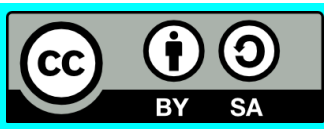
CUDA Programming

Moreno Marzolla
Dip. di Informatica—Scienza e Ingegneria (DISI)
Università di Bologna

moreno.marzolla@unibo.it



Acknowledgments

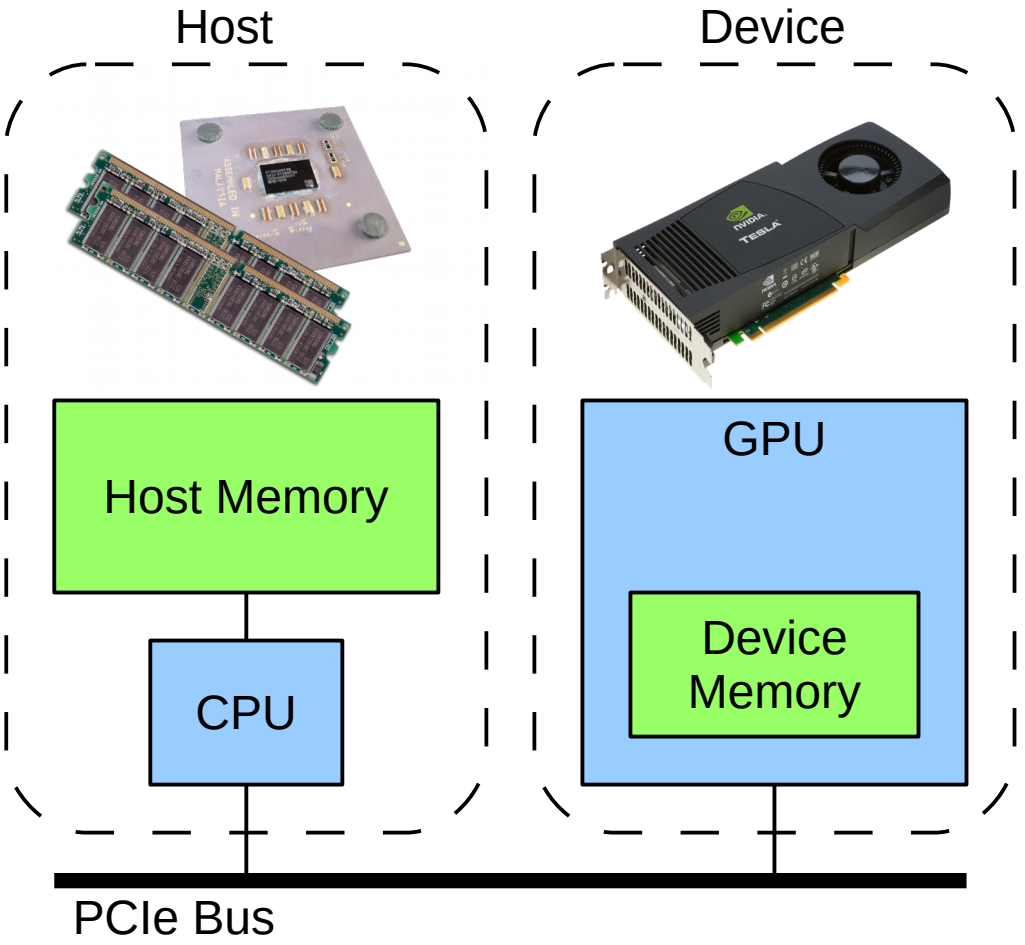
- Most of the content of this presentation is from Mark Harris (Nvidia Corporation), "*CUDA C/C++ BASICS*"
  - http://developer.download.nvidia.com/compute/developertrainingmaterials/presentations/cuda_language/Introduction_to_CUDA_C.pptx
- Salvatore Orlando (Univ. Ca' Foscari di Venezia)
- Tim Mattson (Intel Labs)
  - "*Hands-on Intro to CUDA for OpenCL programmers*"
- CUDA C programming guide
  - http://docs.nvidia.com/cuda/cuda-c-programming-guide/

Introduction

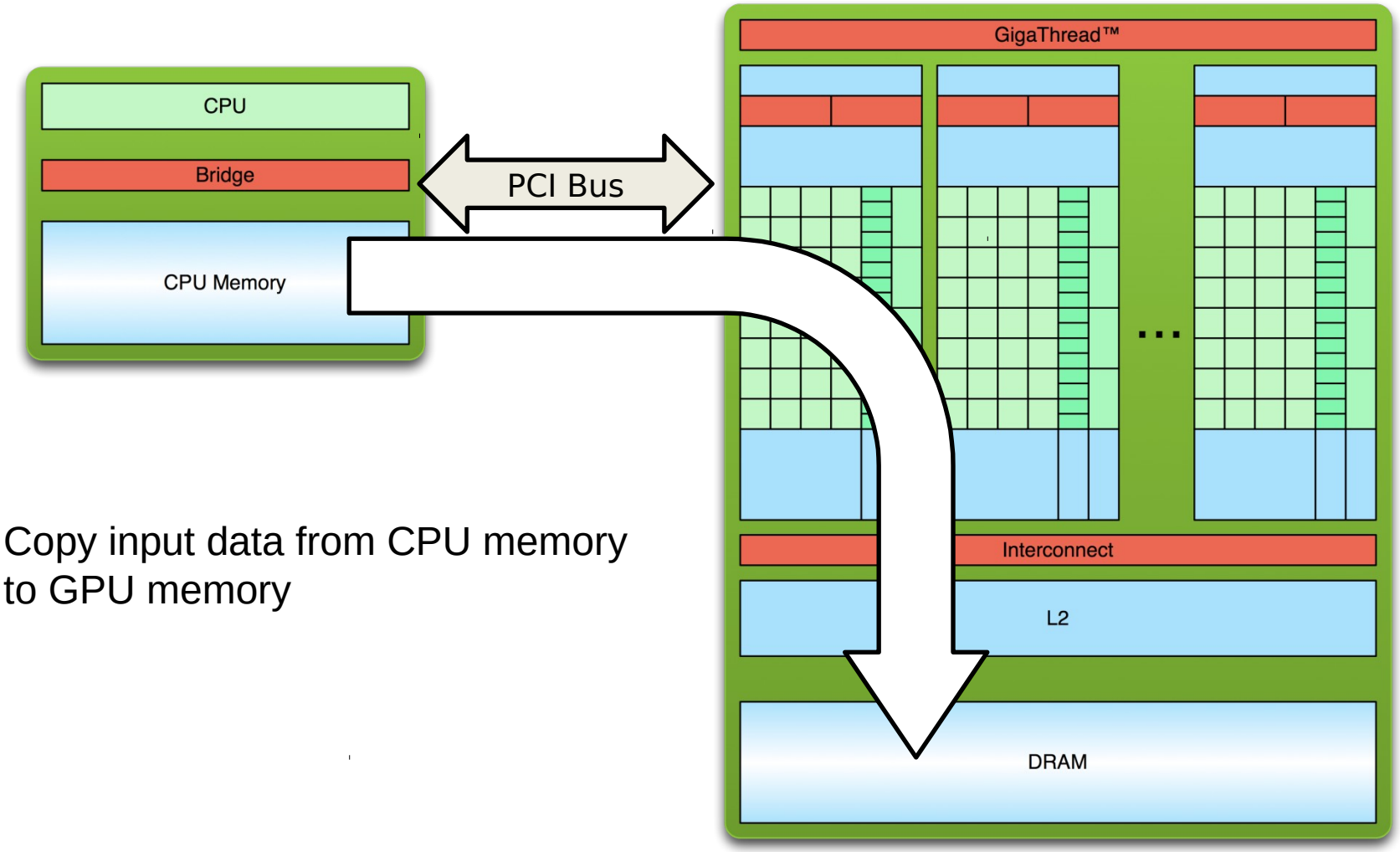
- Manycore GPUs (Graphics Processing Units) are available in almost all current hardware platforms
- Originally, these processors have been designed for graphics applications
  - Because of the high potential of data parallelism in graphics applications, the design of GPU architectures relied on specialized processor cores
- In addition to graphics processing, GPUs can also be employed for general non-graphics applications
  - If data parallelism is large enough to fully utilize the high number of compute cores in a GPU
- The trend to use GPUs for general numerical applications has inspired GPU manufacturers, such as NVIDIA, to develop the programming environment CUDA and OpenCL

Terminology

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

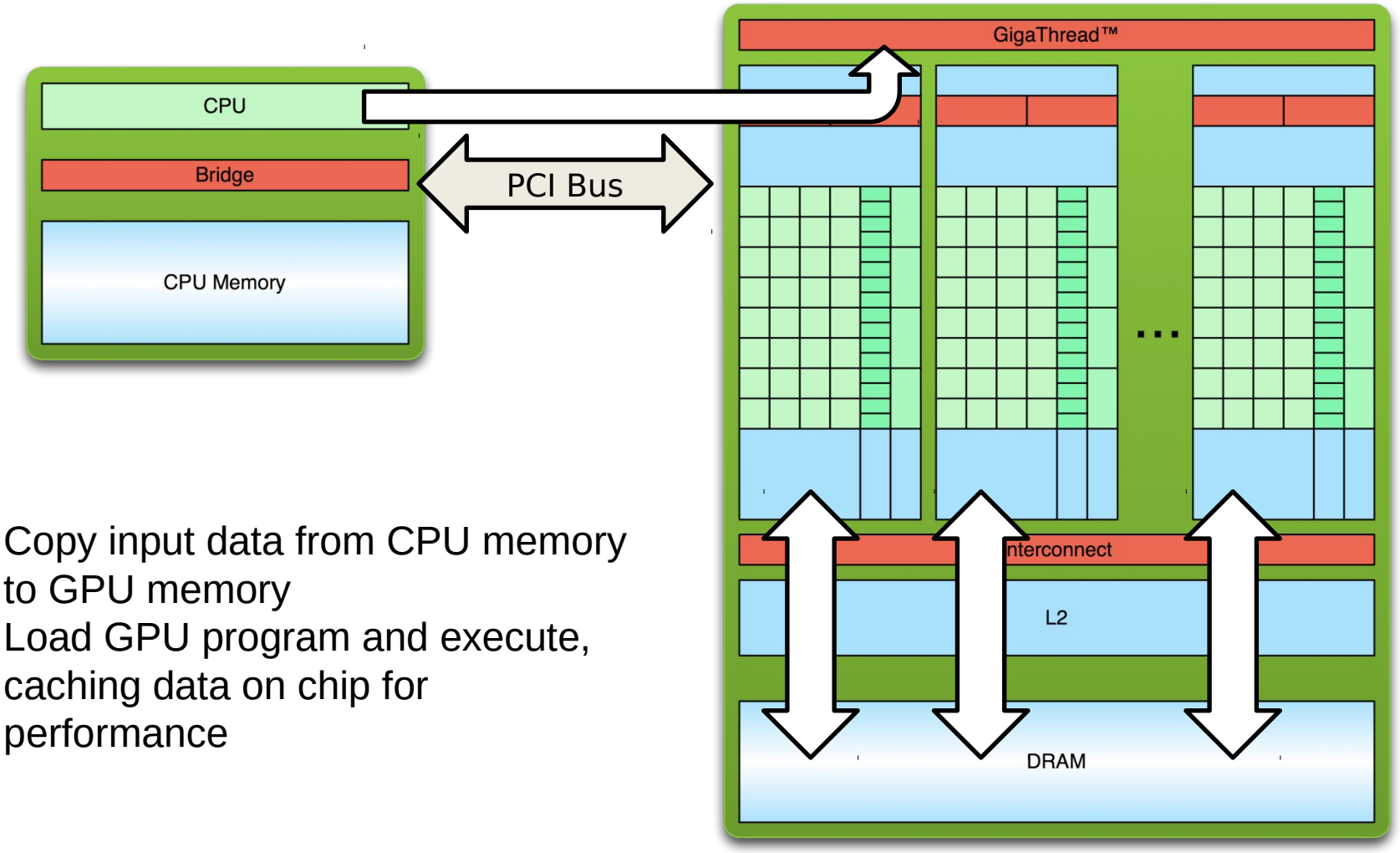


Simple Processing Flow



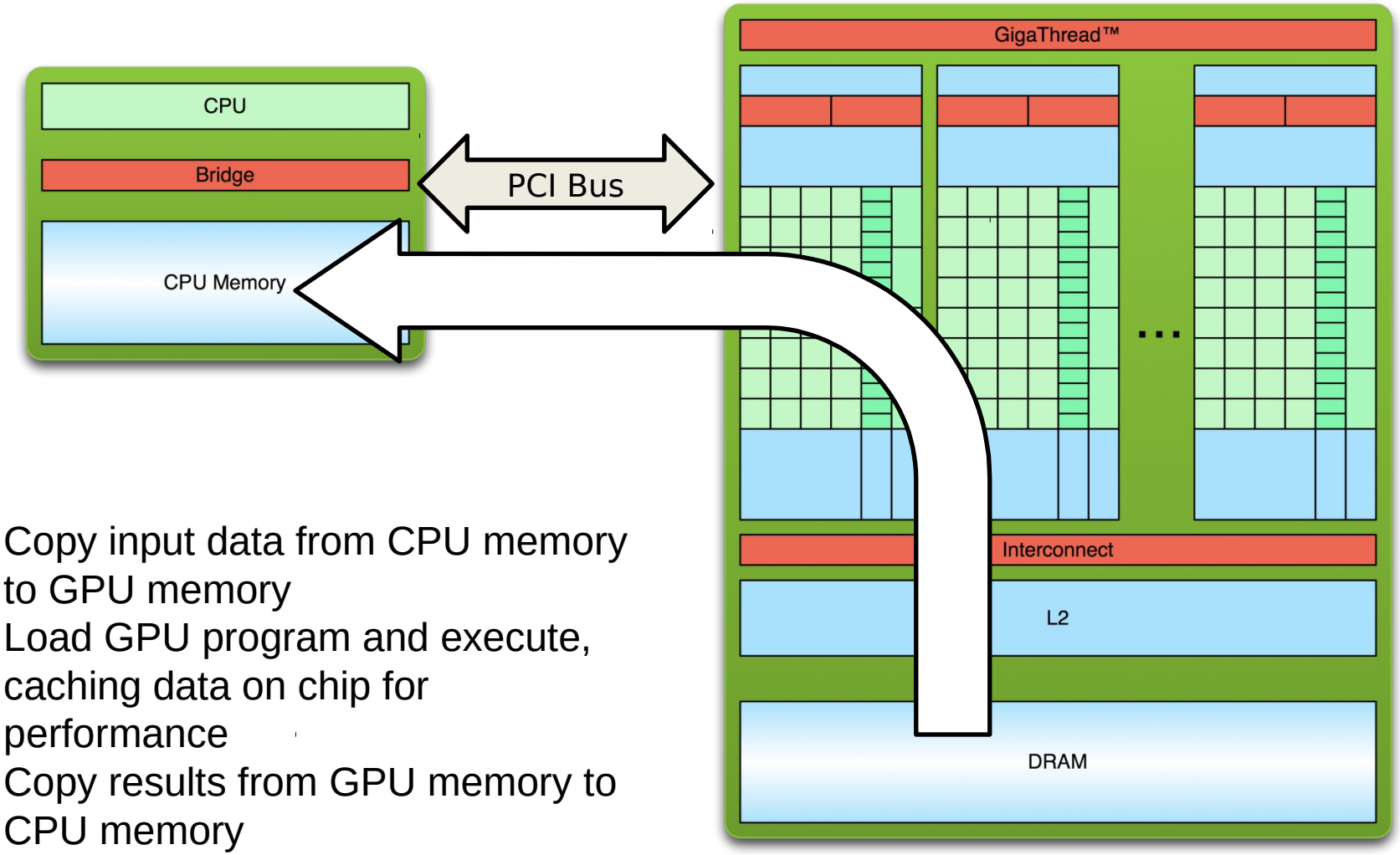
1. Copy input data from CPU memory to GPU memory

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

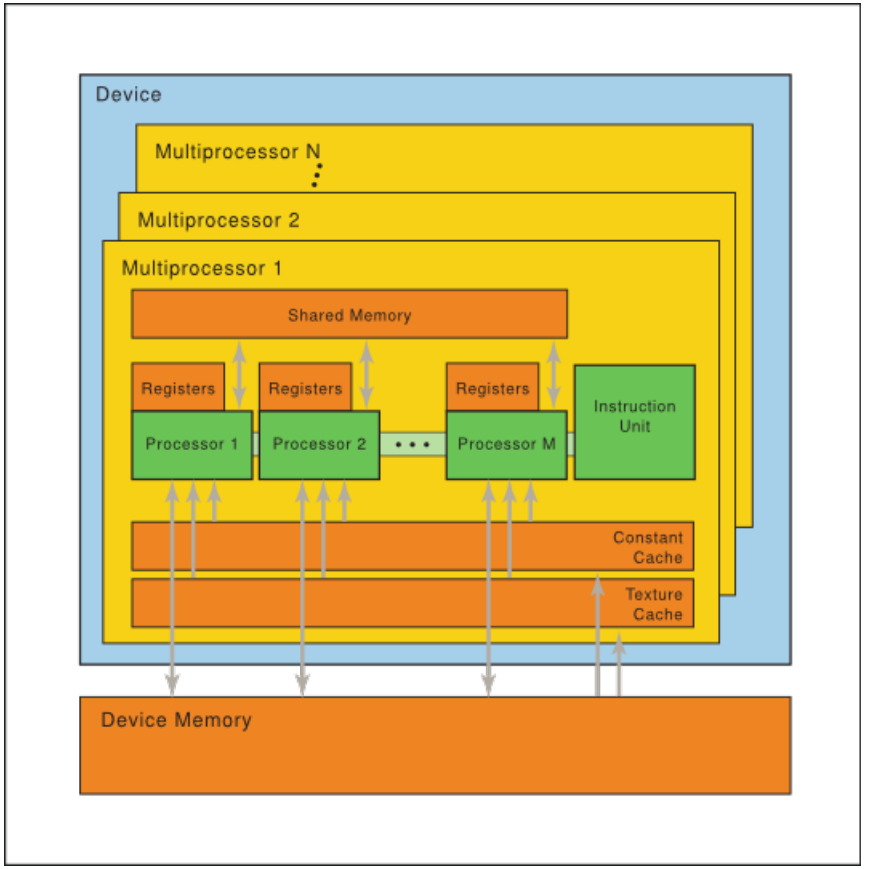
Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

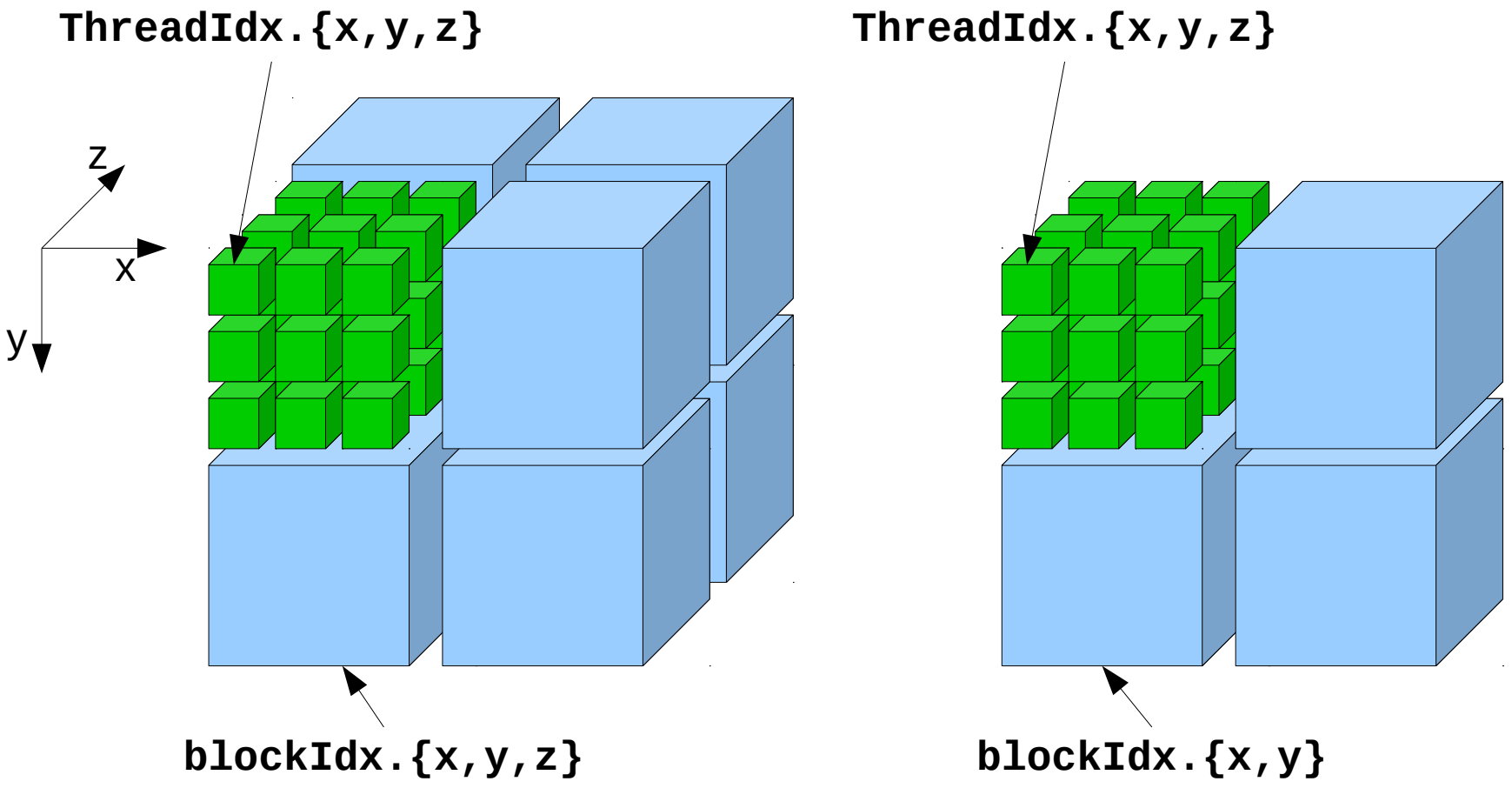
Terminology

- A GPU comprises several multi-threaded SIMD processors
  - SIMD processor = Streaming Multiprocessors (SMs) composed of many Streaming Processors (SPs)
- Each SIMD processors has several functional units (cores) that can execute the same SIMD instruction on different data
- The actual number of SIMD processors depends on the GPU model
  - For example, the NVIDIA GTX480 GPU has up to 15 SMs



http://docs.nvidia.com/cuda/parallel-thread-execution/

Blocks and Grid



Compute Capability ≥ 2.x

Compute Capability < 2.x

Hands-on introduction to CUDA programming

Hello World!

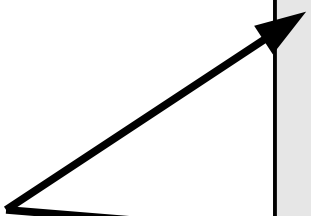
- Standard C that runs on the host
- The NVIDIA compiler (**nvcc**) can be used to compile programs, even with no device code

```
/* cuda-hello0.cu */
#include <stdio.h>
int main(void)
{
    printf("Hello World!\n");
    return 0;
}
```

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

Hello World! with Device Code

Two new syntactic elements



```
/* cuda-hello1.cu */
#include <stdio.h>
__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 keyword **__global__** indicates a function that:
  - Runs on the device
  - Is called from host code
  - **__global__** functions must return **void**
- **nvcc** separates source code into host and device components
  - Device functions (e.g., **mykernel()**) are processed by the NVIDIA compiler
  - Host functions (e.g., **main()**) are processed by the standard host compiler (e.g., **gcc**)

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”
  - We’ll return to the parameters (1,1) in a moment
- That’s all that is required to execute a function on the GPU!

Hello World! with Device Code

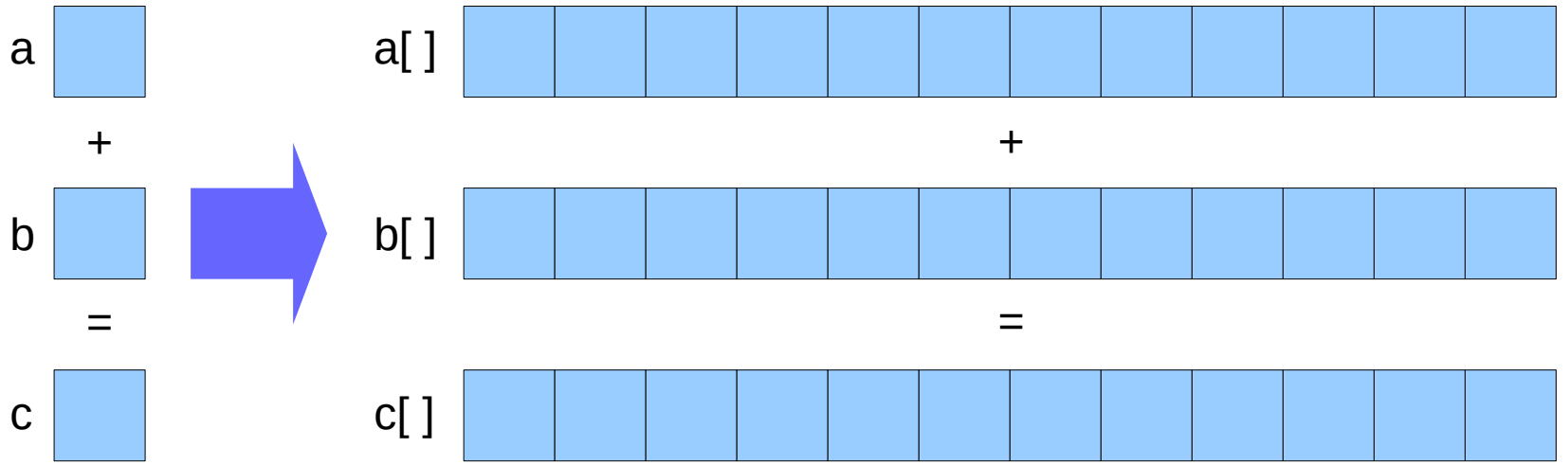
```
__global__ void mykernel(void)
{  }

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

```
$ nvcc cuda-hello1.cu
$ ./a.out
Hello World!
```

- **`mykernel()`** does nothing

Parallel Programming in CUDA/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 execute on the device
  - **`add()`** will be called from the host

Addition on the Device

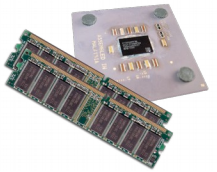
- Note the use of pointers for the variables

```
__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()**

```
/* cuda-vecadd0.cu */
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 */
    const size_t 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;
    /* 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;
}
```

Coordinating Host & Device

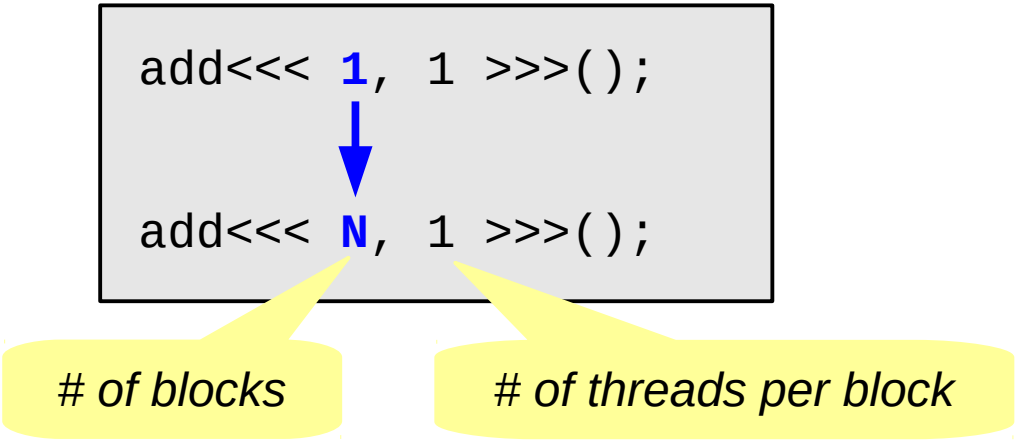
- Kernel launches are asynchronous
  - Control returns to the CPU immediately
- CPU needs to synchronize before consuming the results

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

Running in parallel

Moving to Parallel

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



- 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 its block index using **`blockIdx.x`**

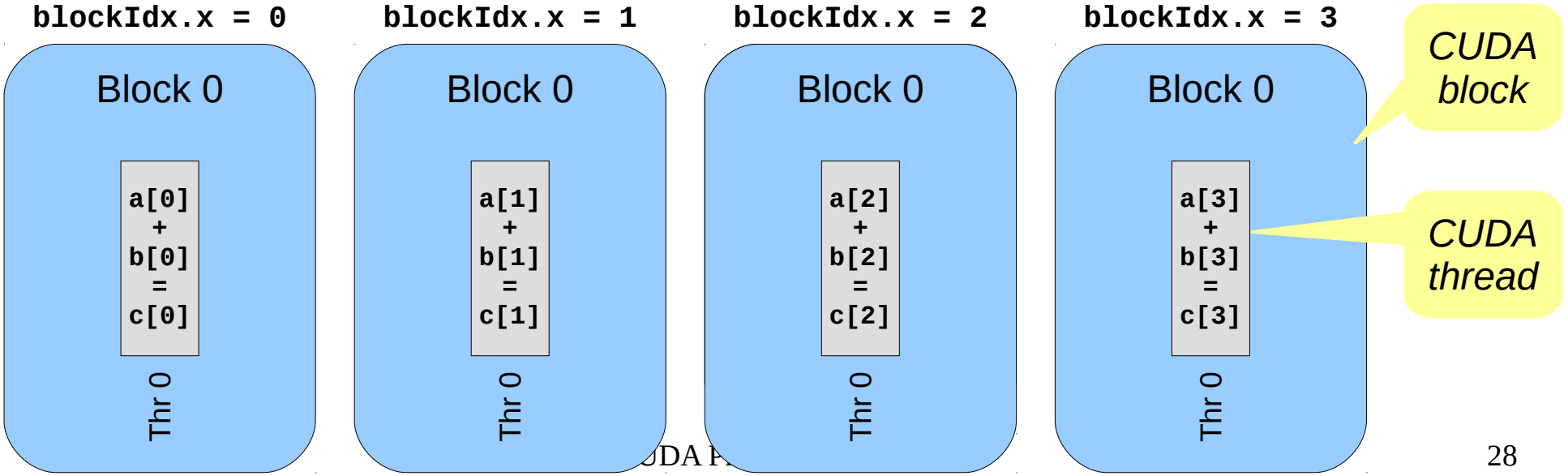
```
__global__ void add(int *a, int *b, int *c)
{
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

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

Vector Addition on the Device

- On the device, each block can execute in parallel

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



```
/* cuda-vecadd1.cu */
#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 */
    const size_t 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); vec_init(a, N);
    b = (int *)malloc(size); vec_init(b, N);
    c = (int *)malloc(size);
    /* 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
  - 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

Introducing threads

CUDA Threads

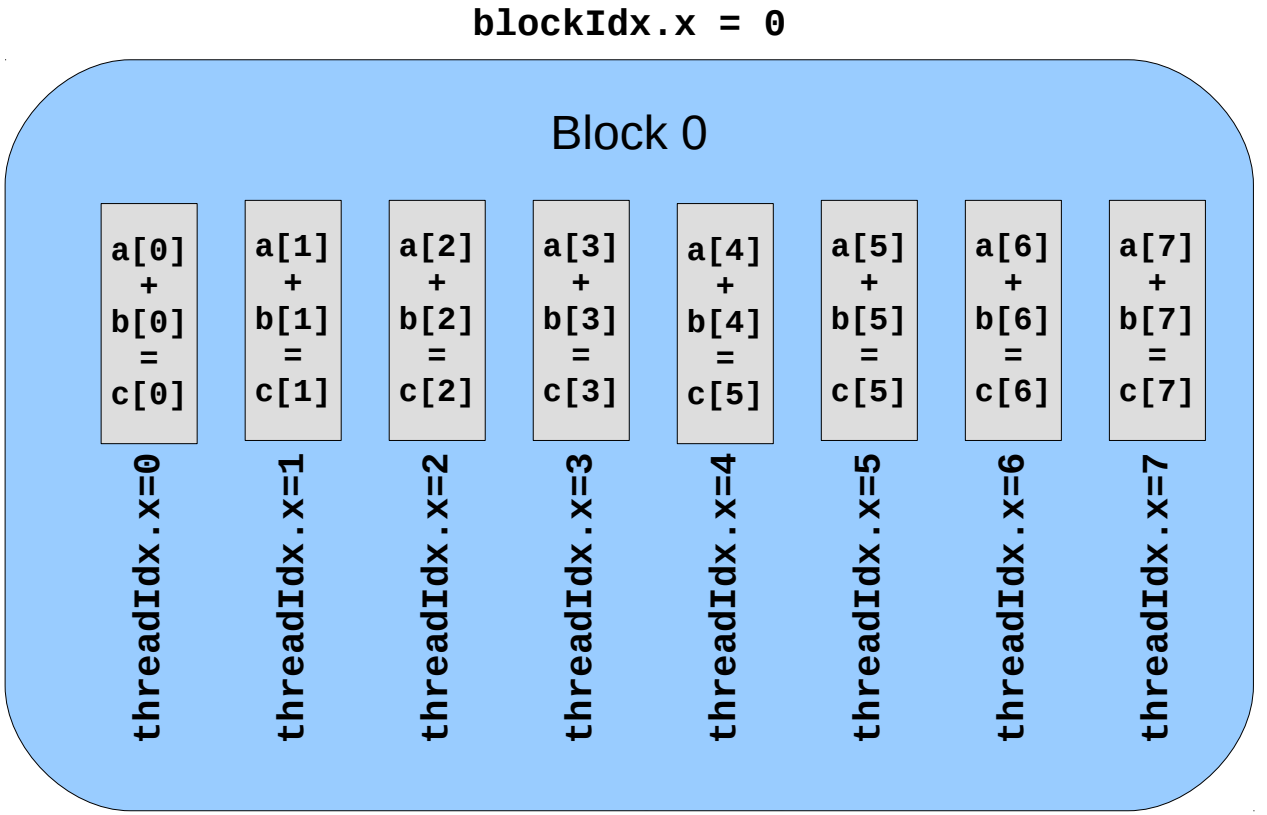
- Terminology: a block can be split into parallel threads
- Let's change **`add()`** to use parallel threads instead of parallel blocks

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

- We use **`threadIdx.x`** instead of **`blockIdx.x`**
- Need to make one change in **`main()`**…

CUDA Threads

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



```
/* cuda-vecadd2.cu */
#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 */
    const size_t 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);
    /* 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;
}
```

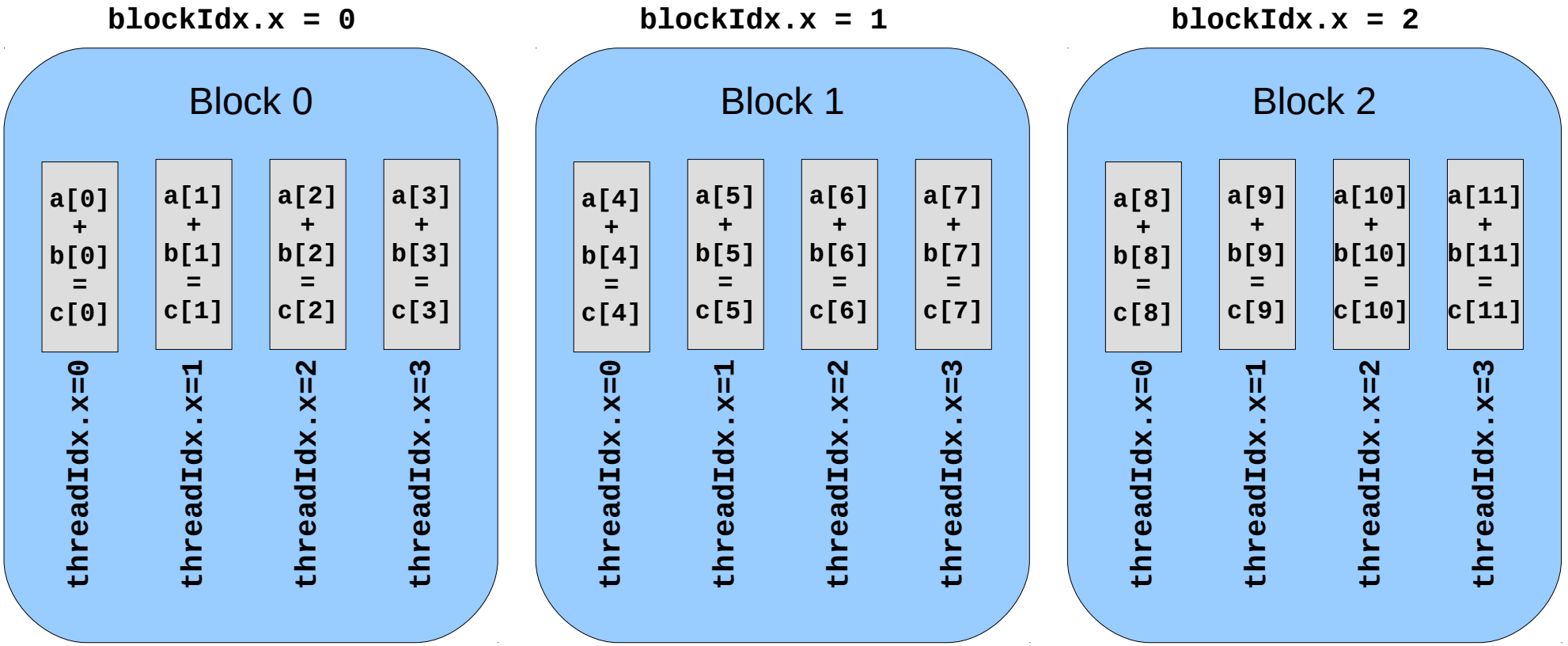
Combining threads and blocks

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
  - Why? We'll come to that…
- First let's discuss data indexing…

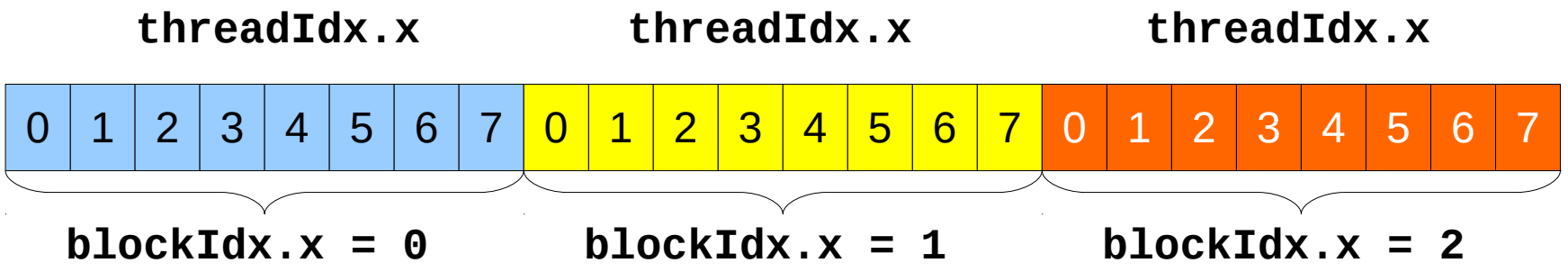
Combining Blocks and Threads

- We must somehow assign to each thread a different array element to operate on



Indexing Arrays with Blocks and Threads

- No longer as simple as using **`blockIdx.x`** or **`threadIdx.x`** alone
- Consider indexing an array with one element per thread, 8 threads per block

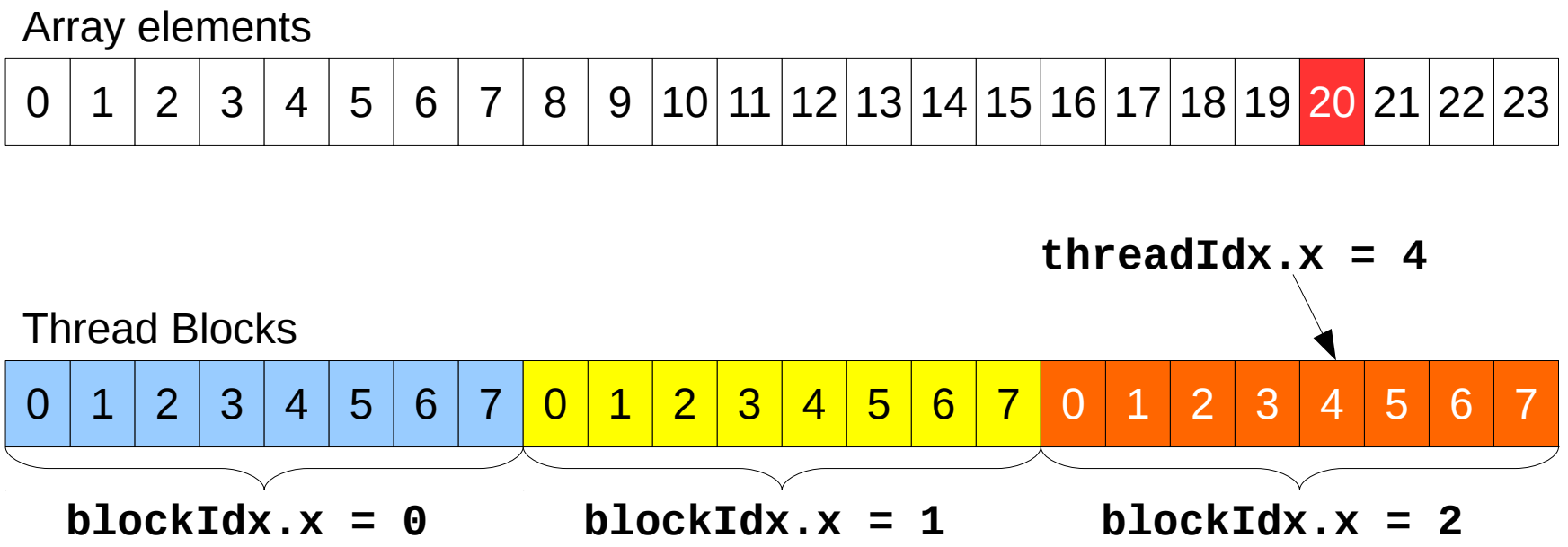


- With *M* threads per block a unique index for each thread is given by:

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

Indexing arrays: Example

- Which thread will operate on the red element?




```
int index = threadIdx.x + blockIdx.x * M;
          =      4      +     2      * 8;
          = 20;
```

Vector Addition with Blocks and Threads

- Use the built-in variable **`blockDim.x`** to get the number of 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

```
#define N (2048*2048)
#define BLKDIM 512

int main(void) {
    …
    /* Launch add() kernel on GPU */
    add<<<N/BLKDIM, BLKDIM>>>(d_a, d_b, d_c);
    …
}
```

Number of blocks

Number of threads per block

- However, the problem size might not be multiple of the block size…

Handling arbitrary vector sizes

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

- Update kernel launch

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

- See cuda-vecadd3.cu

Review

- Launching parallel kernels
  - Launch ~*N* copies of **`add()`** with **`add<<<(N + BLKDIM-1)/BLKDIM, BLKDIM>>>(…)`**;
  - Use **`blockIdx.x`** to access block index
  - Use **`threadIdx.x`** to access thread index within block
- Assign array elements to threads:

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

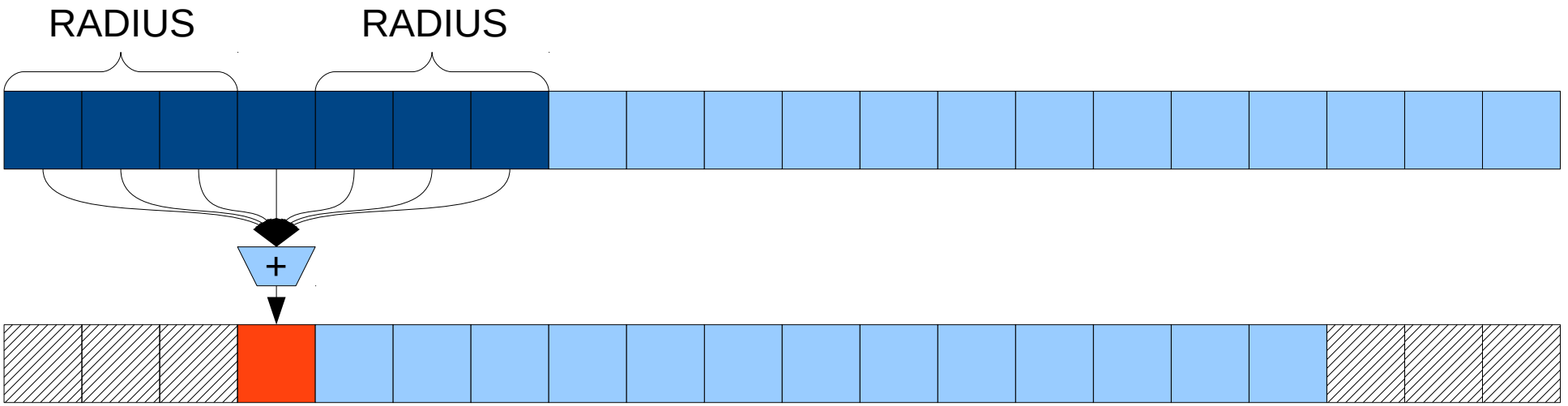
Why Bother with Threads?

- Threads seem unnecessary
  - They add a level of complexity
  - What do we gain?
- Unlike parallel blocks, threads have mechanisms to:
  - Communicate
  - Synchronize
- To look closer, we need a new example…

Cooperating threads

1D Stencil

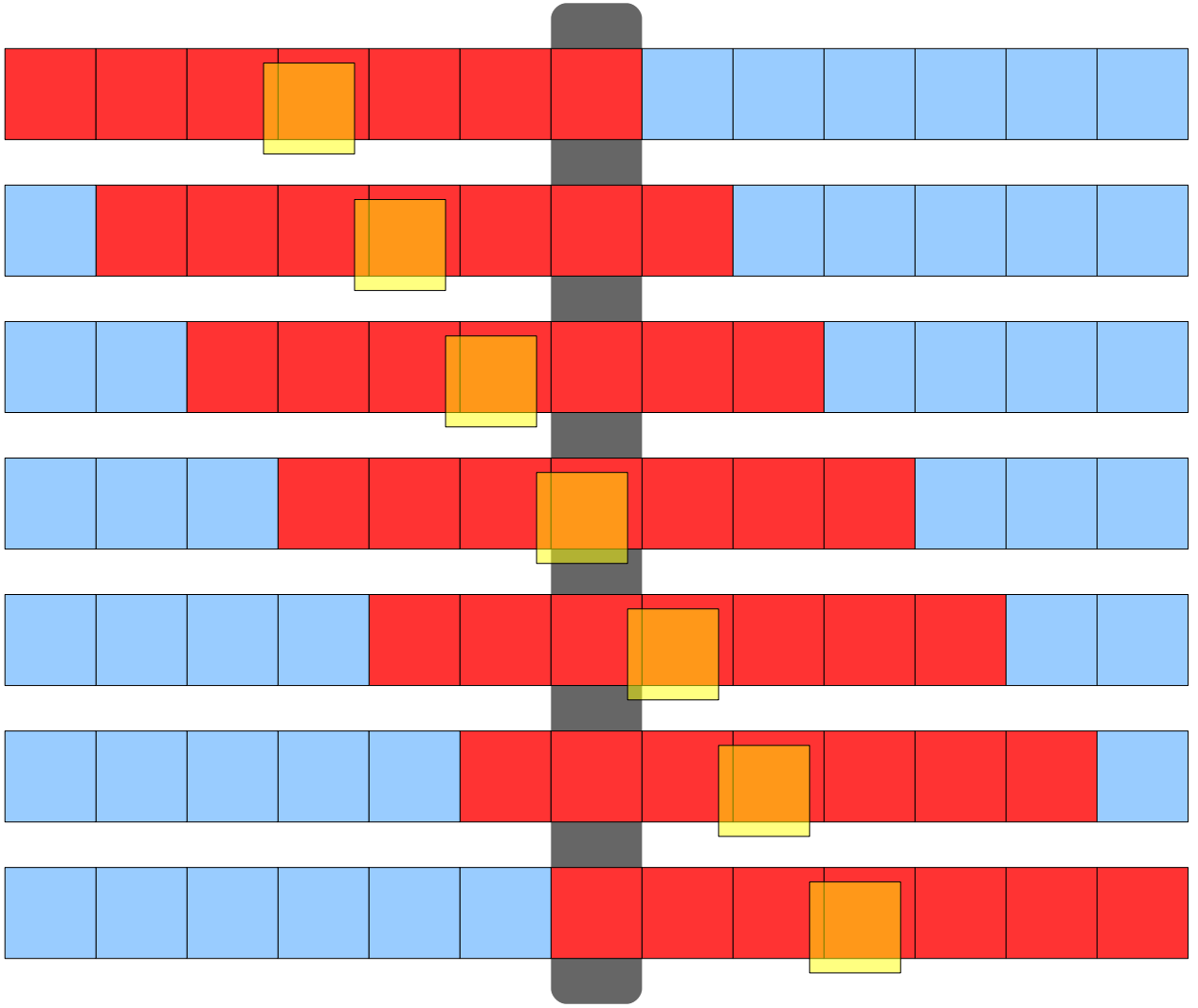
- Consider applying a stencil to a 1D array of elements
  - Each output element is the sum of input elements within a given radius
- If RADIUS is 3, then each output element is the sum of 7 input elements
  - The first and last RADIUS elements of the output array are not computed



See cuda-stencil1d.c

Implementing Within a Block

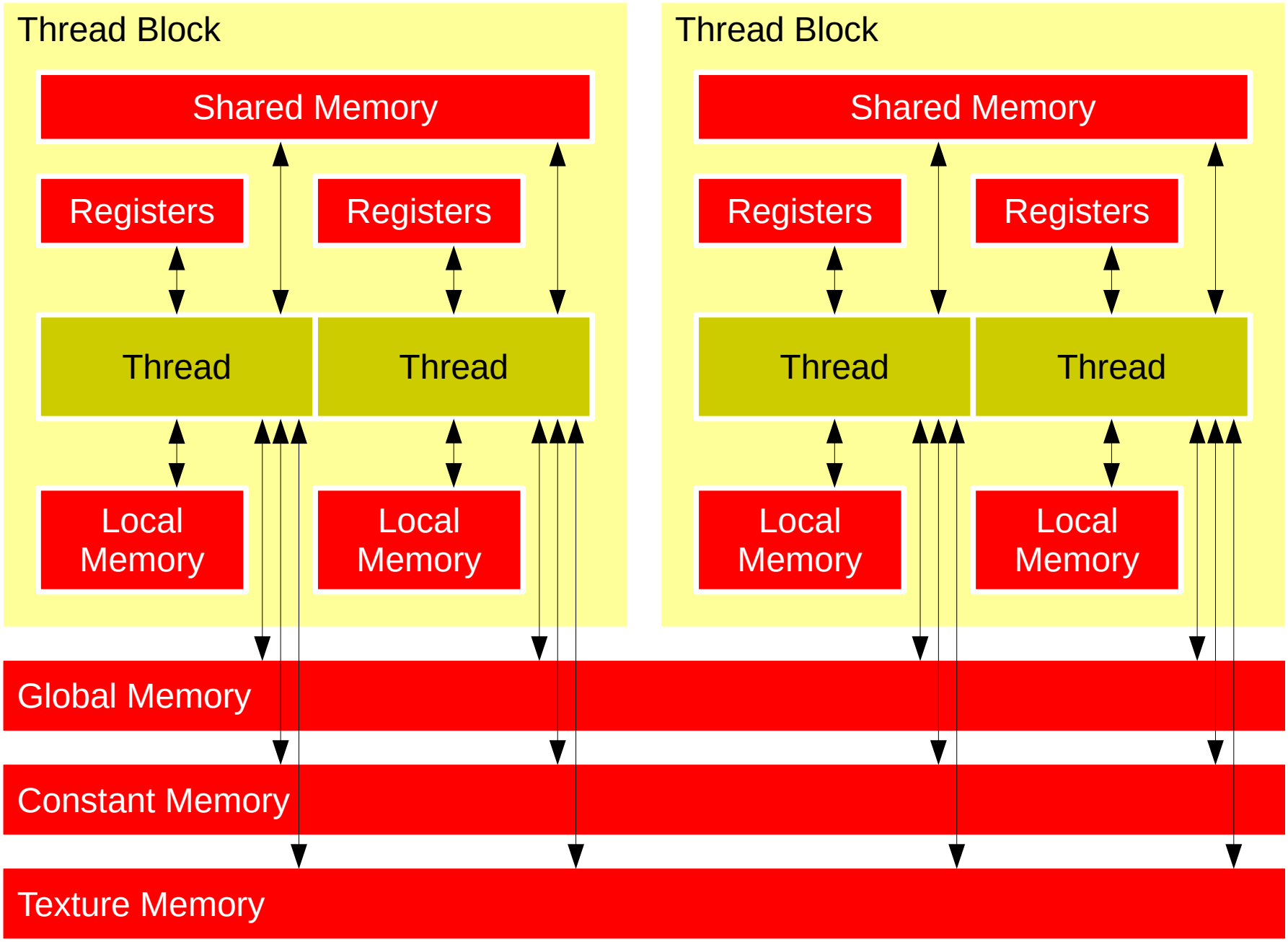
- Each thread processes one output element
  - **`blockDim.x`** elements per block
- Input elements are read several times
  - With radius 3, each input element is read seven times
  - With radius *R*, each input element is read (2*R*+1) times



Sharing Data Between Threads

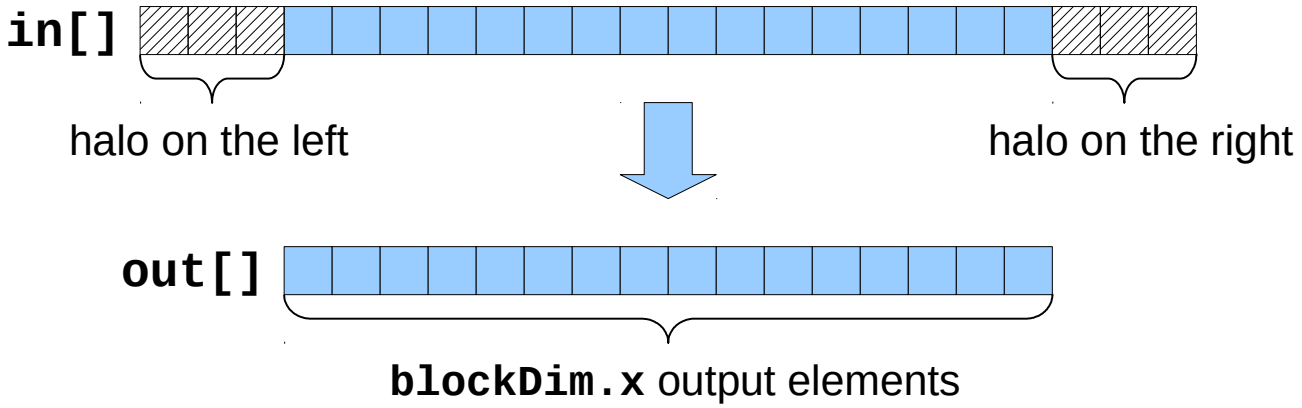
- Global memory accesses are likely to cause a bottleneck due to the limited memory bandwidth
- Within a block, threads can share data via shared memory
  - Extremely fast on-chip memory, user-managed
  - Think of it as a user-managed local cache
- Declare using **`__shared__`**, allocated per thread-block
- Data is not visible to threads in other blocks

CUDA memory model



Implementing with shared memory

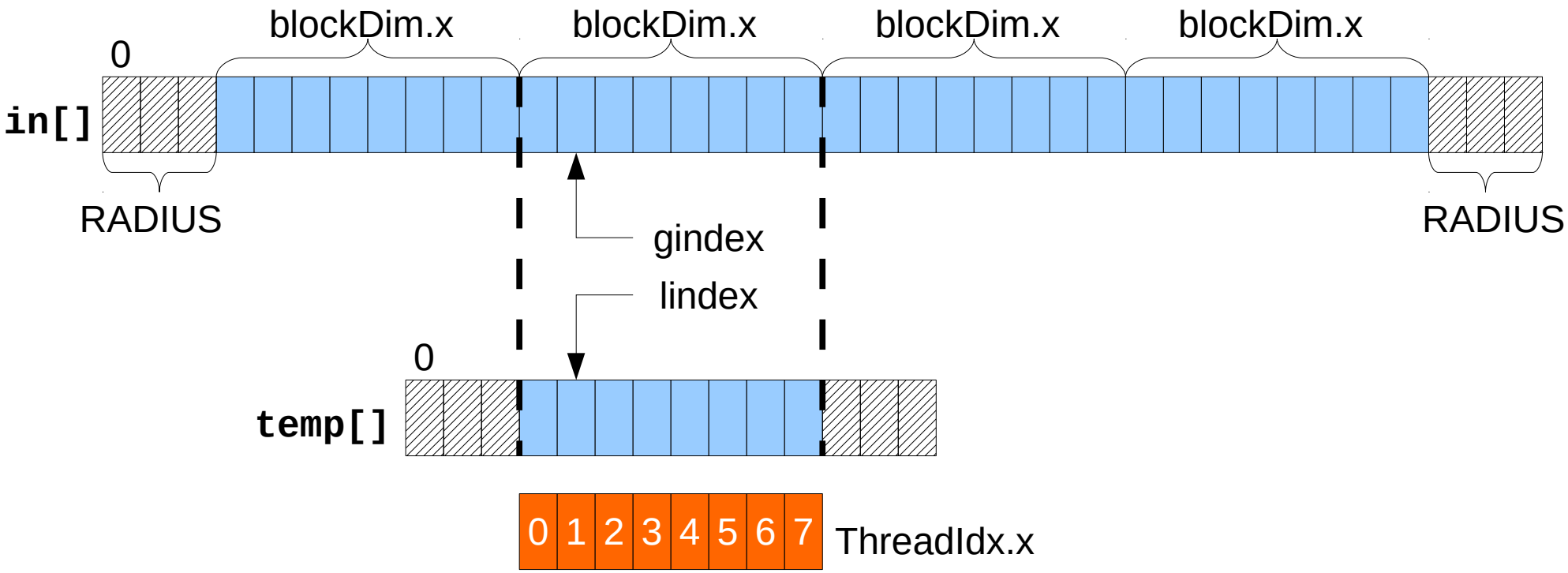
- Cache data in shared memory
  - Read (**`blockDim.x`** + 2 × radius) input elements from global memory to shared memory
  - Compute **`blockDim.x`** output elements
  - Write **`blockDim.x`** output elements to global memory
  - Each block needs a halo of *radius* elements at each boundary



Implementing with shared memory

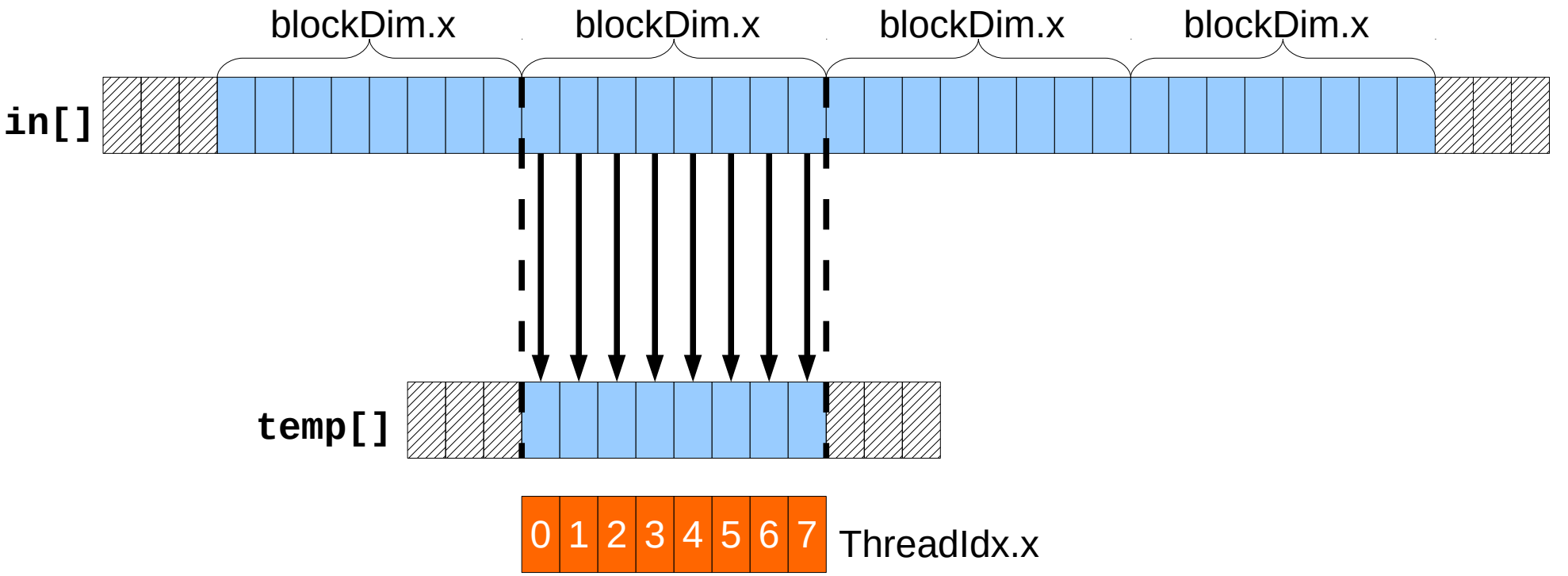
- Let us make a few simplifying assumptions
  - The array length is a multiple of the thread block size
  - The input (and output) array already includes an halo of **`2*RADIUS`** elements
    - The halo is ignored for the output array
- Idea
  - Each thread block keeps a local cache of **`blockDim.x + 2*RADIUS`** elements
  - Each thread copies one element from the global array to the local cache
  - The first **`RADIUS`** threads also take care of of filling the halo

Implementing with shared memory



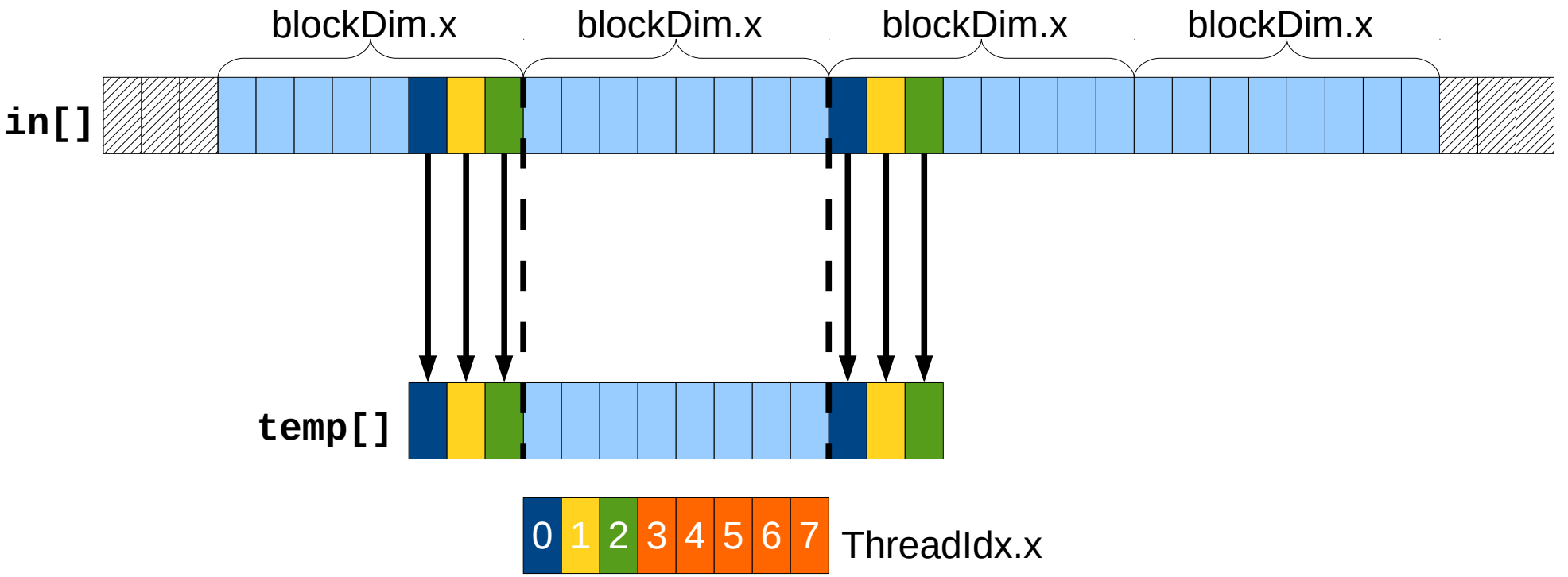

```
__shared__ int temp[BLKDIM + 2 * RADIUS];
const int gindex = threadIdx.x + blockIdx.x * blockDim.x + RADIUS;
const int lindex = threadIdx.x + RADIUS;
/* … */
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
     temp[lindex - RADIUS] = in[gindex - RADIUS];
     temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}
```

Implementing with shared memory




```
__shared__ int temp[BLKDIM + 2 * RADIUS];
const int gindex = threadIdx.x + blockIdx.x * blockDim.x + RADIUS;
const int lindex = threadIdx.x + RADIUS;
/* … */
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
     temp[lindex - RADIUS] = in[gindex - RADIUS];
     temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}
```

Implementing with shared memory




```
__shared__ int temp[BLKDIM + 2 * RADIUS];
const int gindex = threadIdx.x + blockIdx.x * blockDim.x + RADIUS;
const int lindex = threadIdx.x + RADIUS;
/* … */
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}
```

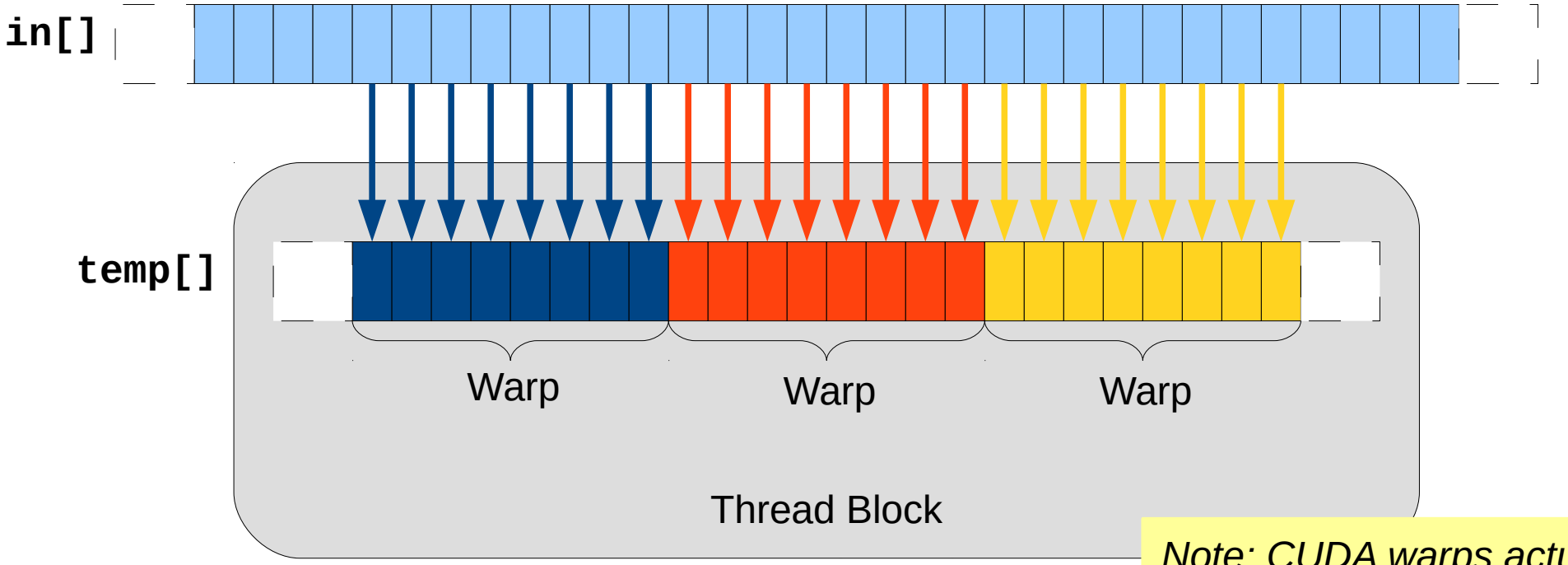
Stencil kernel (does not work!)

Wrong!

```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLKDIM + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x + RADIUS;
    int lindex = threadIdx.x + RADIUS;
    int result = 0, offset;
    /* Read input elements into shared memory */
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + blockDim.x] = in[gindex + blockDim.x];
    }
    /* Apply the stencil */
    for (offset = -RADIUS ; offset <= RADIUS ; offset++) {
        result += temp[lindex + offset];
    }
    /* Store the result */
    out[gindex] = result;
}
```

The problem

- All threads are not necessarily fully synchronized
- Suppose that thread (**`blockDim.x`** - 1) reads the halo before thread 0 has fetched it
  - Data race!



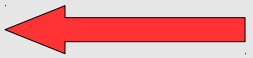
Note: CUDA warps actually contain 32 threads

The solution: `__syncthreads()`

- Synchronizes all threads within a block
  - Used to prevent RAW / WAR / WAW hazards
- All threads must reach the barrier
  - In conditional code, the condition must be uniform across the block

Stencil kernel that works

```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLKDIM + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x + RADIUS;
    int lindex = threadIdx.x + RADIUS;
    int result = 0, offset;
    /* Read input elements into shared memory */
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + blockDim.x] = in[gindex + blockDim.x];
    }
    __syncthreads();
    /* Apply the stencil */
    for (offset = -RADIUS ; offset <= RADIUS ; offset++) {
        result += temp[lindex + offset];
    }
    /* Store the result */
    out[gindex] = result;
}
```



See cuda-stencil1d-shared.c

Review

- Use **`__shared__`** to declare a variable/array in shared memory
  - Data is shared between threads in a block
  - Not visible to threads in other blocks
- Use **`__syncthreads()`** as a barrier
  - Use to prevent data hazards