DESENVOLVIMENTO DE APLICAÇÕES EM CUDA

Profa. Dra. Denise Stringhini
ICT/UNIFESP – São José dos Campos/SP

ERAD-SP 2017
São Carlos/SP

Sumário

- Arquitetura das GPUS
  - Kepler
  - Pascal
  - *Capabilities*
- Introdução à CUDA
  - Computação heterogênea
  - Blocos, threads, indexação
  - Memória compartilhada
  - Gerenciamento de erros e dispositivo

Arquitetura das GPUs

- Os principais fabricantes são a NVIDIA e a AMD.
  - Podem atuar em conjunto com CPUs Intel ou AMD.
- Paralelismo do tipo SIMD.
- Programa principal executa na CPU (*host*) e inicia as *threads* na GPU (*device*).
- Tem sua própria hierarquia de memória e os dados são transferidos através de um barramento *PCI express*.

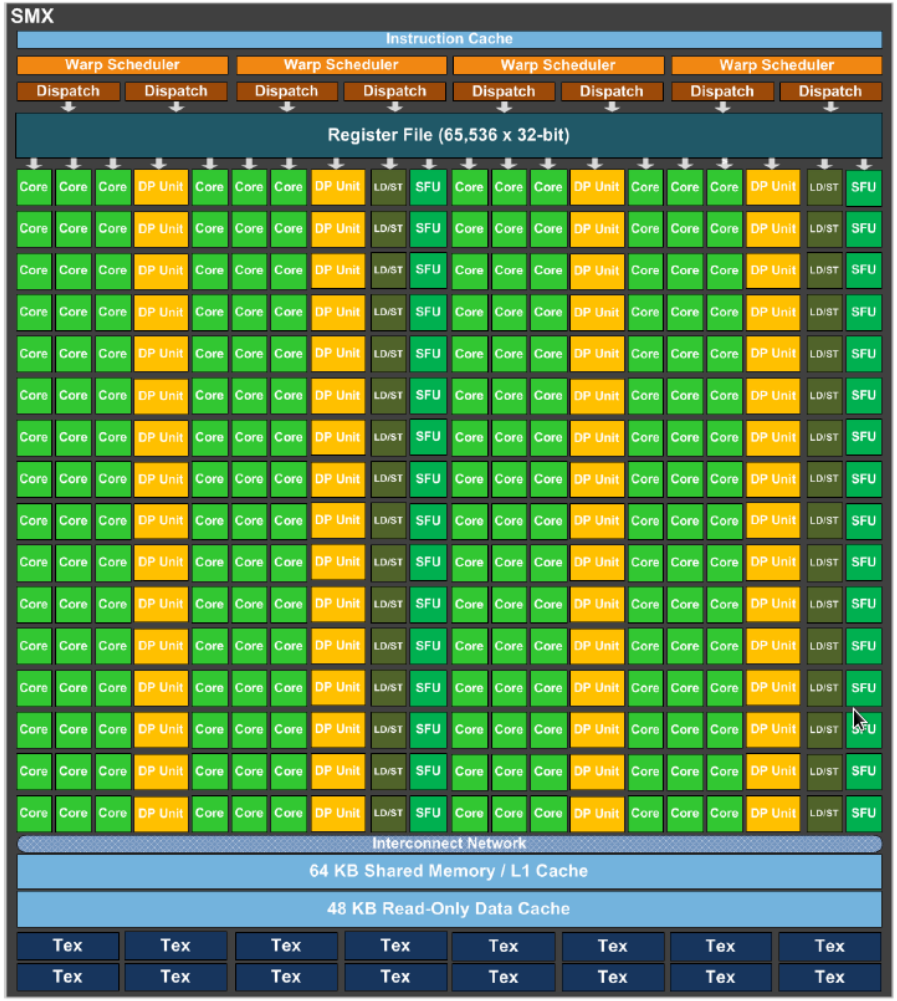
Arquitetura Kepler

- Até 15 SMX com 192 núcleos cada.
  - 2880 cores
- Paralelismo dinâmico.
- Hyper-Q possibilita disparar *kernels* simultaneamente.

**Cluster Halley (ICMC): Nvidia GTX 650 – 1GB; 2 SMX (384 cores)*

Arquitetura Kepler: SMX

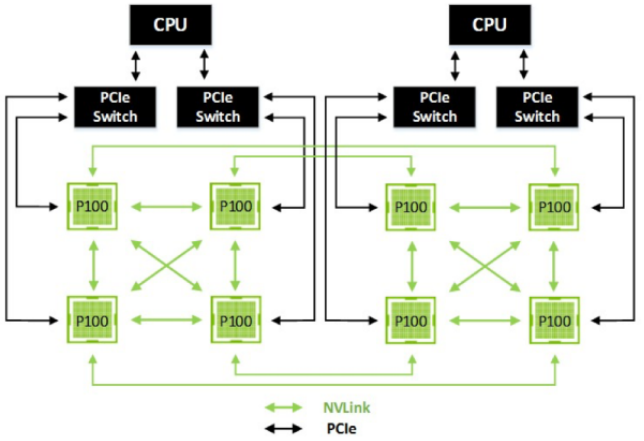
- 192 núcleos de precisão simples
- 64 unidades de precisão dupla
- 32 SFUs
- 32 unidades de *load/ store*
- 4 *warps* concorrentes

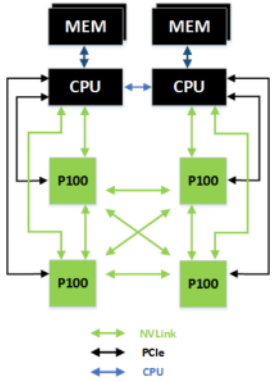


Arquitetura Pascal

Arquitetura Pascal

- Até 60 SMs com 64 cores cada
  - total de 3840 cores
- 16 GB de memória
- Interconexão NVLink
  - comunicação com sistema de memória e outras GPUs
  - 40 GB/s (PCIe 3.0 = 16GB/s)





Arquitetura Pascal: SM

- 64 single-precision (FP32) CUDA Cores
  - menos cores que as anteriores, porém maior número de SMs
  - threads compartilham menos recursos como registradores e cache.
  - maior concorrência

Tesla Products	Tesla K40	Tesla M40	Tesla P100 (NVLink)	Tesla P100 (PCIe)
GPU / Form Factor	Kepler GK110 / PCIe	Maxwell GM200 / PCIe	Pascal GP100 / SXM2	Pascal GP100 / PCIe
SMs	15	24	56	56
TPCs	15	24	28	28
FP32 CUDA Cores / SM	192	128	64	64
FP32 CUDA Cores / GPU	2880	3072	3584	3584
FP64 CUDA Cores / SM	64	4	32	32
FP64 CUDA Cores / GPU	960	96	1792	1792
Base Clock	745 MHz	948 MHz	1328 MHz	1126 MHz
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz	1303 MHz
FP32 GFLOPs[1]	5040	6844	10608	9340
FP64 GFLOPs[1]	1680	213	5304	4670

Tesla Products	Tesla K40	Tesla M40	Tesla P100 (NVLink)	Tesla P100 (PCIe)
GPU / Form Factor	Kepler GK110 / PCIe	Maxwell GM200 / PCIe	Pascal GP100 / SXM2	Pascal GP100 / PCIe
Texture Units	240	192	224	224
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2	3072-bit HBM2 (12GB) 4096-bit HBM2 (16GB)
Memory Bandwidth	288 GB/s	288 GB/s	732 GB/s	549 GB/s (12GB) 732 GB/s (16GB)
Memory Size	Up to 12 GB	Up to 24 GB	16 GB	12 GB or 16 GB
L2 Cache Size	1536 KB	3072 KB	4096 KB	4096 KB
Register File Size / SM	256 KB	256 KB	256 KB	256 KB
Register File Size / GPU	3840 KB	6144 KB	14336 KB	14336 KB
TDP	235 Watts	250 Watts	300 Watts	250 Watts
Transistors	7.1 billion	8 billion	15.3 billion	15.3 billion
GPU Die Size	551 mm^2	601 mm^2	610 mm^2	610 mm^2
Manufacturing Process	28-nm	28-nm	16-nm	16-nm

Compute Capability

- A **compute capability** de um dispositivo descreve sua arquitetura, por exemplo
  - Número de registradores
  - Tamanho das memórias
  - Quantidade de cores por SM
  - Paralelismo máximo
  - etc

Capabilities

GPU	Kepler GK110	Maxwell GM200	Pascal GP100
Compute Capability	3.5	5.2	6.0
Threads / Warp	32	32	32
Max Warps / Multiprocessor	64	64	64
Max Threads / Multiprocessor	2048	2048	2048
Max Thread Blocks / Multiprocessor	16	32	32
Max 32-bit Registers / SM	65536	65536	65536
Max Registers / Block	65536	32768	65536
Max Registers / Thread	255	255	255
Max Thread Block Size	1024	1024	1024
CUDA Cores / SM	192	128	64
Shared Memory Size / SM Configurations (bytes)	16K/32K/48K	96K	64K

Prática

- Compilar e executar **deviceQuery**
  - amostra fornecida pela NVIDIA em *samples*
  - apresenta detalhes sobre o(s) dispositivo(s) instalado(s)

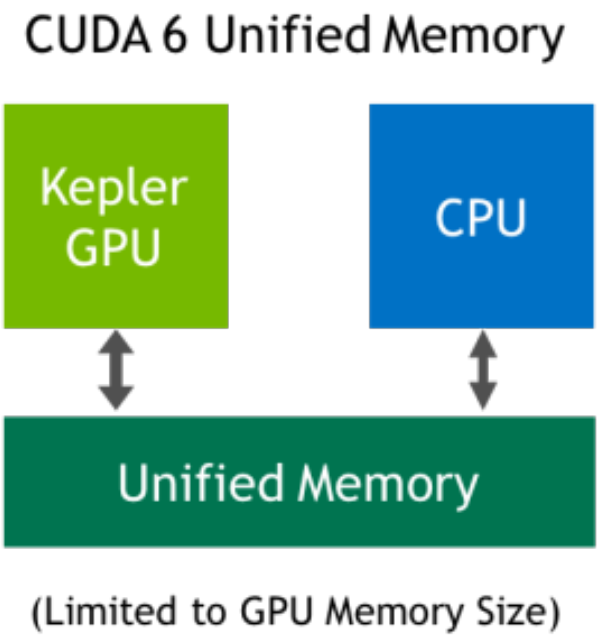
- Pasta: 0-deviceQuery

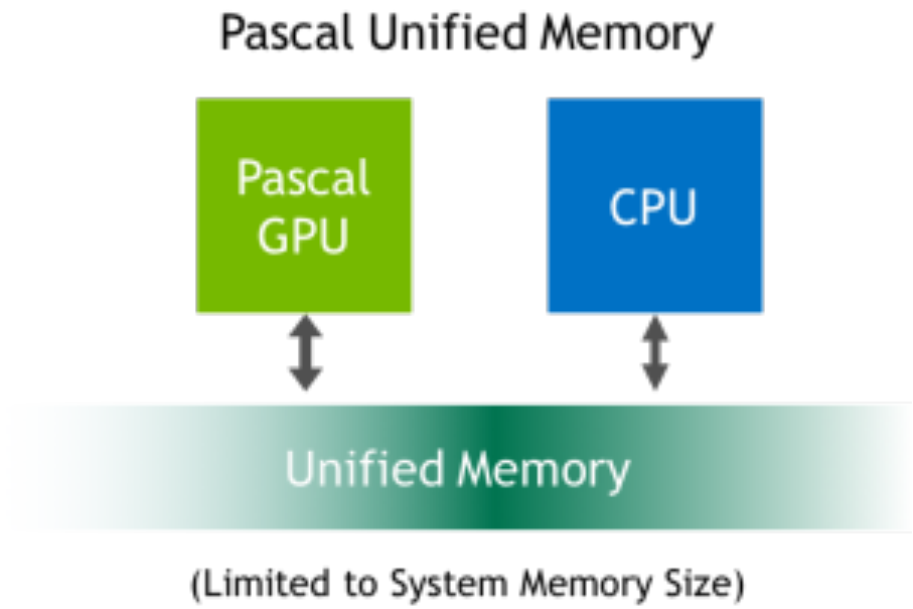
Memória Unificada

- Disponível a partir de CUDA 6

- A memória pode ser gerenciada automaticamente a partir de um ponteiro alocado na CPU
  - transferências automáticas (veremos mais tarde)

Memória Unificada



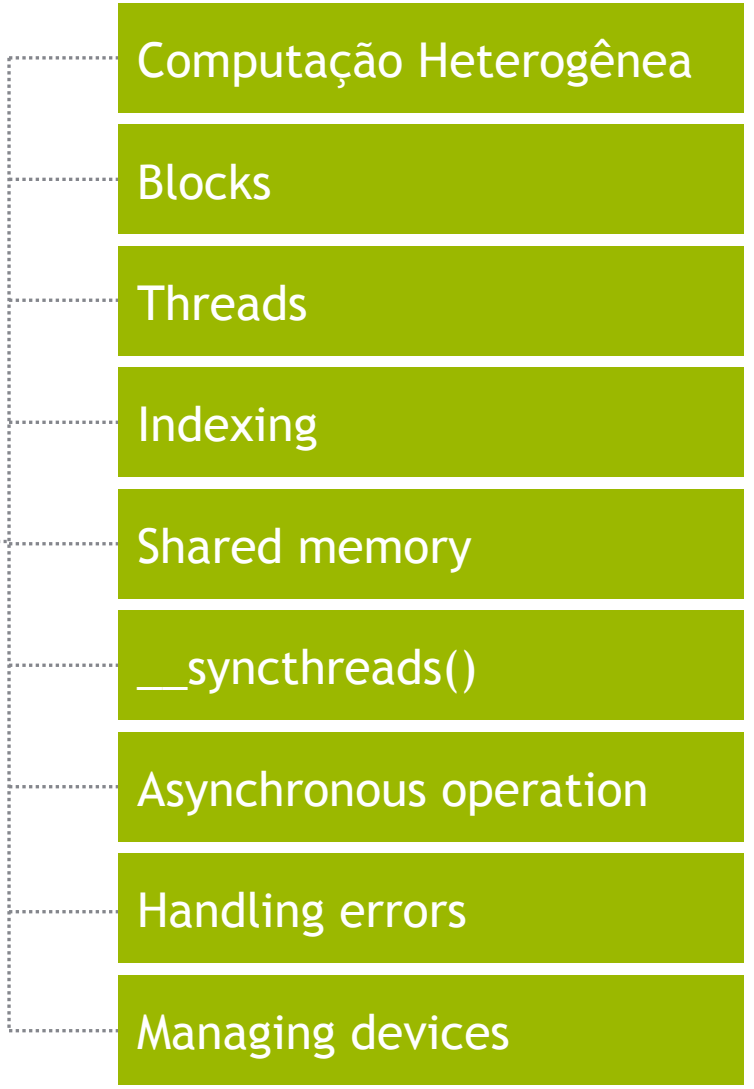


Arquitetura CUDA

- Possibilita paralelismo na GPU para computação de propósito geral

- CUDA C/C++
  - Baseada no padrão C/C++
  - Pequeno conjunto de extensões para permitir programação heterogênea.
  - API para gerenciamento de dispositivos, memória, etc.

Conceitos


Computação Heterogênea
Blocks
Threads
Indexing
Shared memory
__syncthreads()
Asynchronous operation
Handling errors
Managing devices

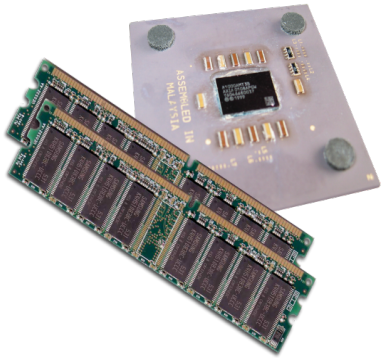
HELLO WORLD!

Conceitos

- Computação Heterogênea
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

Computação Heterogênea

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



Host

Device

Computação Heterogênea

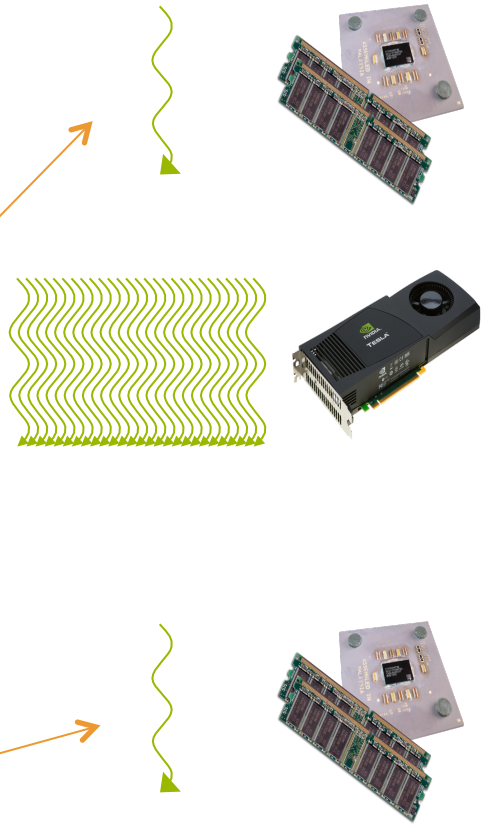
```
#include <iostream>
#include <algorithm>

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 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();

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

        // Store the result
        out[gindex] = 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((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_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;
}
```

parallel fn

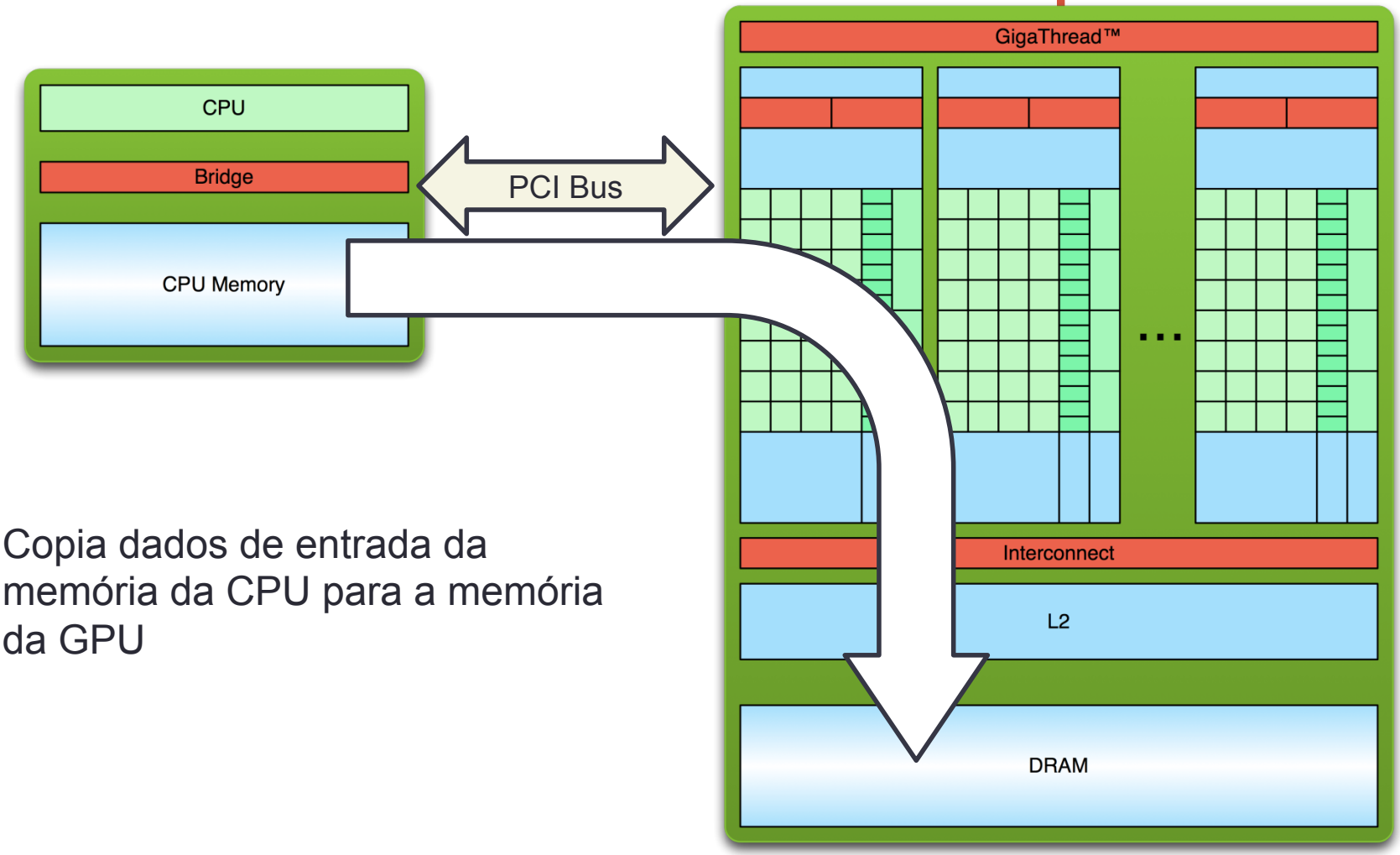
serial code

parallel code

serial code

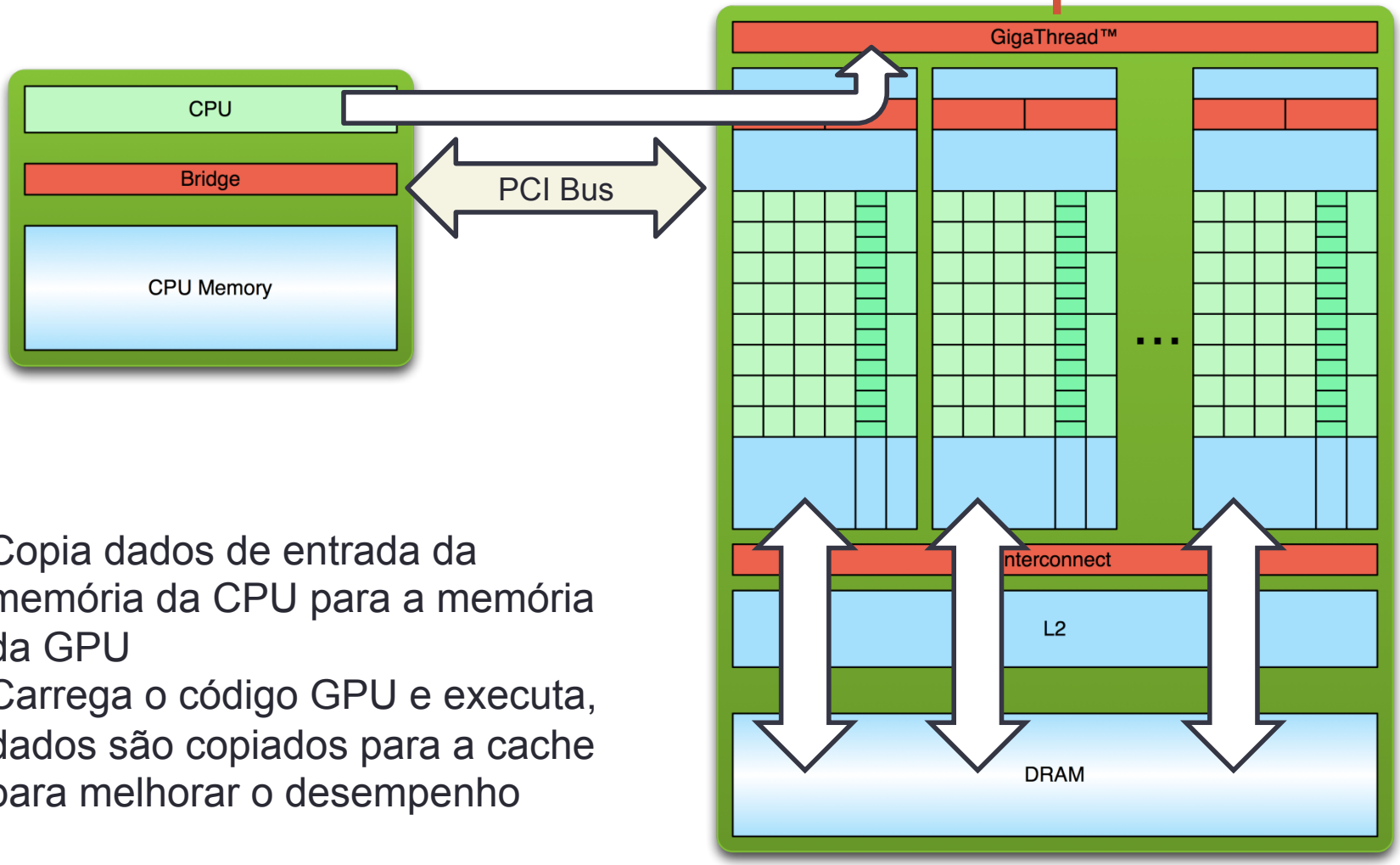


Fluxo de Processamento Simples



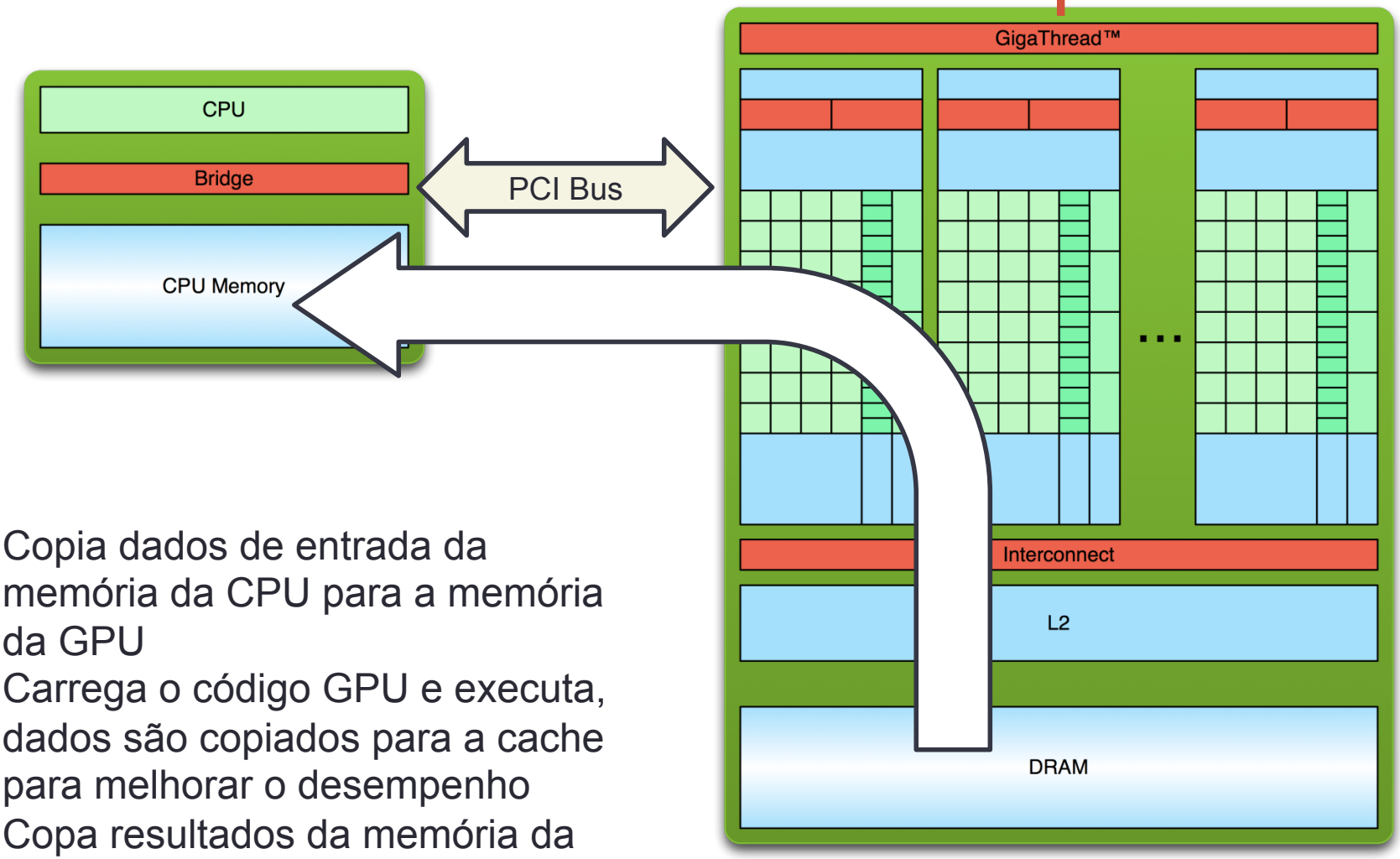
1. Copia dados de entrada da memória da CPU para a memória da GPU

Fluxo de Processamento Simples



1. Copia dados de entrada da memória da CPU para a memória da GPU
2. Carrega o código GPU e executa, dados são copiados para a cache para melhorar o desempenho

Fluxo de Processamento Simples



1. Copia dados de entrada da memória da CPU para a memória da GPU
2. Carrega o código GPU e executa, dados são copiados para a cache para melhorar o desempenho
3. Copa resultados da memória da GPU para a memória da CPU

Hello World!

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

- Código C padrão que roda no host
- Compilador NVIDIA (**nvcc**) pode ser usado para compilar programas sem código para o dispositivo (*device)*

```
$ nvcc
hello_world.cu

$ ./a.out
Hello World!
$
```

Hello World!

```
__global__ void mykernel(void) {
}

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

- Dois novos elementos sintáticos…

Hello World!

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

- `__global__` indica uma função que:
  - Executa no device
  - É chamada a partir do código do host

- `nvcc` separa o código fonte em componentes host e device
  - Funções device (e.g. **`mykernel()`**) processadas pelo compilador NVIDIA
  - Funções host (e.g. **`main()`**) processadas pelo compilador padrão do host
    - **`gcc`**, **`cl.exe`**

Hello World!

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

- Sintaxe que indica uma chamada do código do *host* para o código do *device*
  - Também chamado de “kernel launch”
  - Parâmetros (1,1) serão vistos adiante

- Isto é tudo o que é necessário para executar um código na GPU!

Hello World!

```
__global__ void mykernel(void){
}

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

```
$ nvcc
hello.cu

$ a.out
Hello World!
$
```

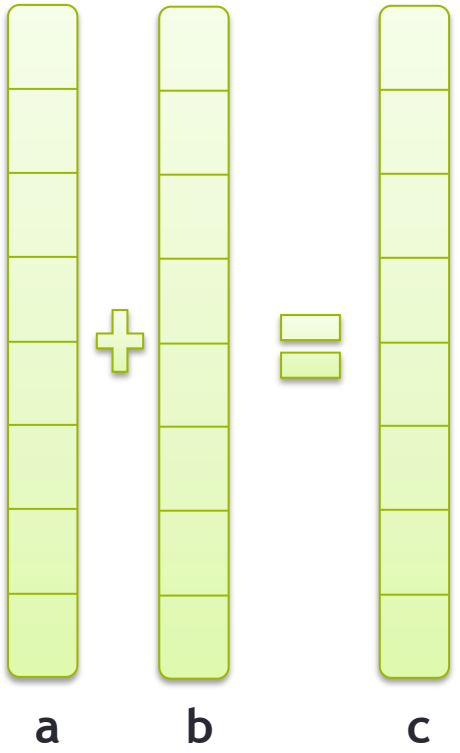
- `mykernel()` ainda não faz nada ☹!

Prática

- Pasta: 1-HelloWorld
- Exercícios em: README_ERADSP2017.txt

Programação Paralela em CUDA C/C++

- Precisamos de um exemplo com paralelismo!
- Começaremos somando dois inteiros e evoluindo para a soma de vetores.



Adição no Dispositivo

- Um kernel simples para somar dois números:

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

- `__global__` é uma palavra reservada em CUDA C/C++ que significa:
  - `add()` será executada no device
  - `add()` será chamada do host

Adição no Dispositivo

- Note que ponteiros são usados como variáveis

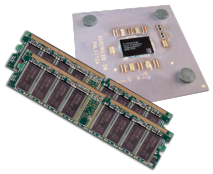
```
__global__ void add(int *a, int *b, int *c) {
        *c = *a + *b;
}
```

- `add()` executa no device, portanto `a`, `b` e `c` devem apontar para a memória do device

- Precisamos alocar memória na GPU

Gerenciamento de Memória

- A memórias do host e do device são entidades separadas
  - *Device* : ponteiros para a memória do dispositivo
    - Devem ser passados de/para a memória do host
    - Não podem ser “desreferenciados”no host
  - *Host :* ponteiros para a memória da CPU
    - Podem ser passados de/para o device
    - Não podem ser “desreferenciados”no device



- CUDA API para gerenciar a memória do device
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - Similar equivalentes em C `malloc()`, `free()`, `memcpy()`

Adição no Dispositivo: `add()`

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

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

- Vamos olhar a main()…

Adição no Dispositivo: `main()`

```
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 = sizeof(int);

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

        // Setup input values
        a = 2;
        b = 7;
```

Adição no Dispositivo: `main()`

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

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

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

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

Memória Unificada

- Facilita a alocação de memória oferecendo um espaço de endereçamento único acessível a partir da CPU e da(s) GPU(s).
- A função **cudaMallocManaged()** retorna um ponteiro que pode ser usado a partir do código do host (CPU) ou do dispositivo (GPU).

Memória Unificada: exemplo

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

            // Allocate space for device copies of
a, b, c
            cudaMallocManaged(&d_a, size);
            cudaMallocManaged (&d_b, size);
            cudaMallocManaged (&d_c, size);

            // Setup input values
            *d_a = 2;
            *d_b = 7;
```

Memória Unificada: exemplo

```
// Launch add() kernel on GPU
        add<<<1,1>>>(d_a, d_b, d_c);
        cudaDeviceSynchronize();

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

EXECUÇÃO PARALELA

Conceitos

- Computação Heterogênea
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

Acrescentar Paralelismo

- Computação na GPU deve ter paralelismo massivo
  - Como executar o código em paralelo no dispositivo?

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



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

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

Adição de Vetores no Dispositivo

- Com `add()` executando em paralelo, podemos realizar adição de vetores

- Terminologia: cada invocação paralela de `add()` é chamada de bloco (*block*)
  - O conjunto de blocos é chamado de grade (grid)
  - Cada invocação pode referenciar seu índice de 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` para indexar o array, cada bloco usará um índice diferente

Adição de Vetores no Dispositivo

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

- No device, cada bloco pode executar em paralelo:

Adição de Vetores no Dispositivo: `add()`

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

- Vamos olhar a main()…

Adição de Vetores no Dispositivo: `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 Vetores no Dispositivo: `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;
}
```

Profile

- A ferramenta **nvprof** é oferecida juntamente com o CUDA Toolkit.
- É usada na linha de comando retorna dados sobre a execução.

nvprof ./a.out

- Tutorial:
  - https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-nvprof-your-handy-universal-gpu-profiler/

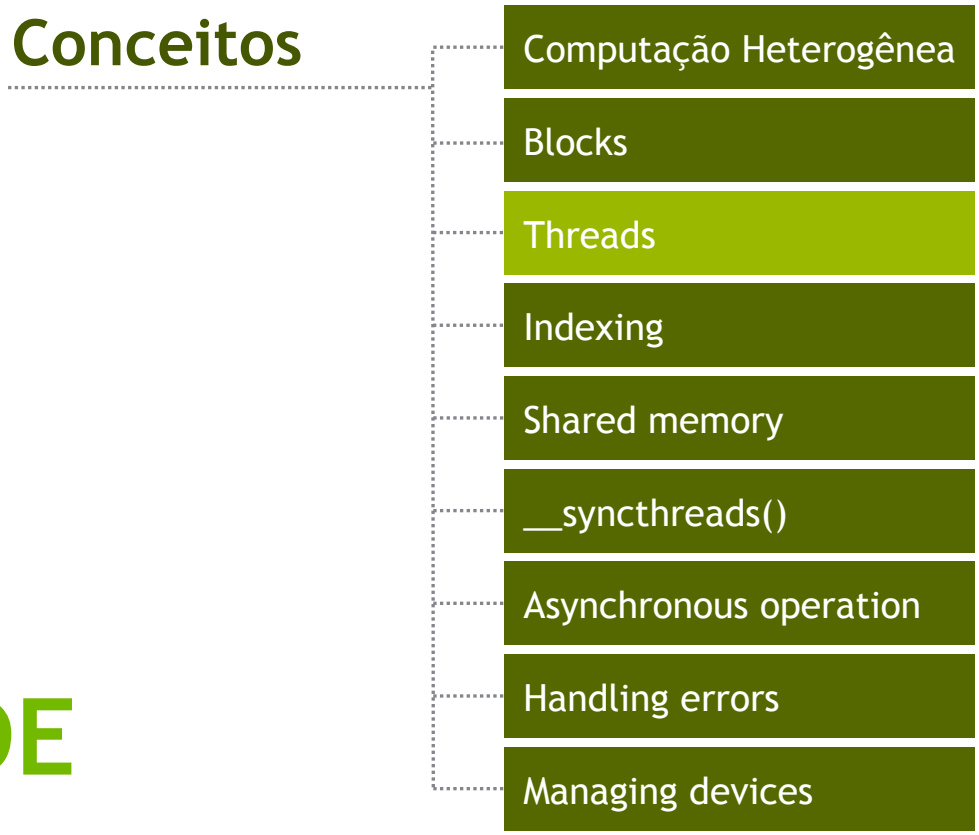
Prática

- Alterar o código: executar com **N blocos**
- Compilar e executar

- Profile
  - guardar tempo para comparação com versões posteriores

- Pasta: **2-VectorAdd**

INTRODUÇÃO DE THREADS



CUDA Threads

- Terminologia: um bloco pode ser dividido em threads paralelas
- Vamos alterar `add()` para usar *threads* paralelas ao invés de blocos

```
__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 de uma alteração na **`main()`**...

Adição de Vetores 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 de Vetores 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;
}
```

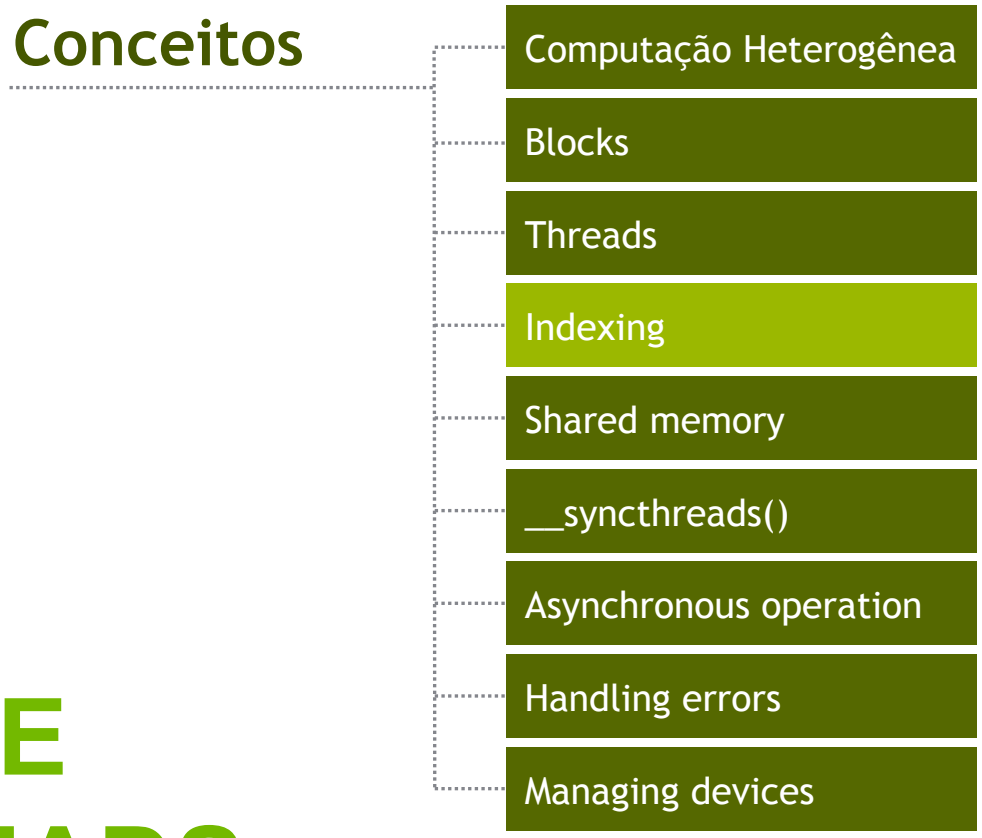
Prática

- Alterar o código: executar com **N threads**
- Compilar e executar

- Profile
  - guardar tempos para comparação entre as versões

- Pasta: **2-VectorAdd**

COMBINAÇÃO DE BLOCOS E THREADS



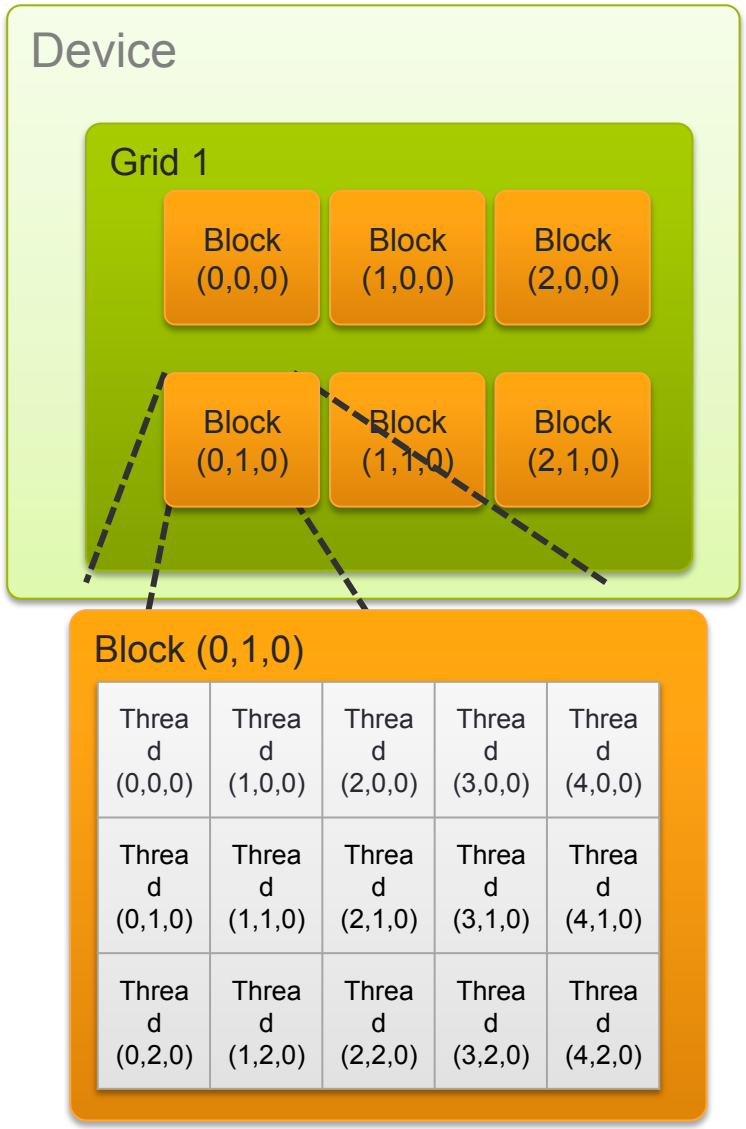
Combinação de Blocos e Threads

- Vimos adição de vetores em paralelo usando:
  - Vários blocos com uma thread cada
  - Um bloco com várias threads
- Vamos adaptar a adição de vetores combinando blocos e threads
- Como? Voltaremos a isso…
- Primeiro vamos ver a indexação de dados.

IDs e Dimensões

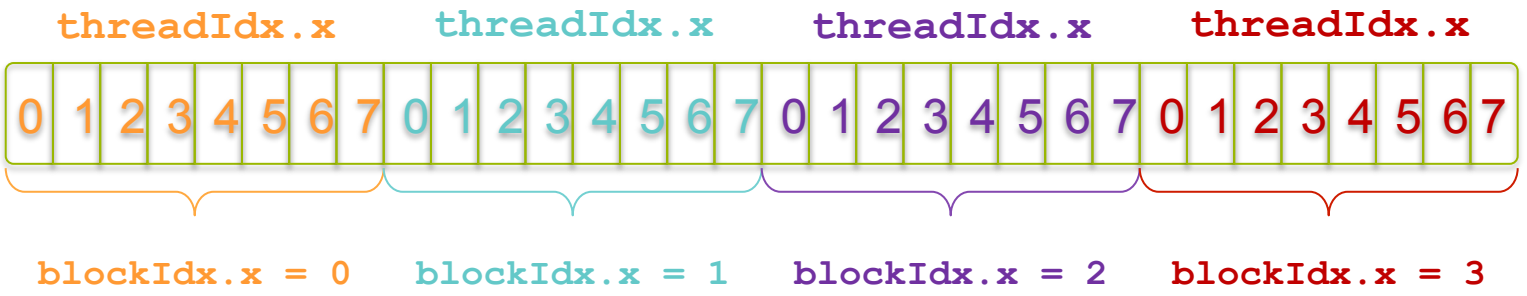
- Um kernel é lançado comum uma grade (*grid*) de blocos de threads
  - `blockIdx` e `threadIdx` são 3D
  - Inicialmente será mostrada somente uma dimensão (`x`)

- Variáveis embutidas:
  - `threadIdx`
  - `blockIdx`
  - `blockDim`
  - `gridDim`



Indexação de Arrays com Blocos e Threads

- Considere indexar um array com um elemento por thread (8 threads/block)

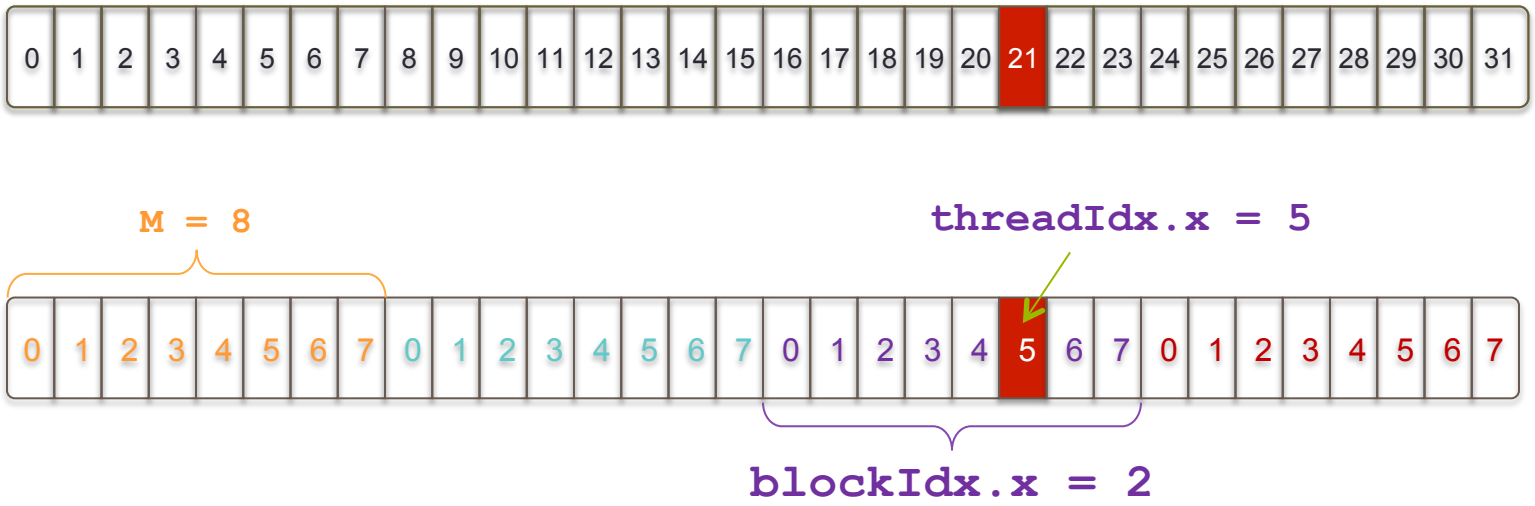


- Com M threads/block um índice único para cada thread é dado por:

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

Indexação de Arrays : Exemplo

- Qual thread vai operar sobre o elemento vermelho?




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

Adição de Vetores com Blocos e Threads

- Usar a variável embutida `blockDim.x` para threads por bloco

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

- Versão combinada de `add()` para usar 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];
}
```

- Que mudanças precisam ser feitas na `main()`?

Adição 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 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;
}
```

Tamanhos de Vetores Arbitrários

- Problemas típicos dificilmente são múltiplos de `blockDim.x`
- Deve-se evitar acessar posições além do final dos arrays

```
__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];
}
```

- Atualizar a chamada do kernel:

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

Por que usar Threads?

- Threads parecem desnecessárias (temos os blocos…)
  - Elas adicionam um nível de complexidade
  - Qual é o ganho?

- Diferente dos blocos paralelos, threads oferecem mecanismos para:
  - Comunicação
  - Sincronização

O que acontece na GPU?

- Cada instância (BLOCO) do kernel em execução executa num SM.
- Se o número de instâncias é maior do que o de SMs, mais de uma executará em cada SM
  - se houver suficiente quantidade de registradores e memória compartilhada
  - as outras instâncias executarão depois
- Todas as threads de uma instância acessam a memória compartilhada local
  - mas não enxergam as outras instâncias, mesmo que estejam no mesmo SM
- Não há garantias da ordem de execução

Prática

- Alterar o código: executar com **N blocos e M threads**
- Compilar e executar
- Profile
  - guardar tempos para comparação entre as versões
- Pasta: **2-VectorAdd**

THREADS COOPERATIVAS

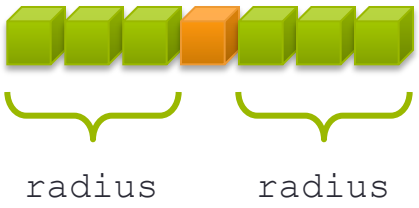
CONCEITOS

- Computação Heterogênea
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

1D Stencil

- Considere aplicar um stencil de 1D a um array 1D
  - Cada elemento de saída é a soma de seus vizinhos dentro de um raio

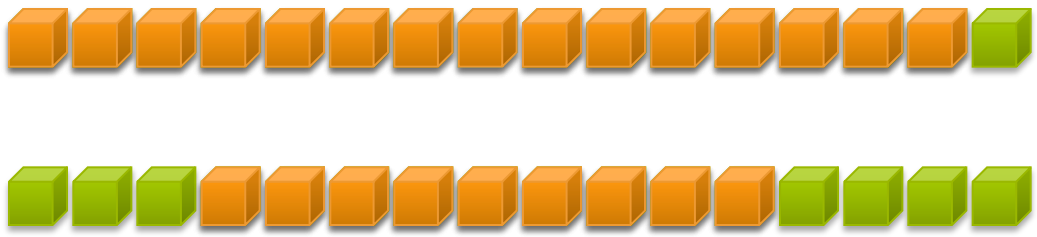
- Se o raio é 3, então cada elemento de saída será a soma de 7 elementos de entrada:



Implementação dentro de um Bloco

- Cada thread processa um elemento de saída
  - blockDim.x elementos por bloco

- Elementos de entrada são lidos várias vezes
  - Com raio 3, cada elemento de entrada é lido 7 vezes



Prática

- Completar versão simples do código (indexação, lançamento do kernel).

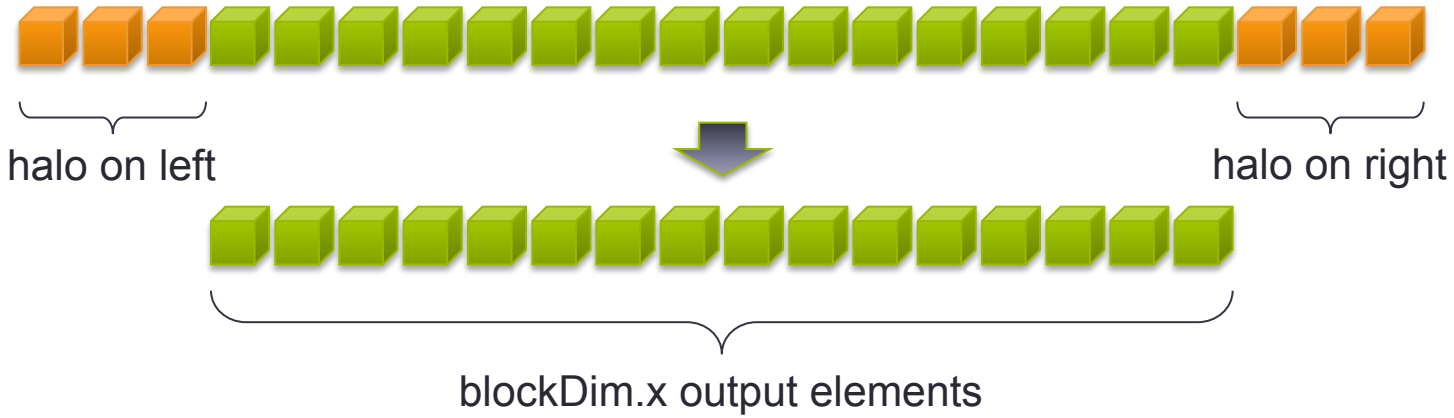
- Profile: guardar tempo de execução

Compartilhamento de Dados entre Threads

- Dentro de um bloco, threads podem compartilhar dados via memória compartilhada (*shared memory*).
- É uma memória localizada no chip, extremamente rápida, gerenciada pelo usuário
- Declarar usando `__shared__`, alocada por bloco
- Dados não são visíveis a threads de outros blocos

Implementação com Memória Compartilhada

- Trazer os dados para a *shared memory*
  - Ler (blockDim.x + 2 * radius) elementos de entrada a partir da memória global para a *shared memory*
  - Computar blockDim.x elementos de saída
  - Escrever blockDim.x elementos de saída na memória global
  - Cada bloco precisa de uma margem (*halo*) de raio elementos em cada extremidade

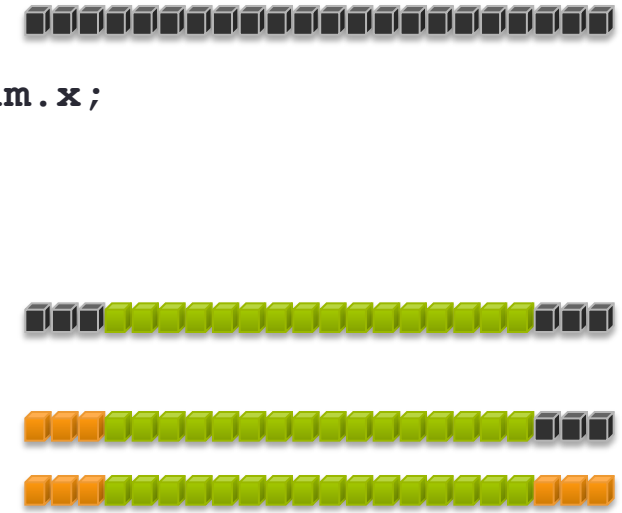


Prática

- Completar versão 2:
  - acrescentar código para trazer os dados para a memória compartilhada
  - processar localmente
  - copiar o resultado para a memória global

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;
}
```

Prática

- Compilar e executar versão com ***shared memory***.
  - Executar várias vezes até observar resultados diferentes (!)
  - Por que isto acontece?

Data Race!

- O exemplo de stencil não vai funcionar…

- Suponha que a thread 15 leia o halo antes que a thread 0 o tenha buscado…

Store at temp[18]

```
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
  temp[lindex - RADIUS = in[gindex - RADIUS];
  temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}

int result = 0;
result += temp[lindex + 1];
```

Skipped, threadIdx > RADIUS

Load from temp[19]

__syncthreads()

- `void __syncthreads();`

- Sincroniza todas as threads dentro de um bloco
  - Usado para prevenir erros RAW / WAR / WAW

- Todas as threads devem chegar na barreira
  - Em código condicional, a condição deve ser uniforme em todo o bloco

Prática

- Acrescente `__syncthreads()` no código
  - Onde?

- Profile: compare o tempo entre a versão sem memória compartilhada e esta última.

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;
}
```

Quantidade de Blocos Ativos

- Cada bloco requer uma certa quantidade de recursos:
  - threads
  - registradores (regs por thread x num threads)
  - shared memory (estática x dinâmica)
- Juntos, estes parâmetros determinam a quantidade de blocos que podem executar simultaneamente em cada SM
  - o máximo depende da *capability*

GERENCIAMENTO DO DISPOSITIVO

CONCEITOS

- Computação Heterogênea
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

Coordenação de Host & Device

- Lançamentos de kernel são assíncronos
  - Controle retorna à CPU imediatamente
- CPU precisa sincronizar antes de consumir os resultados

`cudaMemcpy()`	Bloqueia a CPU até que a cópia esteja completa Cópia inicia quando todas as chamadas CUDA precedentes estejam completas
`cudaMemcpyAsync()`	Assíncrona, não bloqueia a CPU
`cudaDeviceSynchronize()`	Bloqueia a CPU até que todas as chamadas CUDA precedentes estejam completas

Comunicação de Erros

- Todas as chamadas da CUDA API retornam um código de erro (`cudaError_t`)
  - Erro na própria chamada da API

    OU
  - Erro numa operação assíncrona anterior (e.g. kernel)

- Para obter o código do último erro:

```
cudaError_t cudaGetLastError(void)
```

- Para obter a string que descreve o erro:

```
char *cudaGetErrorString(cudaError_t)
  printf("%s\n", cudaGetErrorString(cudaGetLastError()));
```

Gerenciamento do Dispositivo

- A aplicação pode verificar e selecionar GPUs

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

- Múltiplas threads de CPU podem compartilhar um device

- Uma única thread de CPU pode gerenciar múltiplos devices

  `cudaSetDevice(i)` to select current device

  `cudaMemcpy(…)` for peer-to-peer copies †

† requires OS and device support

Materiais utilizados

- Training Material and Code Samples (NVIDIA):
  - developer.nvidia.com/cuda-education
- Inside Pascal: NVIDIA's Newest Computing Platform (Parallel ForAll):
  - devblogs.nvidia.com/parallelforall/inside-pascal/
- Course on CUDA Programming on NVIDIA GPUs (Oxford):
  - people.maths.ox.ac.uk/gilesm/cuda/

Bibliografia

- Programming Massively Parallel Processors: a Hands-on Approach - David Kirk and Wen-mei Hwu
- CUDA by Example: An Introduction to General-Purpose GPU Programming - Jason Sanders and Edward Kandrot