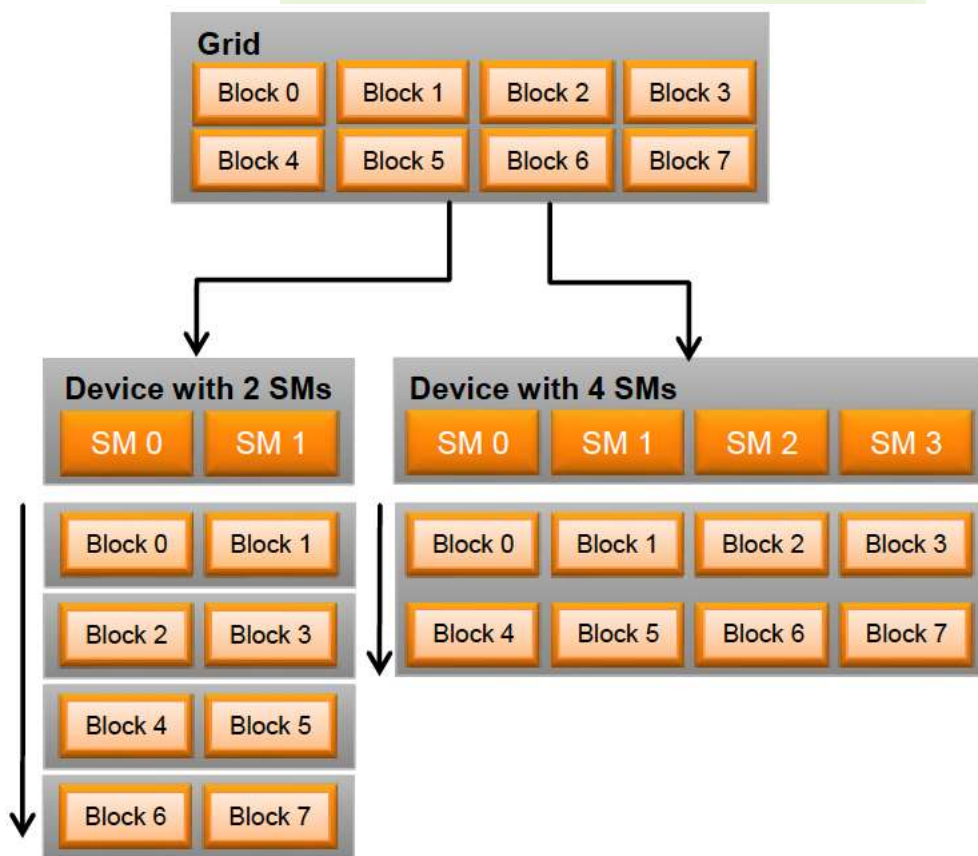
Streaming Multiprocessors na GPU

NVIDIA GPU	Number of SMs
Tesla K40	15
Tesla K80	2 x 13
Tesla P100	56
Quadro M3000M	8





# Tipos de Memória da GPU



- Os blocos são distribuídos através dos SMs.
- Não temos controle sobre como será a distribuição.
- Um bloco vai ser executado em um único SM.



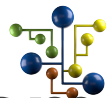




# Tipos de Memória da GPU

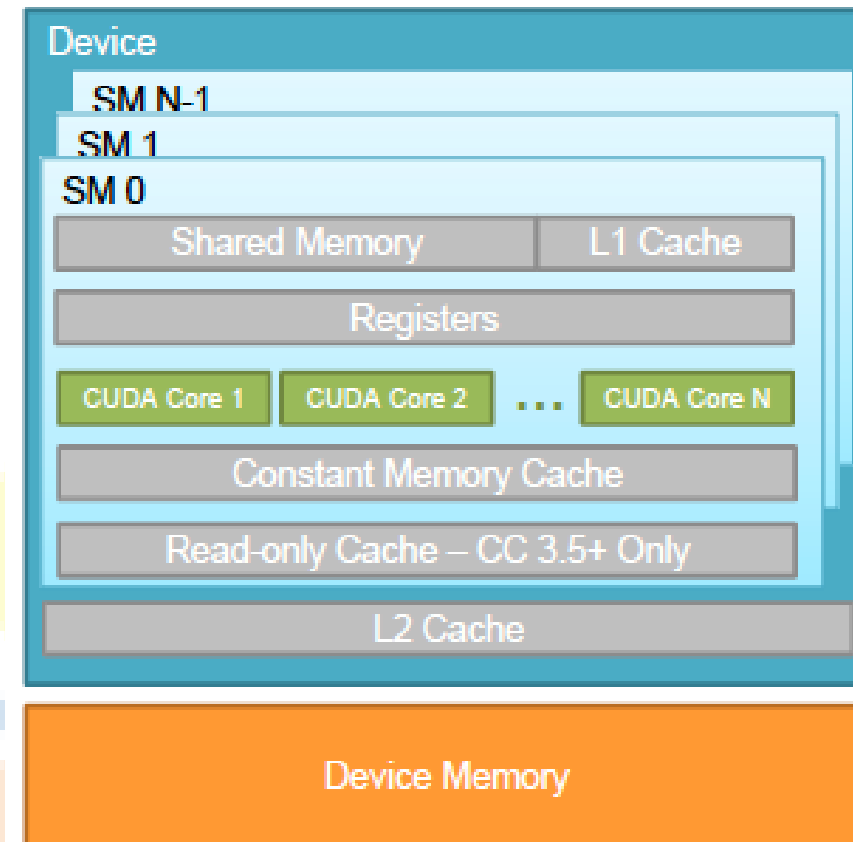
- Muitos tipos de memória estão disponíveis na GPU, com diferentes níveis de performance.
- Os dados precisam ser mapeados para o tipo de memória correto:
  - Shared Memory
  - Registradores
  - Constant Cache
  - Device Memory
  - Read-only Cache





# Tipos de Memória da GPU

- Muitos tipos de memória estão disponíveis na GPU, com diferentes níveis de performance.
- Os dados precisam ser mapeados para o tipo de memória correto:
  - Shared Memory
  - Registradores
  - Constant Cache
  - Device Memory
  - Read-only Cache

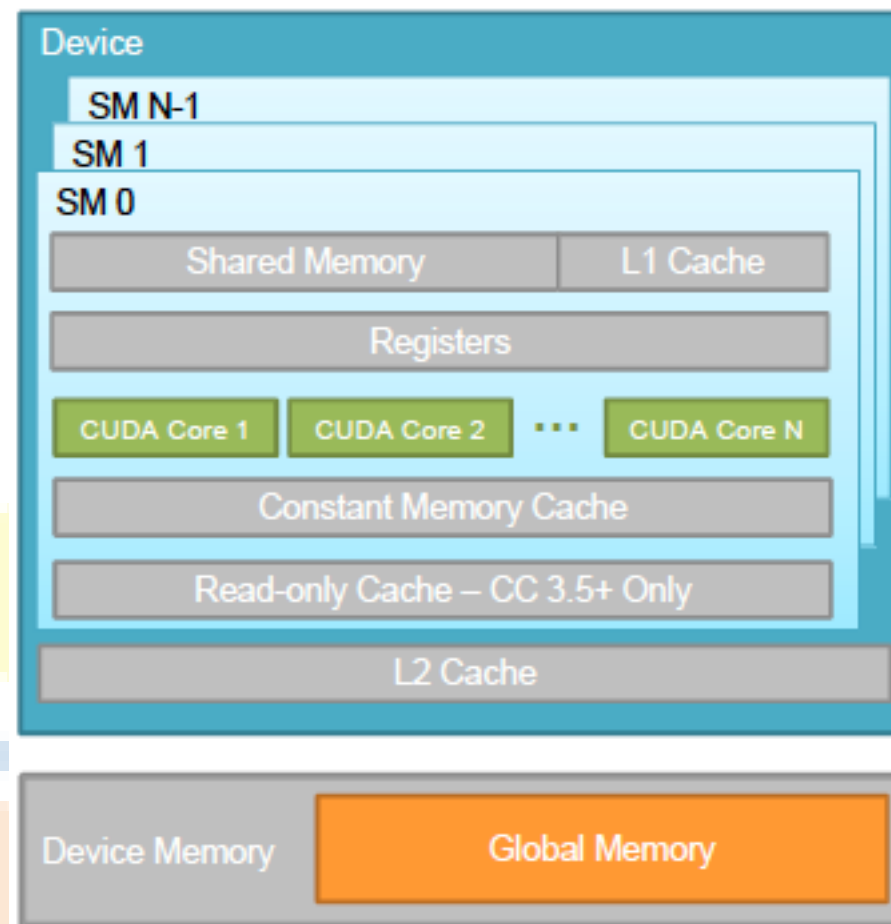




# Tipos de Memória da GPU

## Device Memory (Global Memory):

- Disponível a todas as threads.
- Dados nesta memória persistem entre execuções de kernels.
- Precisamos explicitamente alocar a memória usando `cudaMalloc` e `cudaFree`.

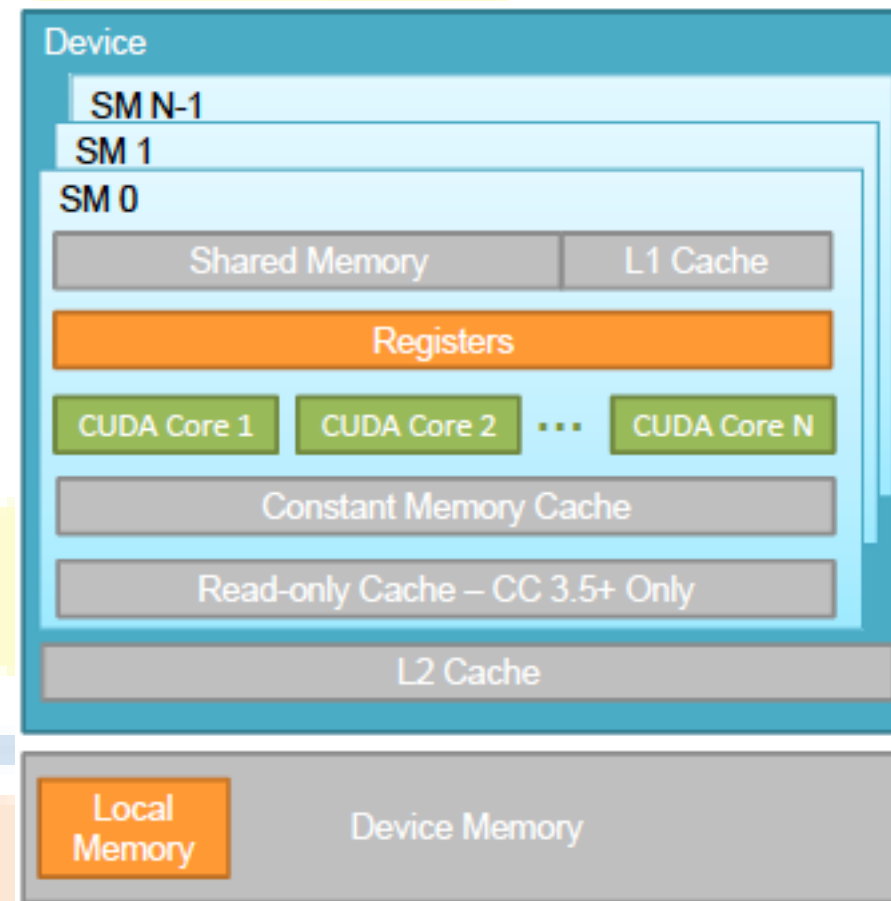


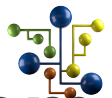


# Tipos de Memória da GPU

## Memória por Thread (Registradores):

- O compilador controla onde as variáveis são armazenadas na memória física.
- Podem ser de 2 tipos:
  - Registradores (on-chip) – tipo mais rápido de memória existem na GPU.
  - Local Memory (off-chip) – o compilador aloca parte da Device Memory, quando o número de registradores não for mais suficiente.

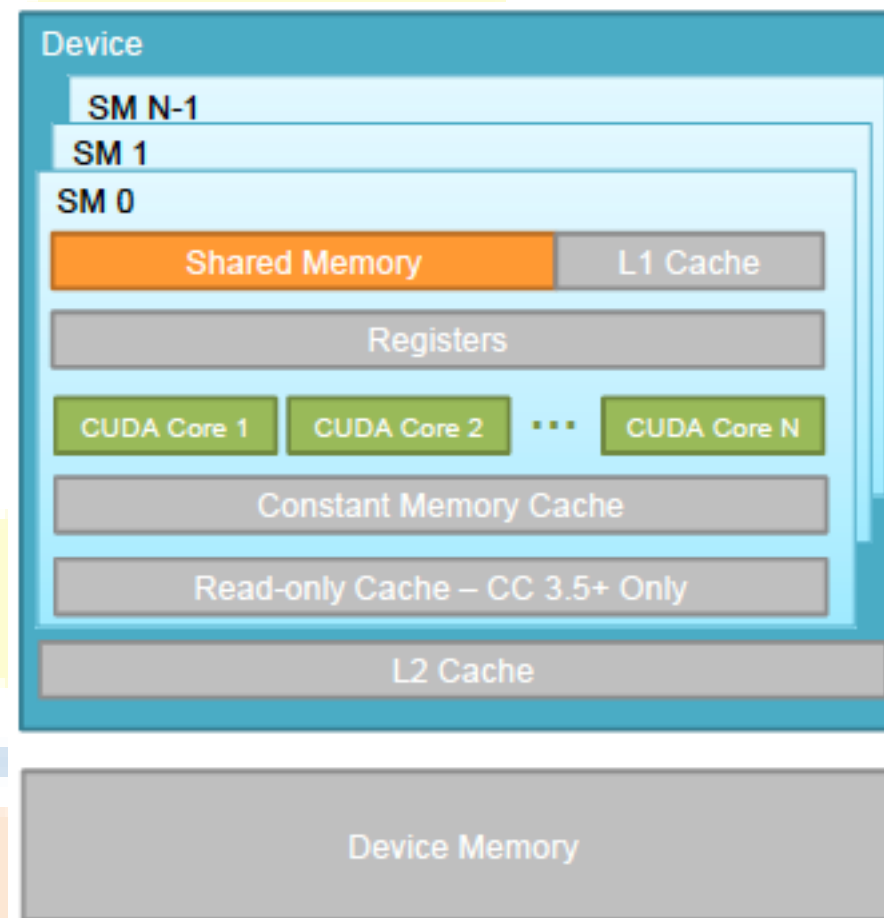




# Tipos de Memória da GPU

## Shared Memory:

- Memória de alta performance.
- Performance superior a Device Memory.
- Somente visível para threads dentro de um mesmo bloco.
- Semelhantes as benefícios de se usar o cache da CPU.
- Precisa ser explicitamente gerenciada pelo desenvolvedor.





# Tipos de Memória da GPU

## CUDA Syntax - Shared Memory

```
// Static shared memory syntax

#define BLOCK_SIZE 256

__global__ void kernel(float* a)
{
    __shared__ float sData[BLOCK_SIZE];
    int i;
    i = blockIdx.x * blockDim.x + threadIdx.x;
    sData[threadIdx.x] = a[i];
    __syncthreads();
    ...
    a[i] = sData[blockDim.x - 1 - threadIdx.x];
}

int main(void)
{
    ...
    kernel<<<nBlocks, BLOCK_SIZE>>>(...);
    ...
}
```

- O qualificador `__shared__` é usado para declarar variáveis/arrays na shared memory.
- A shared memory não é visível para o host.
- As threads podem ler/escrever na shared memory.



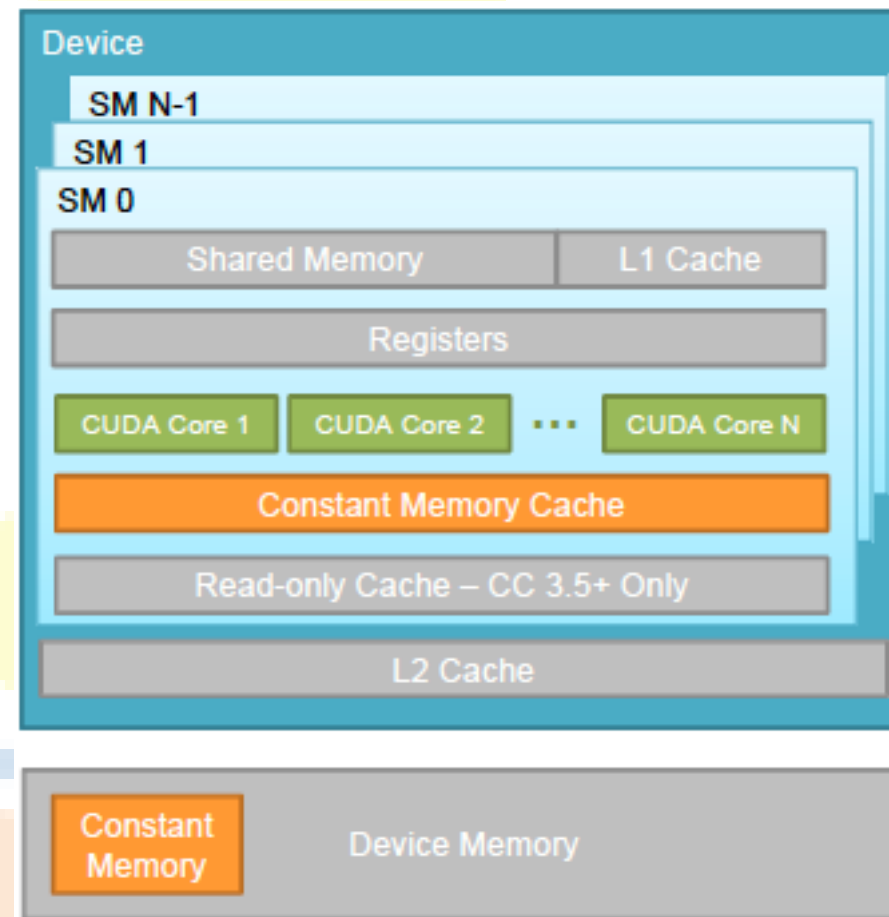




# Tipos de Memória da GPU

## Constant Memory:

- Região especial no Device Memory.
- Read-only para o kernel.
- Read-write para o host.
- Constantes são declaradas no escopo do arquivo.
- 64 KB.
- `cudaMemcpyToSymbol` para o cache dos valores constantes dentro do SM.





# Tipos de Memória da GPU

## CUDA Syntax – Constant Memory

- O qualificador `__constant__` é usado para declarar variáveis/arrays como constantes residentes na memória.
- Variáveis constantes podem ser lidas/escritas pelo host, mas são read-only para o kernel.

```
__constant__ float staticCoeff = 1.0f;
__constant__ float runtimeCoeff;
__constant__ float runtimeArray[5];

__global__ void kernel(float *array)
{
    array[threadIdx.x] += staticCoeff;
    array[threadIdx.x] *= runtimeCoeff;
    array[threadIdx.x] = runtimeArray[0];
}

int main(void)
{
    float val = calculateCoefficient();
    cudaMemcpyToSymbol(runtimeCoeff, &val,
                        sizeof(val));

    cudaMemcpyToSymbol(runtimeArray,
                        hostArray,
                        5*sizeof(float));

    kernel<<<gSize,bSize>>>(-);
}
```

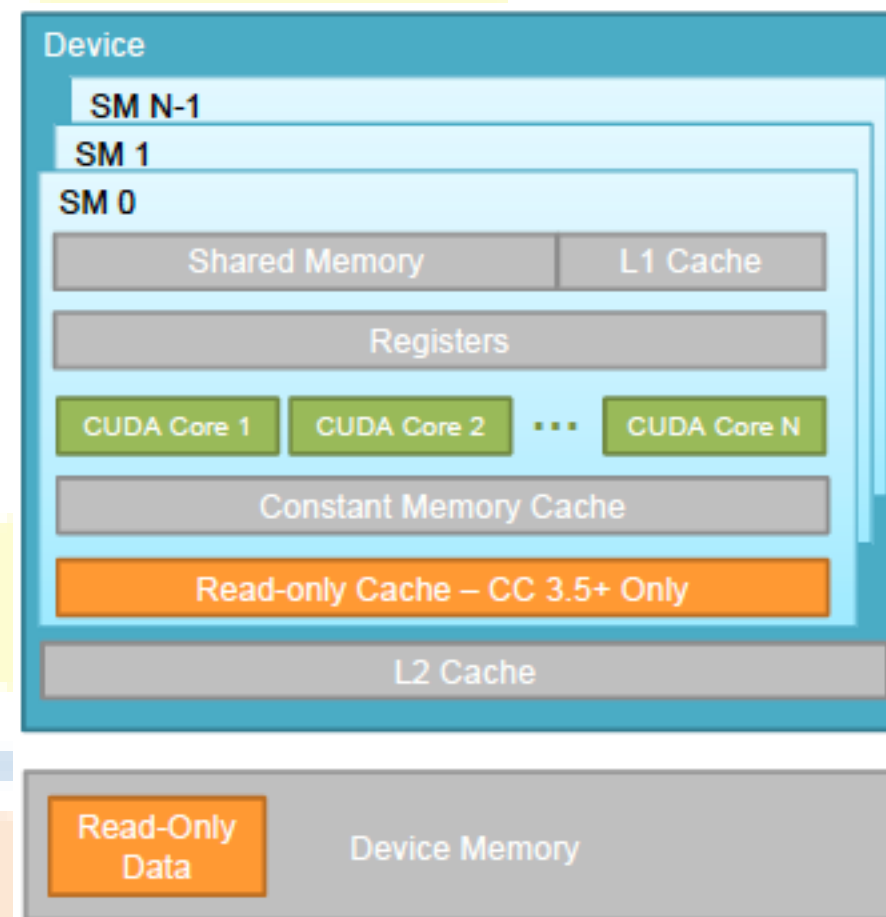




# Tipos de Memória da GPU

## Read-only Cache:

- Usada originalmente como Texture Cache.
- Aloca e gerencia a Device Memory.
- Para ser usada o desenvolvedor deve especificar o qualificador **const \_\_restrict\_\_** na variável.
- 12 a 48 KB.

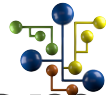




# Tipos de Memória da GPU

Comunicação entre threads é possível em CUDA?





# Tipos de Memória da GPU

Processamento concorrente oferece riscos devido a ordem com que as tarefas são executadas.

Possíveis problemas: race condition, deadlocks, starvation.





# Tipos de Memória da GPU

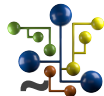
## Modelo de Memória CUDA

Memory Space	Managed by	Physical Implementation	Scope on GPU	Scope on CPU	Lifetime
Registers	Compiler	On-chip	Per Thread	Not visible	Lifetime of a thread
Local	Compiler	Device Memory	Per Thread	Not visible	
Shared	Programmer	On-chip	Block	Not visible	Block lifetime
Global	Programmer	Device Memory	All threads	Read/Write	Application or until explicitly freed
Constant	Programmer	Device Memory	All threads Read-only	Read/Write	



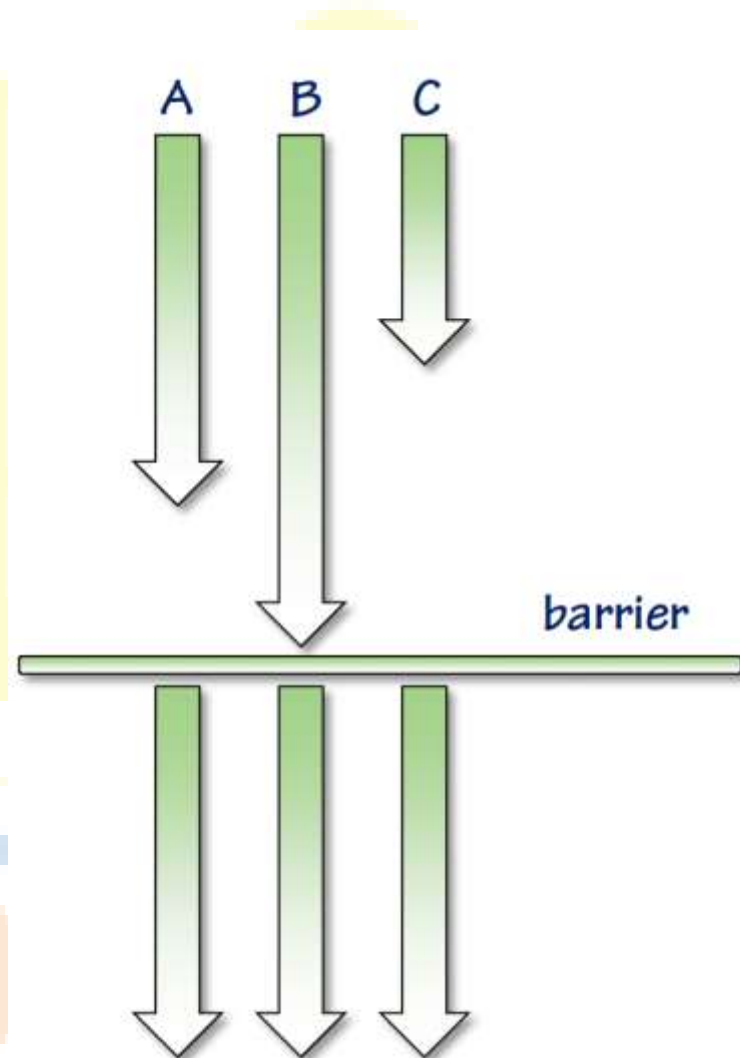


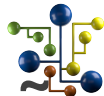
# Sincronização de Threads



# Sincronização de Threads

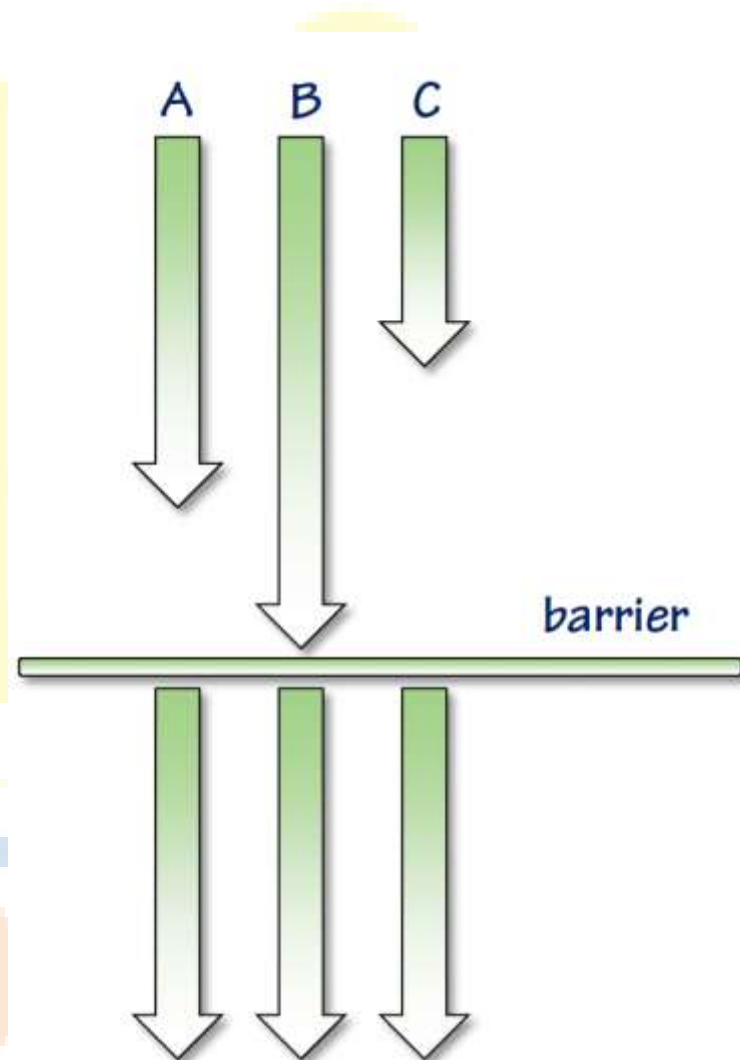
- Threads podem levar diferentes quantidades de tempo para completar uma parte de uma computação.
- Às vezes, você quer que todas as threads atinjam um ponto particular antes de continuar seu trabalho.
- CUDA fornece uma função de barreira de thread chamada `__syncthreads()`.

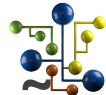




# Sincronização de Threads

- `__syncthreads()` somente sincroniza threads dentro de um bloco

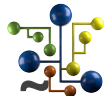




# Sincronização de Threads

Problemas gerados pela concorrência de tarefas, são eliminados ou evitados, através de sincronização.





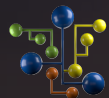
# Sincronização de Threads

Possíveis problemas gerados pela concorrência:

- Tentativa de comunicação entre blocos resulta em comportamento indefinido.
- Comunicação entre threads no mesmo bloco, via shared memory ou global memory geram riscos na concorrência de tarefas.
- Usamos `void __syncthreads();`
  - Sincroniza todas as threads em um bloco
  - Nenhuma thread prossegue até que todas tenham atingido uma barreira







Data Science  
Academy

Data Science Academy [raphaelbsfontenelle@gmail.com](mailto:raphaelbsfontenelle@gmail.com) 615c1fdde32fc361b30c9ec2

# Obrigado



Data Science Academy



Data Science Academy