8 – GPU & Cuda

Quellen:
http://courses.cs.washington.edu/courses/cse471/13sp/lectures/GPUsStudents.pdf

http://on-demand.gputechconf.com/gtc-express/2011/presentations/GTC_Express_Sarah_Tariq_June2011.pdf

Goals

Data Parallelism

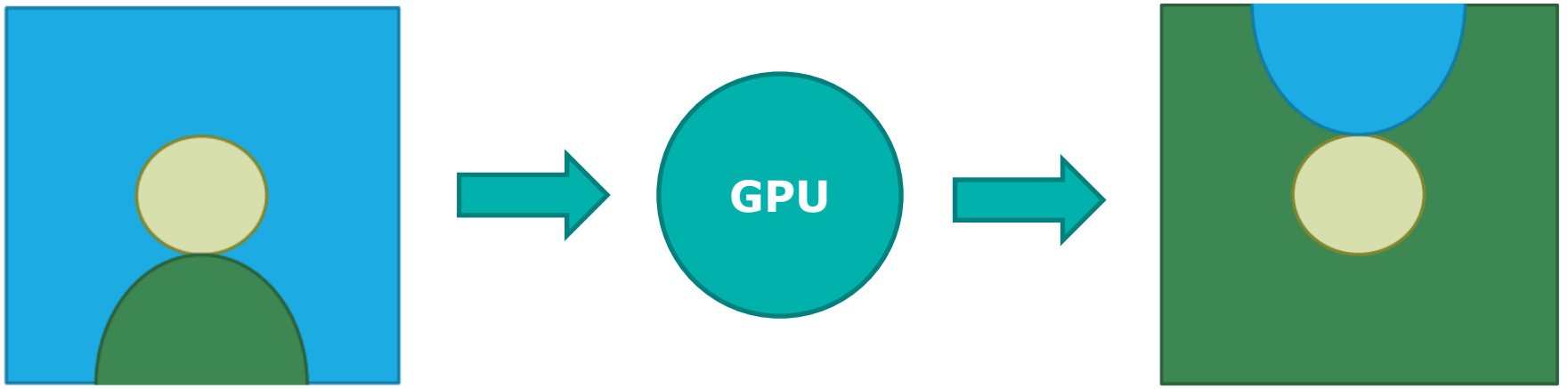
- Workload characteristics

Execution Models / CPU Architectures

- MIMD, SIMD, SIMT

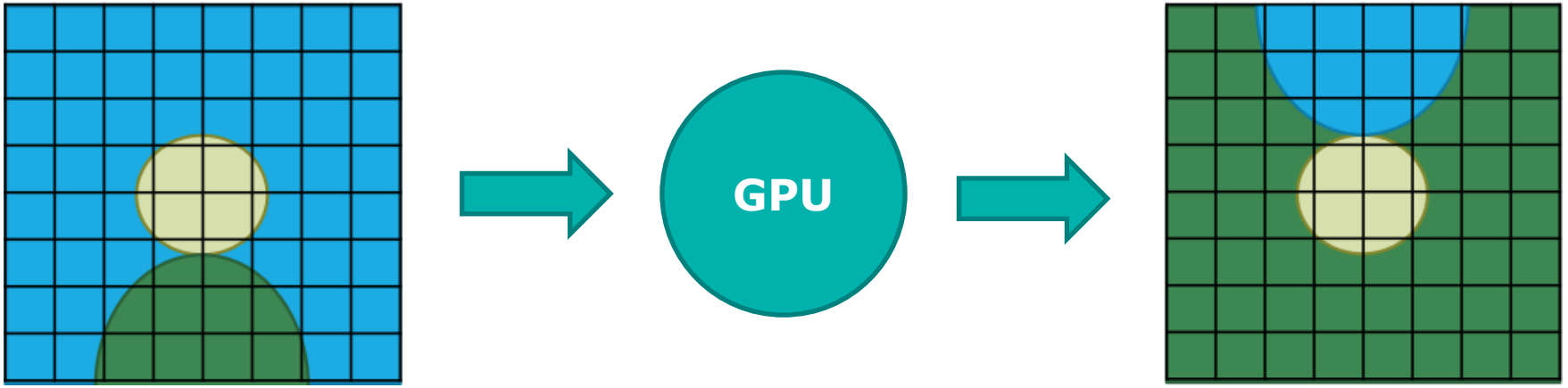
Graphic Workloads

Streaming computation



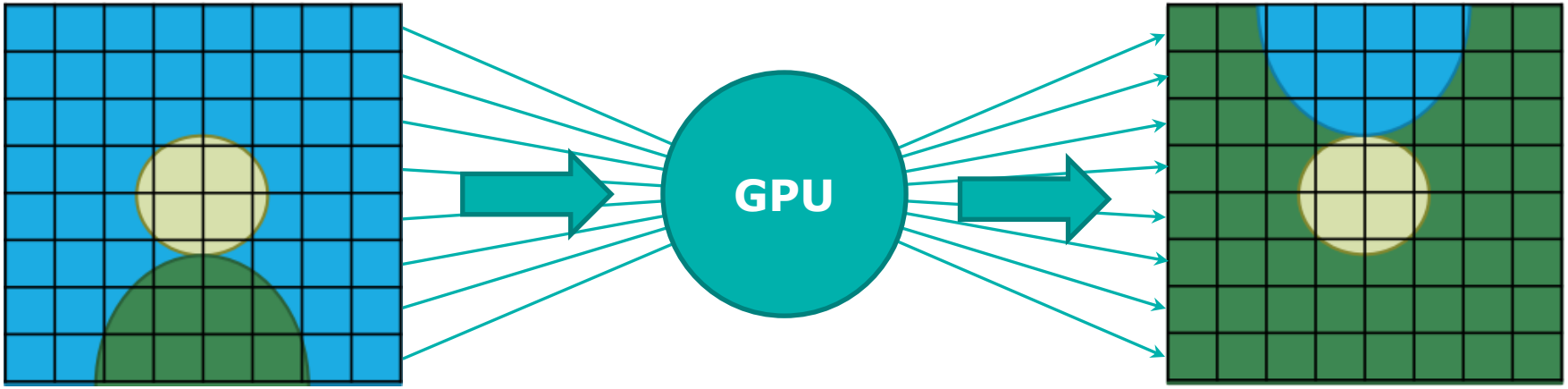
Graphic Workloads

Streaming computation on pixels



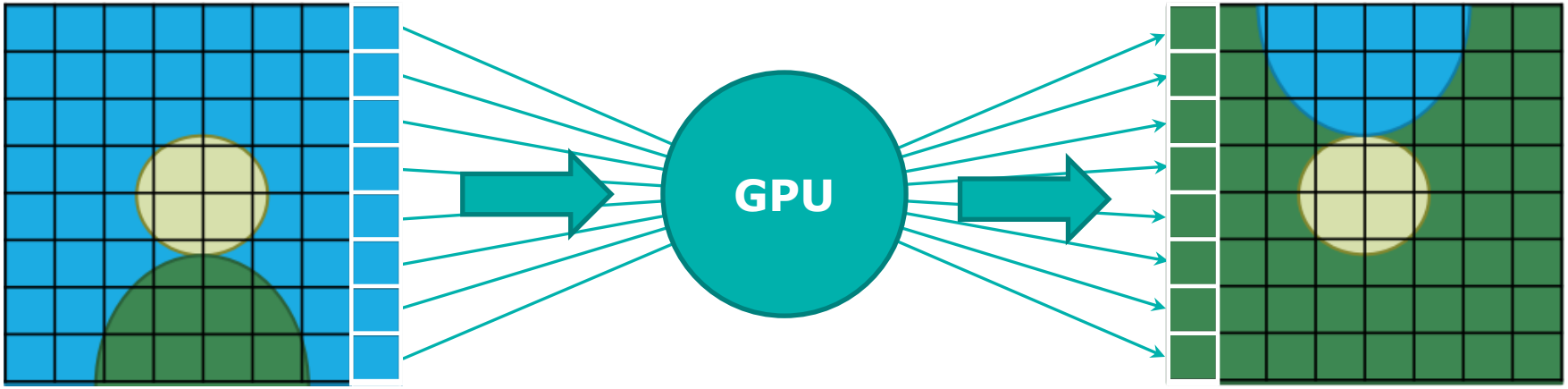
Graphic Workloads

Identical, streaming computation on pixels

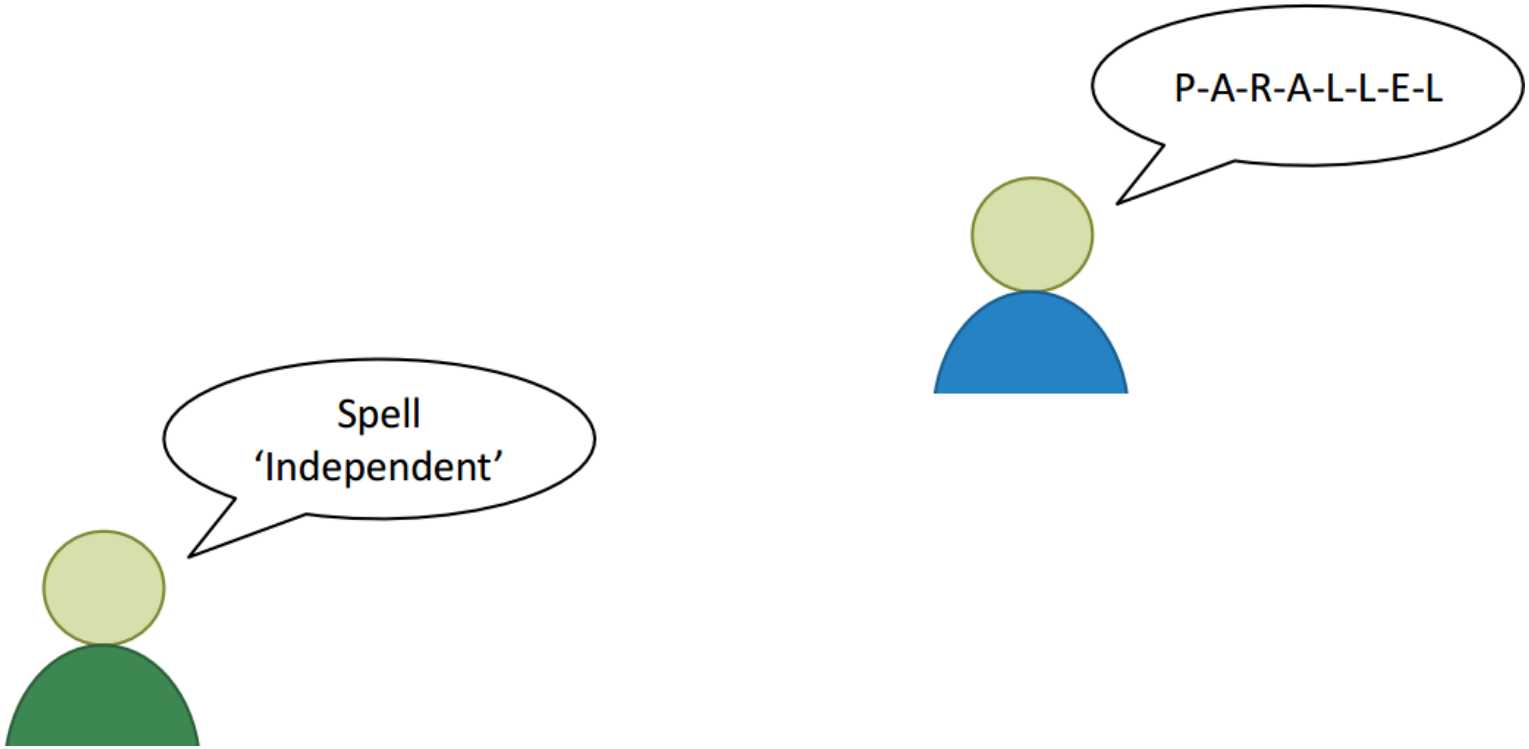


Graphic Workloads

Identical, independent, streaming **computation** on pixels

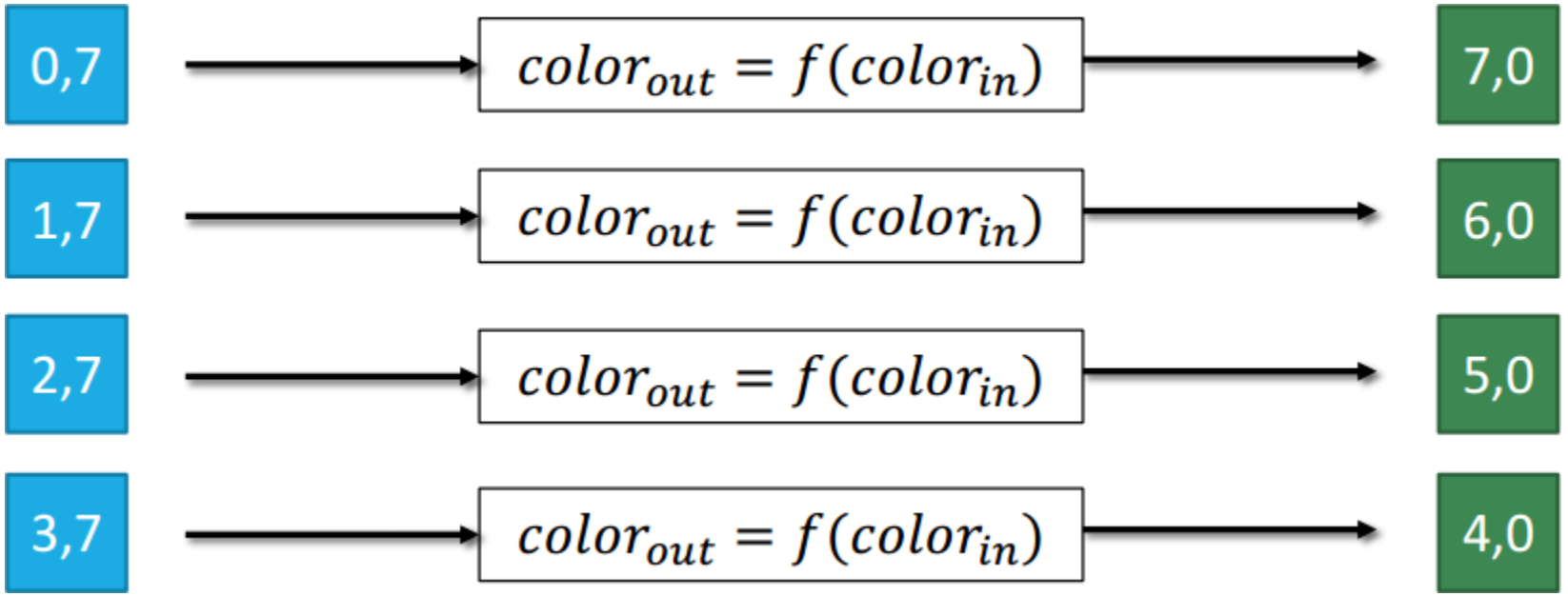


Architecture Spelling Bee



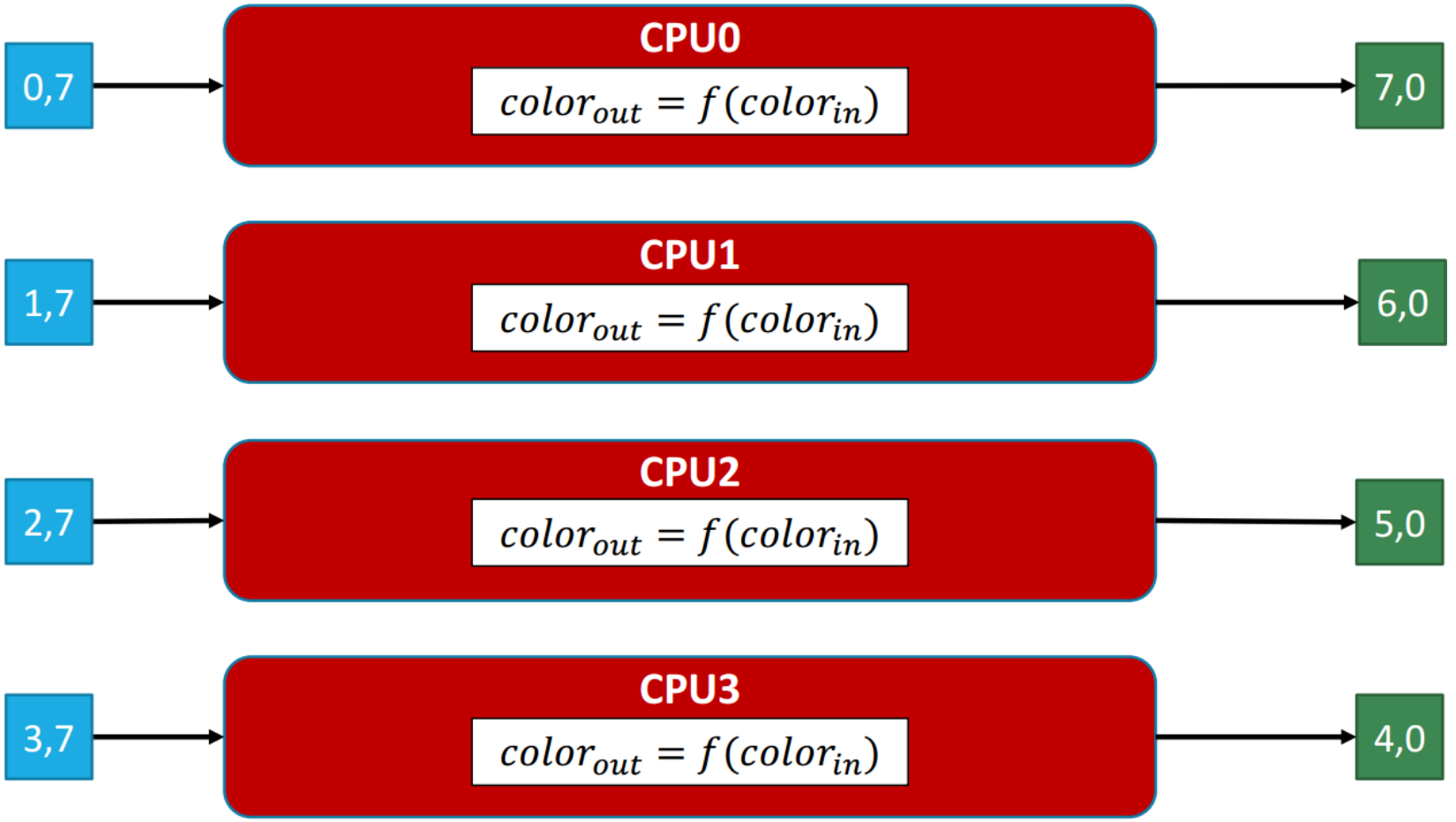
Generalize: Data Parallel Workloads

Identical, independent **computation** on multiple data inputs



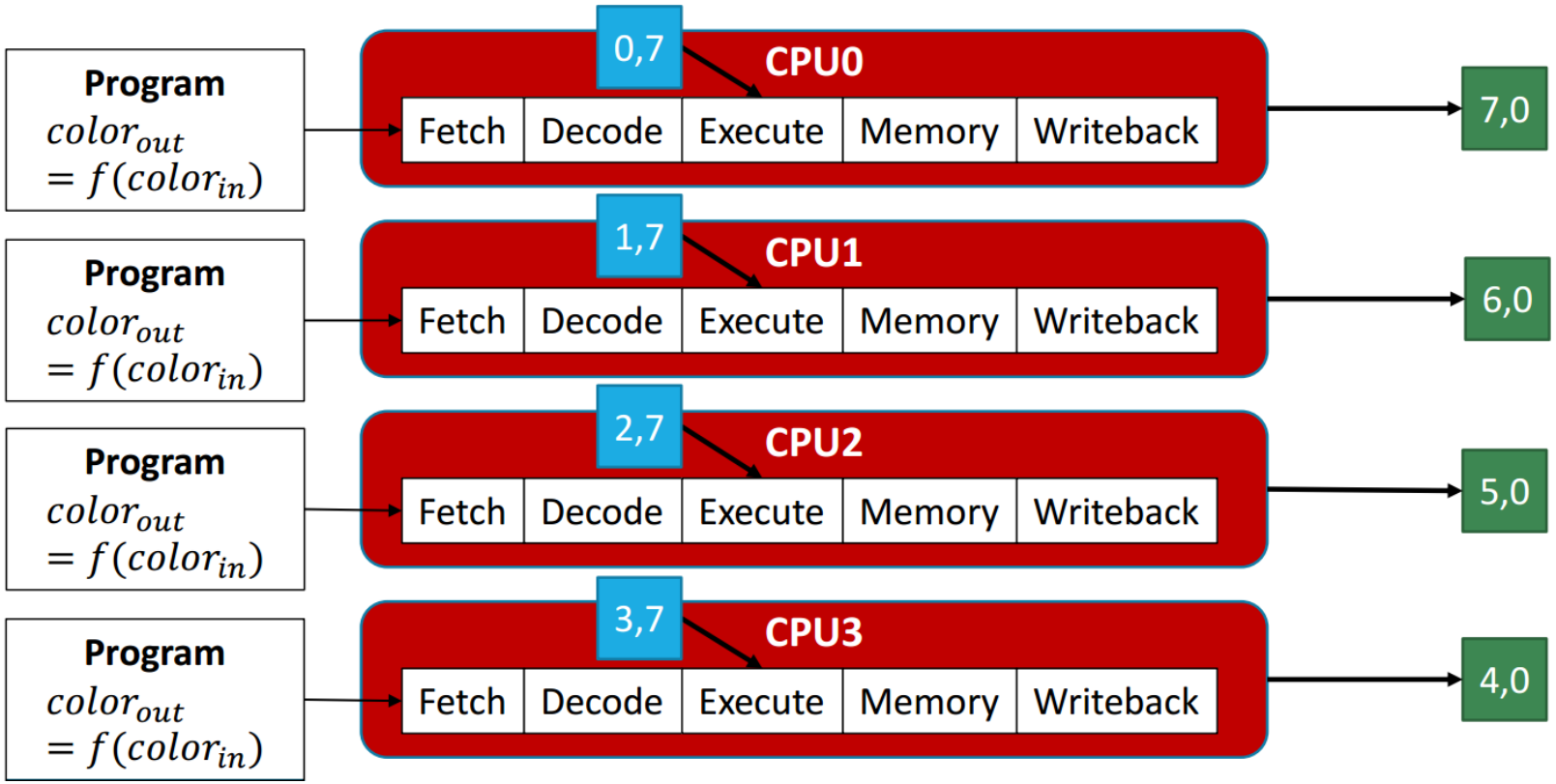
Naive Approach

Split independent work over multiple processors



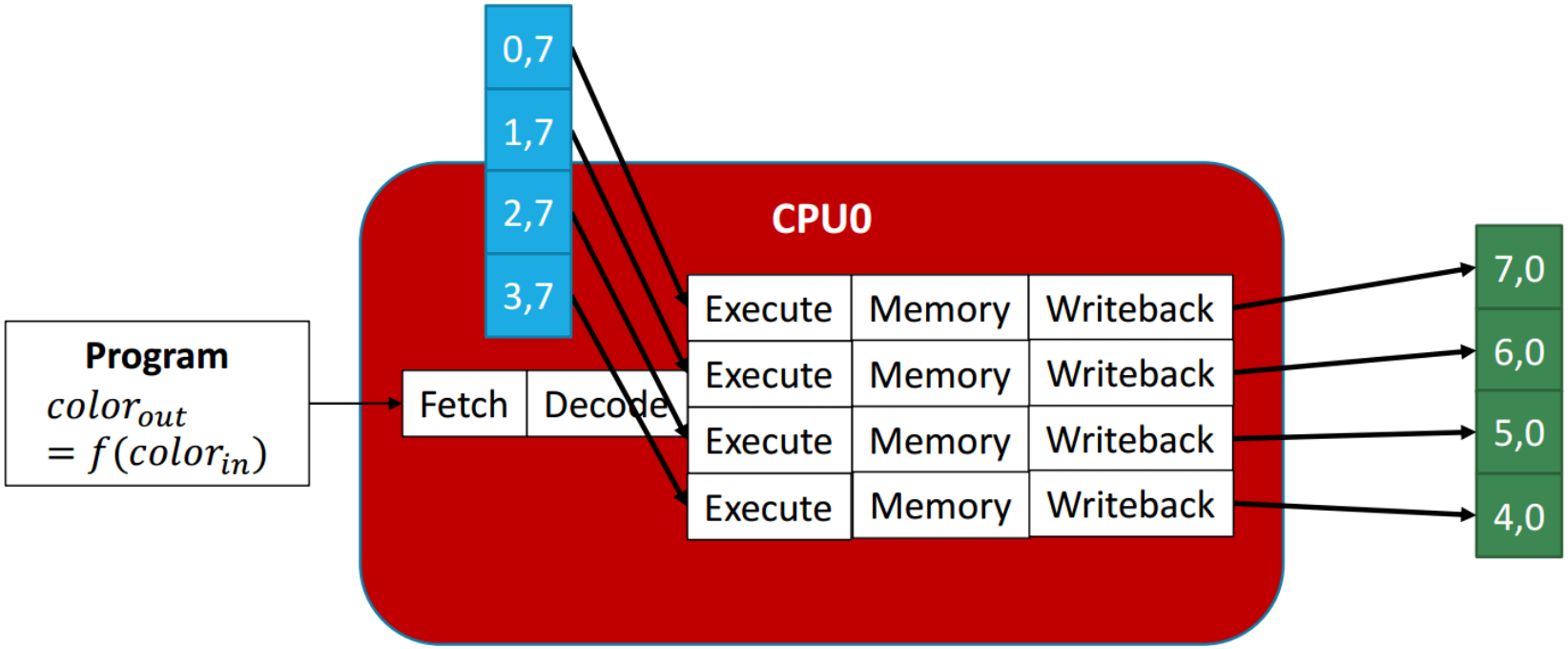
A MIMD Approach

Split independent work over multiple processors



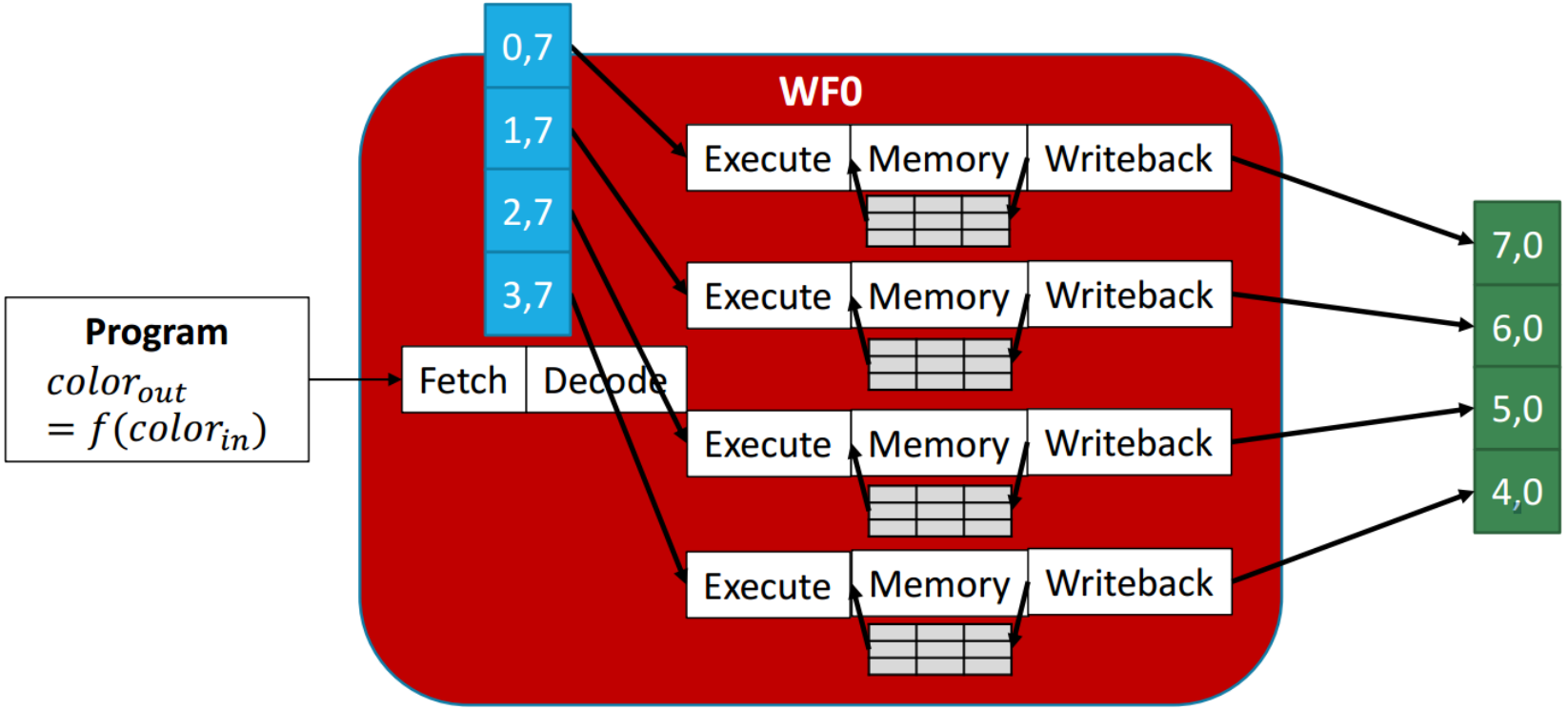
A SIMD Approach

Split identical, independent work over multiple execution units



A SIMT Approach

Split identical, independent work over multiple lockstep threads



Data Parallel Execution Models

MIMD/SPMD

Multiple **independent** threads

SIMD/Vector

One thread with wide execution datapath

SIMT

Multiple **lockstep** threads

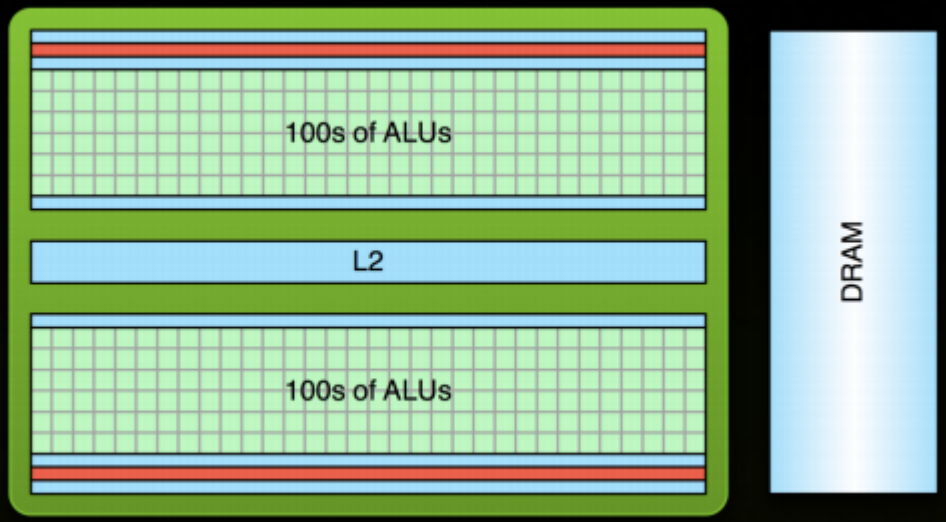
Execution Model Comparison

	MIMD/SPMD	SIMD/Vector	SIMT
			
Example Architecture	Multicore CPUs	x86 SSE/AVX	GPUs
Pros	More general: supports TLP	Can mix sequential & parallel code	Easier to program Gather/Scatter operations
Cons	Inefficient for data parallelism	Gather/Scatter can be awkward	Divergence kills performance

CPU Computing

GPU: Graphics Processing unit

- Traditionally used for real-time rendering
- High computational density (100s of ALUs) and memory bandwidth (100+ GBs/s)
- Troughput processor: 1000s of concurrent threads to hide latency



What is CUDA?

CUDA Architecture

- Expose GPU computing for general purpose
- Retain performance

CUDA C/C++

- Based on industry-standard C/C++
- Small set of extensions to enable heterogeneous programming
- Straightforward APIs to manage devices, memory etc.

This session introduces CUDA C/C++

Introduction to CUDA C/C++

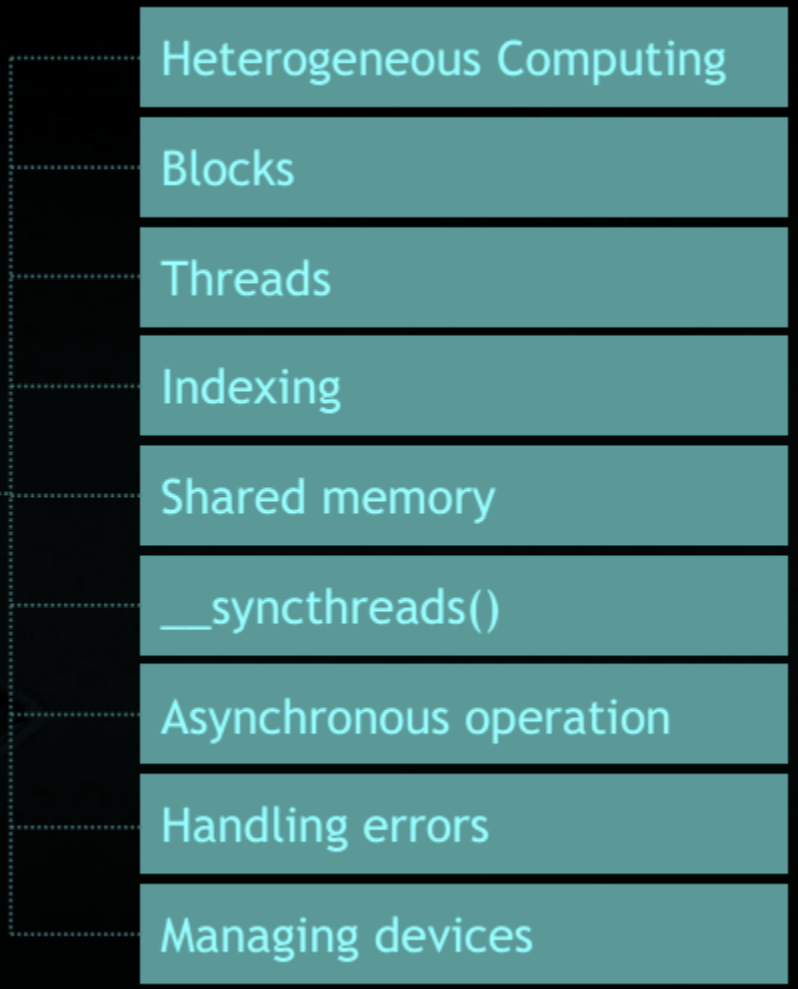
What will you learn in this session?

- Start from „Hello World!“
- Write and launch CUDA C/C++ kernels
- Manage GPU memory

Prerequisites

- You (probably) need experience with C or C++
- You don‘t need GPU experience
- You don‘t need parallel programming experience
- You don‘t need graphics experience

CONCEPTS



Heterogeneous Computing

Terminology

- Host The CPU and its memory (host memory)
- Device The GPU and its memory (device memory)

Heterogeneous Computing

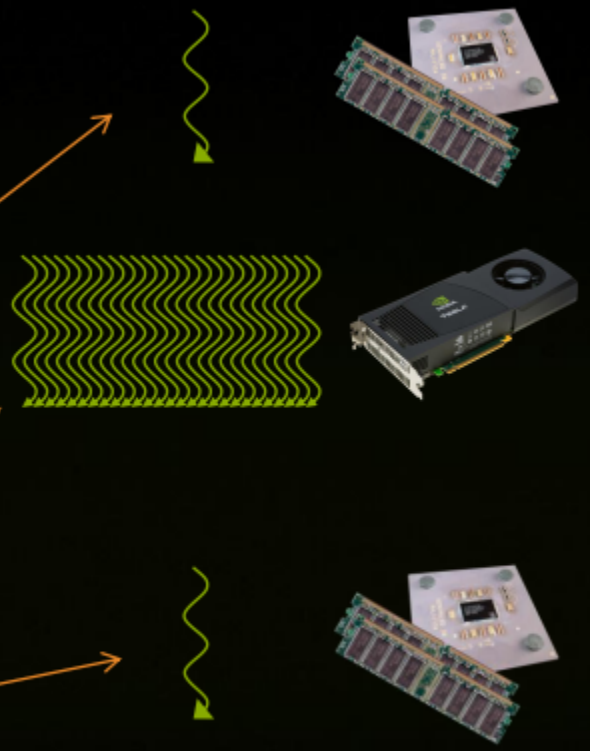
```
#include <iostream>
#include <algorithm>

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



Hello World!

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

- Standard C code that runs on the host

Hello World! with Device Code

```
__global__ void mykernel(void) {
}

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

Hello World! with Device Code

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

- CUDA C/C++ keyword `__global__` indicates a function that:
  - > Runs on the device
  - > Is called from host code

Hello World! with Device Code

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

- Triple angle brackets mark a call from host code to device code
  - > Also called a “kernel launch”

- That‘s all that is required to excecute a function on the GPU!

Parallel Programming in CUDA C/C++

- But wait… GPU computing is about massive parallelism!
- We need a more interesting example…
- We‘ll start by adding two integers and build up to vector addition

Addition on the device

- A simple kernel to add two integers

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

- As before `__global__` is a CUDA C/C++ keyword meaning
  - > `add()` will be executed on the device
  - > `add()` will be called from the host

Addition on the device

- Note that we use pointers for the variables

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

- `add()` runs on the device, so `a`, `b` and `c` must point to device memory

- We need to allocate memory on the GPU

Memory Management

- Host and device memory are separate entities
  - > Device pointers point to GPU memory
    May be passed to/from host code
    May not be dereferenced in host code
  - > Host pointers point to CPU memory
    May be passed to/from device code
    May not be dereferenced in device code
- Simple CUDA API for handling device memory
  - > `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - > Similar to the C equivalents `malloc()`, `free()`, `memcpy()`

Addition on the device: 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;
```

Addition on the device: 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;
}
```

Moving to Parallel

- GPU computing is about massive parallelism
  > So how do we run code in parallel on the device?

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

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

- Instead of executing `add()` once, execute N times in parallel

Vector Addition on the Device

- With `add()` running in parallel we can do vector addition

- Terminology: each parallel invocation of `add()` is referred to as a block
  - > The set of blocks is referred to as a grid
  - > Each invocation can refer to ist block index using `blockIdx.x`

- By using `blockIdx.x` to index into the array, each block handles a different index

Vector Addition on the Device

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

- On the device, each block can execute in paralle

Vector Addition on the 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
        a = (int *)malloc(size); random_ints(a, N);
        b = (int *)malloc(size); random_ints(b, N);
        c = (int *)malloc(size);
```

Vector Addition on the 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;
}
```

Review

- Difference between host and device
  - > Host CPU
  - > Device GPU
- Using `__global__` to declare a function as device code
  - > Executes on the device and is called from the host
- Passing parameters from host code to a device function
- Basic device memory management
  - > `cudaMalloc(), cudaMemcpy(), cudaFree()`
- Launching parallel kernels
  - > Launch `N` copies of `add()` with `add<<<N, 1>>>(…);`
  - > Use `blockIdx.x` to access block index

CUDA Threads

- Terminology: a block can be split into parallel threads
- Using blocks:

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

add<<<N, 1>>>(d_a, d_b, d_c);
```

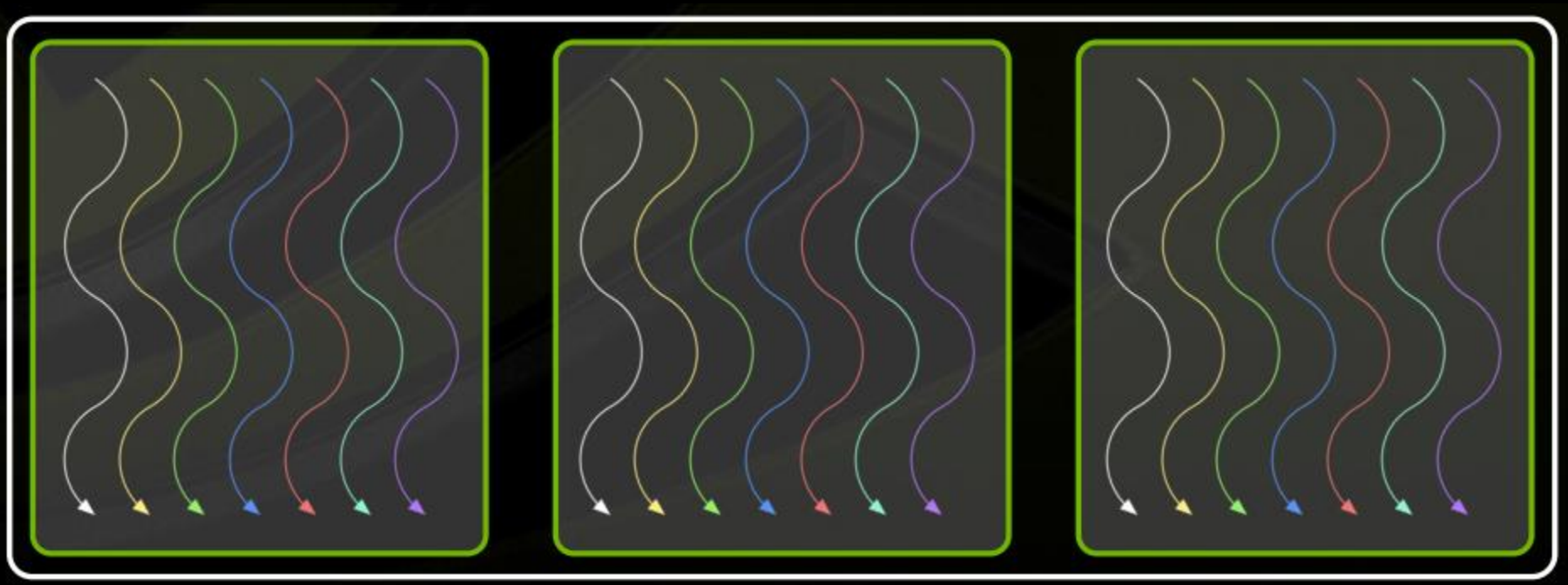
- Using threads:

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

add<<<1, N>>>(d_a, d_b, d_c);
```

Combining Blocks and Threads

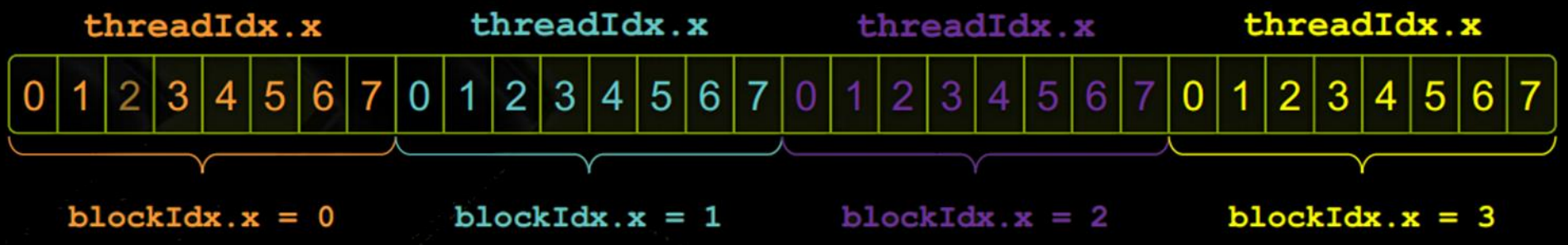
- We have seen parallel vector addition using:
  - > Many blocks with one thread each
  - > One block with many threads

- Let's adapt vector addition to use both blocks and threads



Indexing Arrays with Blocks and Threads

- No longer as simpel as using `blockIdx.x` and `threadIdx.x`
  - > Consider indexing an array with one element per thread (8 threads/block)



- With M threads/block a unique index for each thread is given by
  - > `int index = threadIdx.x + blockIdx.x * M;`

Vector Addition with Blocks and Threads

- Use the built-in variable `blockDim.x` for threads per block

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

- Combined version of `add()` to use parallel threads and parallel blocks

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

- What changes need to be made in `main()`?

Addition with Blocks and 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
        a = (int *)malloc(size); random_ints(a, N);
        b = (int *)malloc(size); random_ints(b, N);
        c = (int *)malloc(size);
```

Addition with Blocks and 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;
}
```

Handling Arbitrary Vector Size

- Typical problems are not friendly multiples of `blockDim.x`

- Avoid accessing beyond the end of the 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];
}
```

- Update the kernel launch:

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

Why Bother with Threads?

- Threads seem unnecessary
  - > They add a level of complexity
  - > What do we gain?

- Unlike parallel blocks, threads have mechanism to:
  - > Communicate
  - > Synchronize

- This goes beyond the scope of this lecture!