

CUDA C/C++ BASICS

Thiago Vivan Bastos
Fernando William Cruz
Bruno Martins Valério Bomfim

O que é CUDA?

- Arquitetura CUDA
 - Utiliza GPU para computação de propósito geral
 - Mantém performance
- CUDA C/C++
 - Baseado no C/C++ padrão
 - APIs para utilizar devices, memória, etc.

Introdução ao CUDA C/C++

- O que vamos aprender nessa seção?
 - começamos com “Hello World!”
 - Escrever e executar CUDA C/C++ kernels
 - Gerenciar memória da GPU
 - Gerenciar comunicação e sincronização

Conceitos

Computação Heterogênea

Blocks

Threads

Indexação

Memoria compartilhada

`__syncthreads()`

Operações assíncronas

lidar com erros

Gerenciar dispositivos

Hello World!

CONCEPTS

Computação Heterogênea

Blocks

Threads

Indexação

Memória compartilhada

`__syncthreads()`

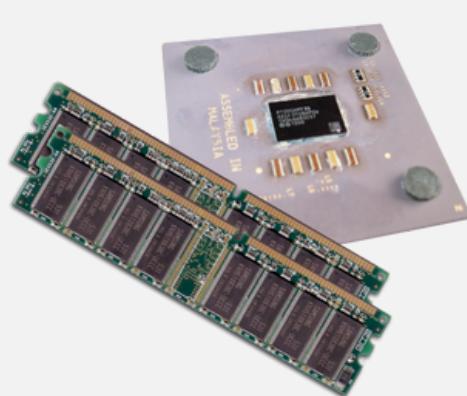
Operações assíncronas

lidar com erros

Gerenciar dispositivos

Computação Heterogênea

- Terminologia:
- *Host* A CPU e sua memoria (host memory)
- *Device* A GPU e sua memoria (device memory)



Host
Hospedeiro



Device
Dispositivo

Computação Heterogênea

```
#include <iostream>
#include <algorithm>

using namespace std;

#define N      1024
#define RADIUS 3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
    shared __int temp[BLOCK_SIZE + 2 * RADIUS];
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;
    int rindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[index] = in[lindex];
    if (threadIdx.x < RADIUS) {
        temp[index - RADIUS] = in[index - RADIUS];
        temp[index + BLOCK_SIZE] = in[index + RADIUS];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[index + offset];

    // Store the result
    out[index] = result;
}

void fill_ints(int *x, int n) {
    fill_n(x, n, 1);
}

int main(void) {
    int *in, *out;           // host copies of a, b, c
    int *d_in, *d_out;       // device copies of a, b, c
    int size = (N + 2*RADIUS) * sizeof(int);

    // Alloc space for host copies and setup values
    in = (int *)malloc(size); fill_ints(in, N + 2*RADIUS);
    out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

    // Alloc space for device copies
    cudaMalloc(&d_in, in_size);
    cudaMalloc(&d_out, out_size, cudaMemcpyHostToDevice);

    // Copy to device
    cudaMemcpy(d_in, in, in_size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_out, out, out_size, cudaMemcpyHostToDevice);

    // Launch stencil_1d() kernel on GPU
    stencil_1d<<N/BLOCK_SIZE,BLOCK_SIZE>>(d_in + RADIUS, d_out + RADIUS);

    // Copy result back to host
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

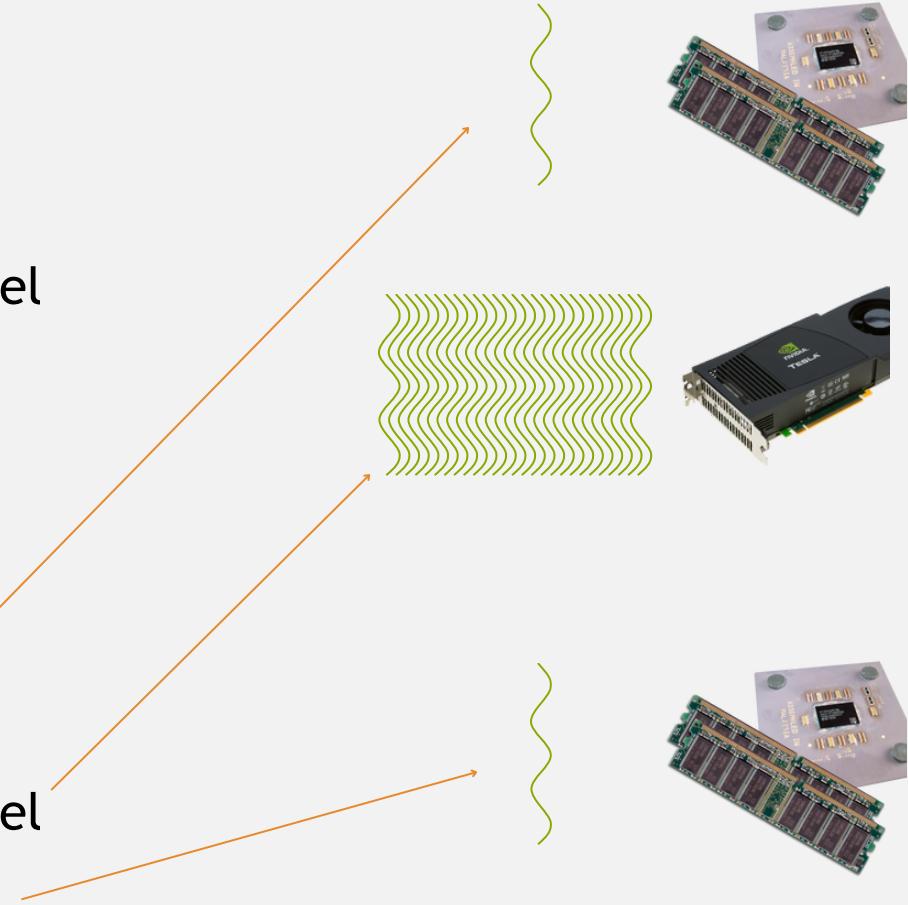
    // Cleanup
    free(in); free(out);
    cudaFree(d_in); cudaFree(d_out);
    return 0;
}
```

parallelizável

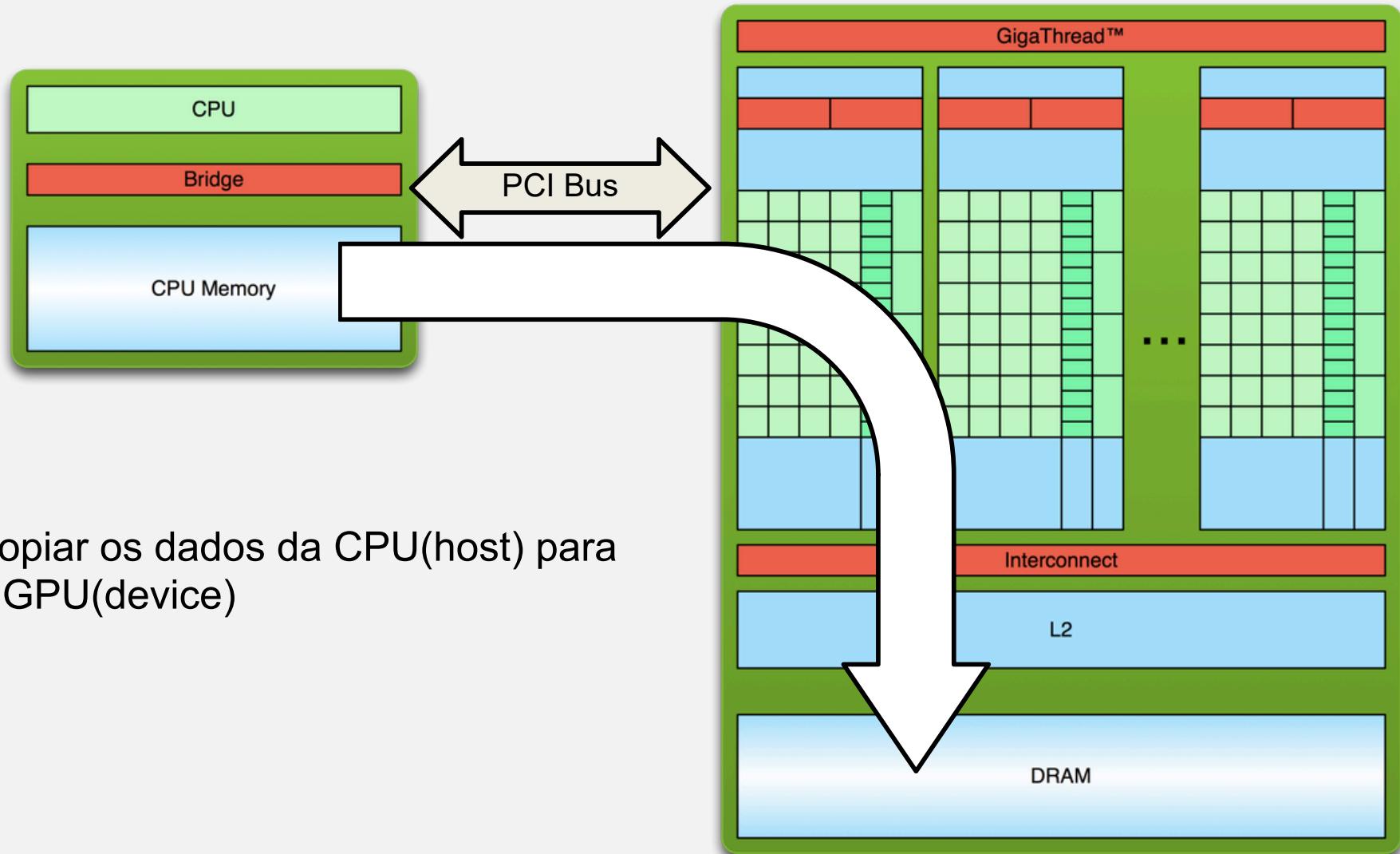
serial

parallelizável

serial

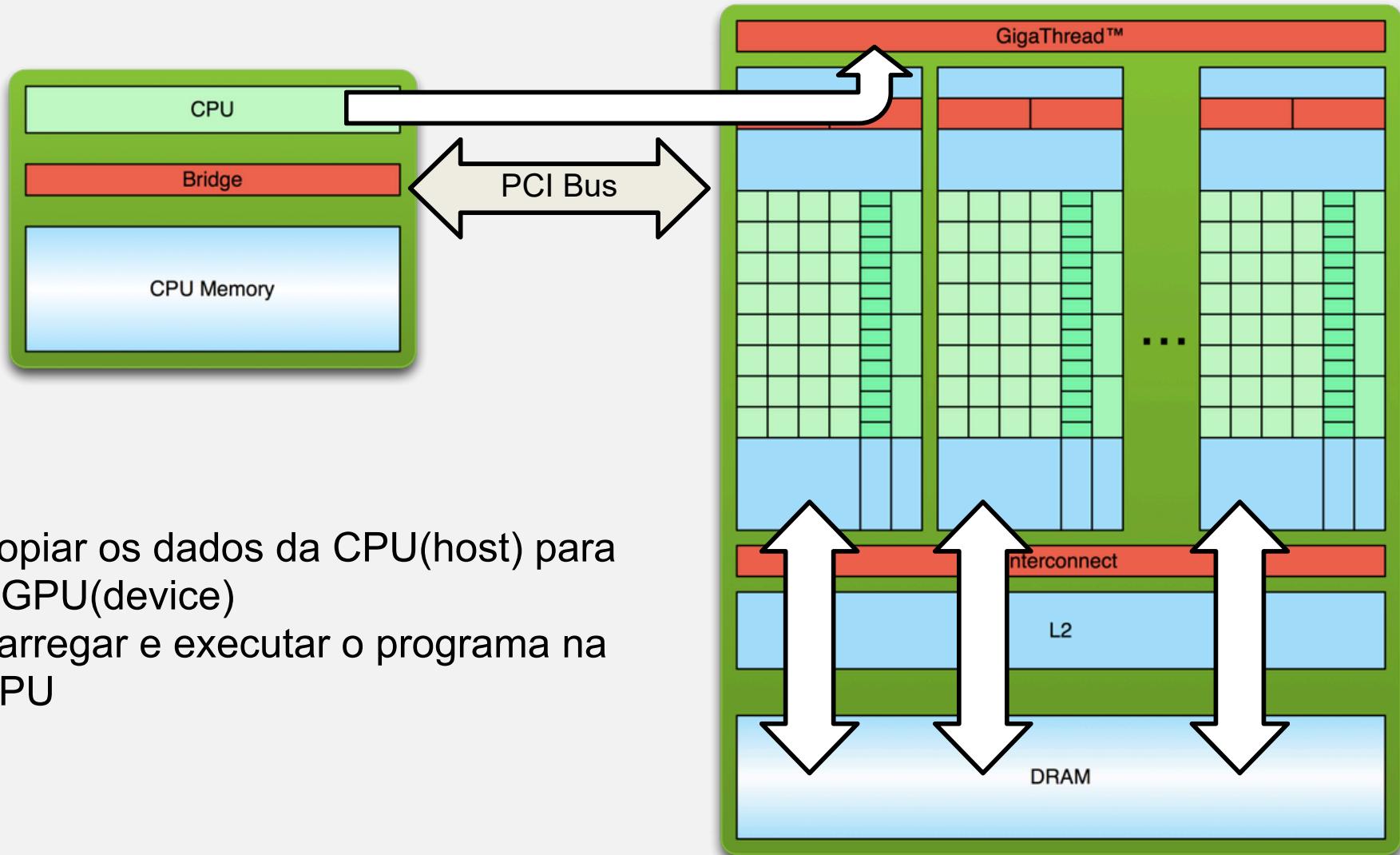


Sequência de Processos

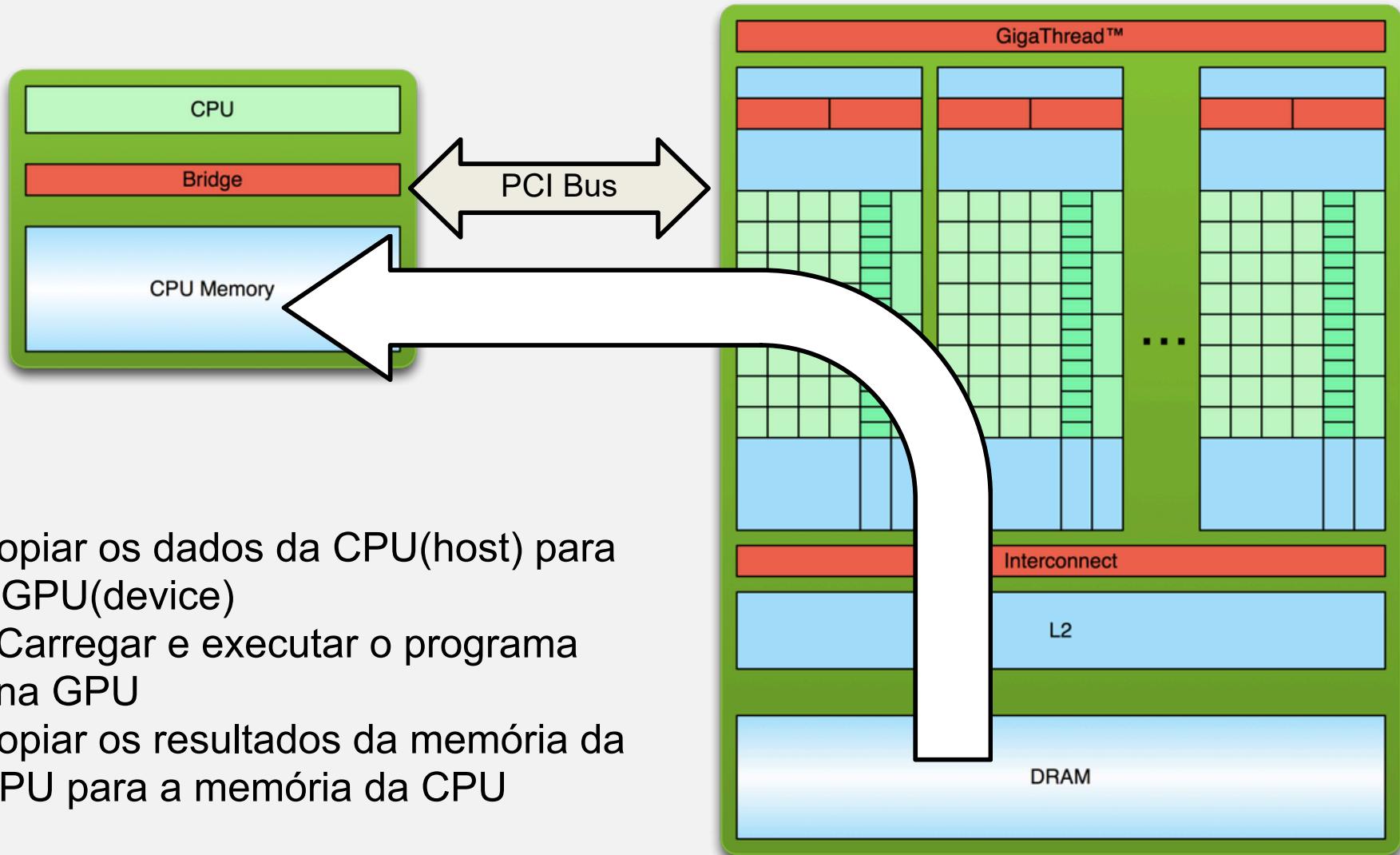


1. Copiar os dados da CPU(host) para a GPU(device)

Sequência de Processos



Sequência de Processos



Hello World!

```
int main(void) {  
    printf("Hello World!\n");  
    return 0;  
}
```

Programa padrão em C que roda na CPU

O compilador NVIDIA(nvcc) pode ser usado para compilar programas mesmo sem utilizar a GPU

Output:

```
$ nvcc  
hello_world.  
cu  
$ a.out  
Hello World!  
$
```

Hello World! com GPU

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

- 2 elementos novos!

Hello World! com GPU

```
__global__ void mykernel(void) {  
}
```

- CUDA C/C++ Palavra chave `__global__` indica uma função que:
 - Roda na GPU(device)
 - é chamada pela CPU(host)
- nvcc o código entre componentes de CPU e GPU
 - Funções do Device (ex. `mykernel()`) processadas pelo compilador NVIDIA
 - Funções do Host (ex. `main()`) processadas pelo compilador padrão
 - `gcc`, `cl.exe`

Hello World! com GPU

```
mykernel<<<1,1>>>();
```

- Triple brackets marcam uma chamada do *host(CPU) para o device(GPU)*
 - Chamado de “kernel launch” (lançamento de kernel)
 - Os parameters (1,1) serão explicados nos próximos slides
- Isso é tudo o necessário para executar uma função na GPU!

Hello World! com GPU

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

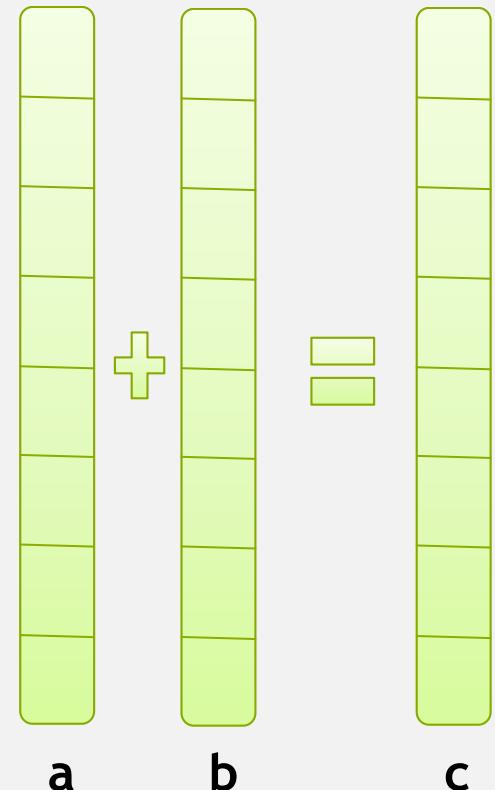
- `mykernel()` não faz nada aqui.

Output:

```
$ nvcc  
hello.cu  
$ a.out  
Hello World!  
$
```

Parallel Programming in CUDA C/C++

- Mas programação com GPU é sobre paralelismo massivo!
- Precisamos de um exemplo melhor...
- Vamos começar somando 2 inteiros para, depois somar vetores!



Adição no Device(GPU)

- Um kernel simples para adicionar 2 inteiros

```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
}
```

- Como explicado, **__global__** é uma palavra chave do CUDA C/C++ que significa
 - **add()** vai ser executada no device(GPU)
 - **add()** vai ser chamada pelo host(CPU)

Adição no Device(GPU)

- Note que usamos ponteiros para as variáveis

```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
}
```

- **add()** roda no device, então **a**, **b** e **c** precisam apontar para a memória do device(GPU)
- Então, precisamos alocar memória na GPU

Gerenciamento de Memória

- A memória do Host e a do device são entidades diferentes
 - **Ponteiros do Device** apontam para a memória da GPU
Podem ser passados do/para o host
Não podem ser desreferenciados/acessados pelo host
 - **Ponteiros do Host** apontam para a memória da CPU
Podem ser passados do/para o Device
Não podem ser desreferenciados/acessados pelo device
- CUDA API para gerenciar a memória do device
 - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
 - Similar aos equivalentes em C: `malloc()`, `free()`, `memcpy()`



Adição no Device(GPU): add()

- Retornando ao nosso kernel **add()**

```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
}
```

- Vamos olhar para a main() agora.

Adição no Device(GPU): main()

```
int main(void) {
    int a, b, c;                                // cópias de a, b, c do Host
    int *d_a, *d_b, *d_c;                        // cópias de a, b, c do Device
    int size = sizeof(int);

    // Alocando espaço para as cópias do device: a, b, c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Valores para o input
    a = 2;
    b = 7;
```

Adição no Device(GPU): main()

```
// Copia os inputs para o device
cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

// Lança o kernel add() na GPU
add<<<1,1>>>(d_a, d_b, d_c);

// Copia os resultados de volta para a CPU host
cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

// Limpeza
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

Rodando em Paralelo

CONCEPTS

Computação Heterogênea

Blocks

Threads

Indexação

Memória compartilhada

`__syncthreads()`

Operações Assíncronas

Handling errors

Managing devices

Falando de Paralelismo

- Computação com GPU é sobre paralelismo massivo
 - Então, como rodamos um código em paralelo no device?



```
add<<< 1, 1 >>>();
```

```
add<<< N, 1 >>>();
```

- Ao invés de executar add() uma vez, execute N vezes em paralelo

Adição de vetor no Device

Com **add()** rodando em paralelo, podemos fazer uma adição vetorial

- Terminologia: cada invocação paralela de **add()** é referida como um bloco/**block**
 - Um conjunto de blocos é referido como um **grid**
 - Cada invocação pode se referir ao índice do seu bloco usando: **blockIdx.x**

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

- Usando **blockIdx.x** como índice no vetor, cada bloco está trabalhando com um índice diferente

Adição de vetor no Device

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

- No device, cada bloco vai executar em paralelo:

Block 0

```
c[0] = a[0] + b[0];
```

Block 1

```
c[1] = a[1] + b[1];
```

Block 2

```
c[2] = a[2] + b[2];
```

Block 3

```
c[3] = a[3] + b[3];
```

Adição de vetor no Device

- Como vai ficar nosso kernel **add()**?

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

- E a main()...?

Adição de vetor no Device: main()

```
#define N 512
int main(void) {
int *a, *b, *c;          // host copies of a, b, c
int *d_a, *d_b, *d_c;    // device copies of a, b, c
int size = N * sizeof(int);

// Alloc space for device copies of a, b, c
cudaMalloc((void **) &d_a, size);
cudaMalloc((void **) &d_b, size);
cudaMalloc((void **) &d_c, size);

// Alloc space for host copies of a, b, c and setup input values
a = (int *)malloc(size); random_ints(a, N);
b = (int *)malloc(size); random_ints(b, N);
c = (int *)malloc(size);
```

Adição de vetor no Device: main()

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU with N blocks
add<<<N,1>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

Revisão (1 of 2)

- Diferença entre *host* e *device*
 - *Host* CPU
 - *Device* GPU
- Usando global para declarar a função como código do device
 - Executa no device
 - Chamada pelo host
- Passar parâmetros do host para o device

Review (2 of 2)

- Gerenciamento básico da memória na GPU
 - `cudaMalloc()`
 - `cudaMemcpy()`
 - `cudaFree()`
- Lançar Kernels paralelos
 - Lauça **N** cópias de **add()** com **add<<<N,1>>>(...);**
 - Usamos **blockIdx.x** para acessar o índice do bloco

Introduzindo Threads

CONCEPTS

Computação Heterogênea

Blocks

Threads

Indexação

Memória compartilhada

`__syncthreads()`

Operações Assíncronas

Lidar com erros

Gerenciar dispositivos

CUDA Threads

- Terminologia: um block pode ser dividido em **threads** paralelas
- Vamos modificar a `add()` para usar threads em paralelo ao invés de *blocos em paralelo*

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

- Usamos **threadIdx.x** ao invés de **blockIdx.x**
- Precisamos mudar a **main()**...

Adição vetorial com Threads: main()

```
#define N 512
int main(void) {
    int *a, *b, *c;           // host copies of a, b, c
    int *d_a, *d_b, *d_c;     // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

Adição vetorial com Threads: main()

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU with N threads
add<<<1,N>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

CONCEPTS

Computação Heterogênea

Blocks

Threads

Indexação

Memória compartilhada

`__syncthreads()`

Operações Assíncronas

Lidar com erros

Gerenciar dispositivos

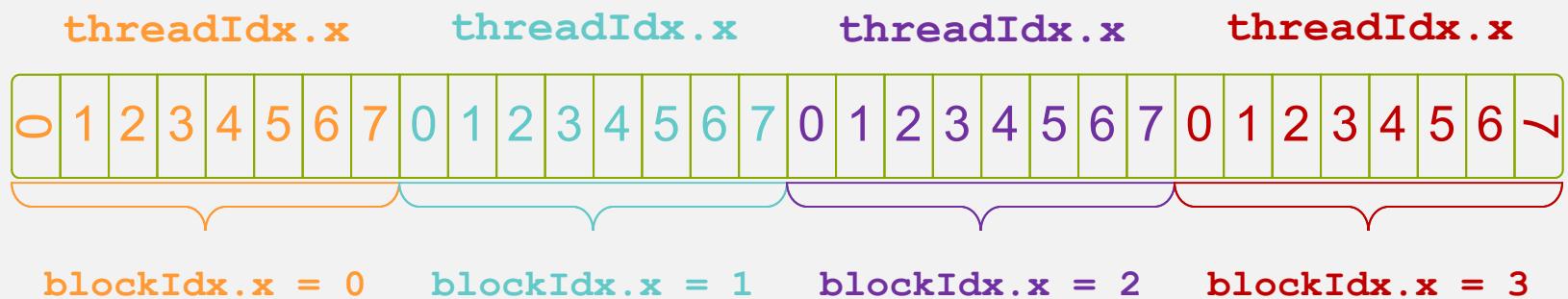
Combinando Threads e Blocks

Combinando Blocks e Threads

- Fizemos a soma vetorial usando:
 - Vários blocos com uma thread em cada
 - Um bloco com várias threads
- Vamos usar agora várias threads e vários blocos!
- Como vai ficar a indexação?

Indexando Arrays com Blocos e Threads

- Não é mais só usar **blockIdx.x** ou **threadIdx.x**
 - Vamos considerar que queremos uma thread para cada elemento do vetor e (8 threads/block)

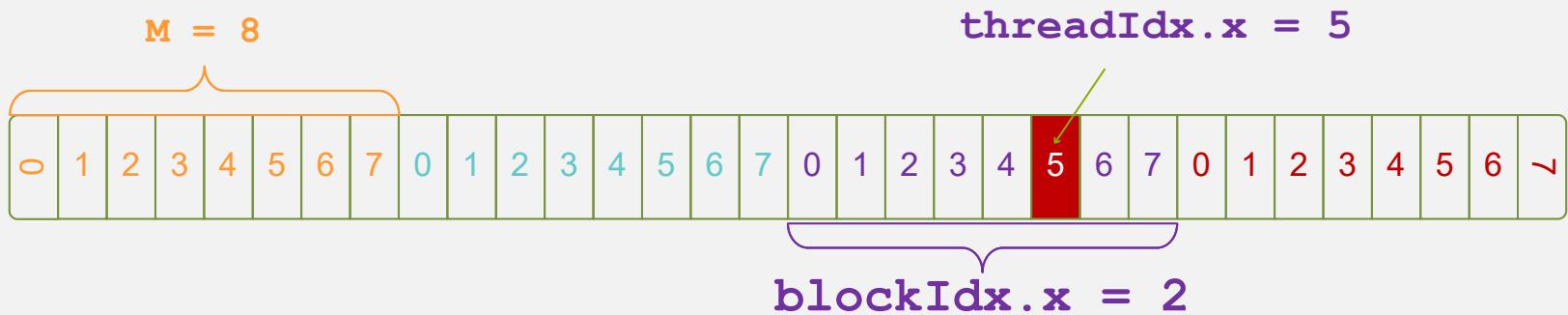


- Com M threads/block podemos achar um índice único para cada thread assim:

```
int index = threadIdx.x + blockIdx.x * M;
```

Indexando Arrays: Exemplo

- Qual thread está operando no elemento vermelho?



```
int index = threadIdx.x + blockIdx.x * M;  
= 5 + 2 * 8;  
= 21;
```

Adição vetorial com Blocos e Threads

- Use a variável **blockDim.x** para threads por block

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

- Versão de add() com threads e blocos paralelos

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

- Como vai ficar a **main()**?

Adição vetorial com Blocos e Threads: main()

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main(void) {
    int *a, *b, *c;           // host copies of a, b, c
    int *d_a, *d_b, *d_c;     // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

Adição vetorial com Blocos e Threads: main()

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

Lidando com vetores de tamanhos arbitrários

- Problemas normalmente não são múltiplos exatos de **blockDim.x**
- Evitando acessar além do fim do array

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

- Mudar o kernel launch:

```
add<<<(N + M-1) / M,M>>>(d_a, d_b, d_c, N);
```

Porque usar Threads?

- Threads parecem desnecessárias
 - Adicionam um nível de complexidade
 - O que ganhamos quando usamos elas?
- Diferente dos blocos, threads possuem mecanismos para:
 - Se comunicar
 - Sincronizarem
- Para entender, precisamos de um exemplo novo...

Revisão

- Lançando kernels paralelos
 - Lance **N** cópias de **add()** com **add<<N/M,M>>**
 - Use **blockIdx.x** para acessar o índice do bloco
 - Use **threadIdx.x** para acessar o índice da thread dentro do bloco
- Alocamos elementos para cada thread com:

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

CONCEPTS

Threads Cooperando

Computação Heterogênea

Blocks

Threads

Indexação

Memória compartilhada

`__syncthreads()`

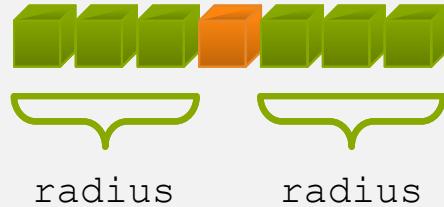
Operações Assíncronas

Lidando com erros

Gerenciando devices

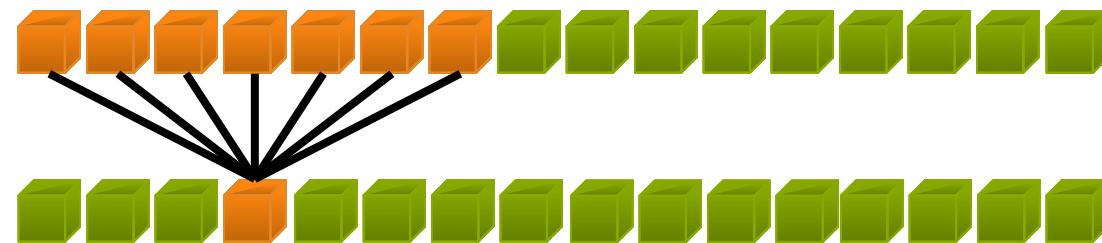
1D Stencil

- Aplicando um stencil 1D em um array de elementos
 - A saída de um elemento é a soma dos elementos em um raio
- Se o raio é 3, então a saída de cada elemento é a soma dos valores de 7 elementos:



Implementação em um bloco

- Cada thread processa o resultado de um elemento
 - blockDim.x elementos por bloco
- Valores dos elementos são lidos várias vezes
 - Com raio = 3, cada elemento é lido 7 vezes.

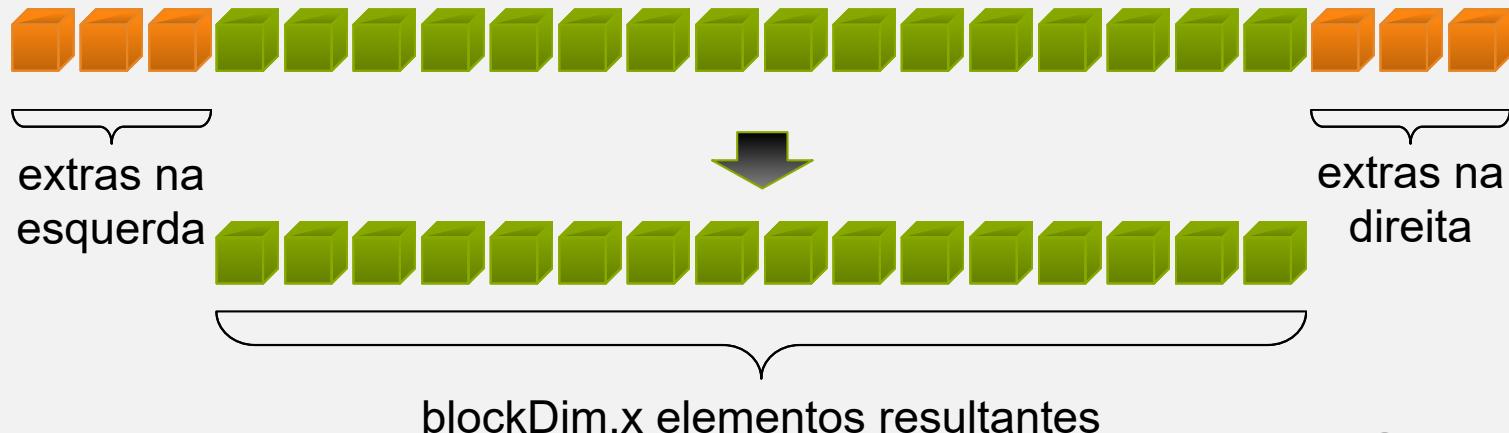


Compartilhando Dados entre Threads

- Terminologia: dentro de um bloco, as threads compartilham dados por **shared memory**
- Extremamente rápido (mais ou menos 5x melhor que a memória global)
- Declare usando **shared**, e é alocada por bloco
- Dado não é visível por threads de outros blocos

Implementando com Shared Memory

- Guardando dados em memória compartilhada
 - Leia (`blockDim.x + 2 * raio`) elementos da memória global e salve na memória compartilhada
 - Calcule o resultado dos `blockDim.x` elementos resultantes
 - Escreva `blockDim.x` elementos na memória global
 - Cada bloco precisa de **elementos extras** em cada extremidade



*Gindex → Global index
Lindex → Local index

Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }
}
```



Stencil Kernel

```
// Apply the stencil
int result = 0;
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];

// Store the result
out[gindex] = result;
}
```

Data Race!

- O exemplo vai dar erro...
 - Supondo que a thread 15 leia dos extras antes da thread 0 ter iniciado ele...

__syncthreads()

- `void __syncthreads();`
- Sincroniza todas as threads dentro do bloco
 - previne falhas RAW / WAR / WAW
- Todas as threads precisam chegar na barreira

Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + radius;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();
}
```

Stencil Kernel

```
// Apply the stencil
int result = 0;
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];

// Store the result
out[gindex] = result;
}
```

Review (1 of 2)

- Lançando threads paralelas
 - Lança N blocos com M threads por bloco usando:
kernel<<<N,M>>>(...);
 - Use **blockIdx.x** para acessar o indice do bloco no grid
 - Use **threadIdx.x** para acessar o indice da thread dentro do bloco

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

Review (2 of 2)

- Use **__shared__** para declarar a variavel/array na memória compartilhada
 - Dados são compartilhados entre threads no mesmo bloco
 - não fica disponível para threads de outros blocos
- Use **__syncthreads()** como uma barreira
 - Use para prevenir data hazards

Managing the Device

CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

`__syncthreads()`

Asynchronous operation

Handling errors

Managing devices

Coordenando Host & Device

- Kernel launches são **assíncronos**
 - retornam para a CPU imediatamente após terminar
- CPU precisa sincronizar antes de consumir o resultado

cudaMemcpy()	Blocks the CPU until the copy is complete Copy begins when all preceding CUDA calls have completed
cudaMemcpyAsync()	Asynchronous, does not block the CPU
cudaDeviceSynchronize()	Blocks the CPU until all preceding CUDA calls have completed

Reportando Erros

- Toda chamada CUDA API retorna um erro (`cudaError_t`)
 - Erro na chamada API
 - ou
 - Erro em uma operação assíncrona anterior (ex. kernel)
- Para consultar o último erro usamos:
`cudaError_t cudaGetLastError(void)`
- Para consultar a descrição do erro:
`char *cudaGetString(cudaError_t)`
`printf("%s\n", cudaGetString(cudaGetLastError()));`

Device Management

- Aplicação pode buscar e selecionar GPUs

```
cudaGetDeviceCount(int *count)  
cudaSetDevice(int device)  
cudaGetDevice(int *device)  
cudaGetDeviceProperties(cudaDeviceProp *prop, int device)
```

- Multiplas threads podem compartilhar um device
- Uma thread pode gerenciar multiplos devices
cudaSetDevice(i) para selecionar o device atual