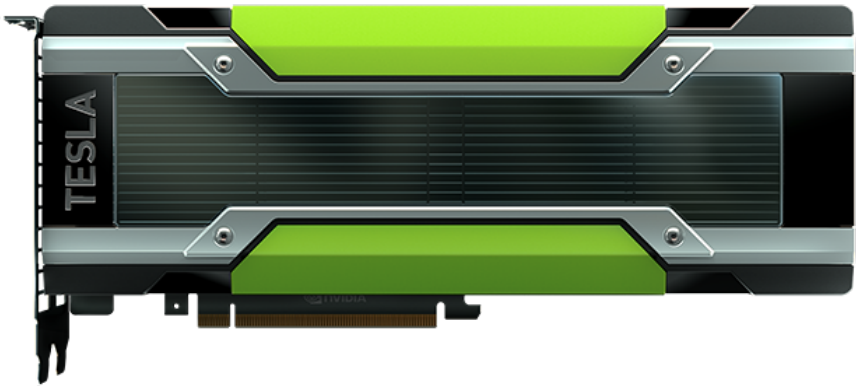
GPU Programming with CUDA

Bryan Mills, PhD

Spring 2017

GPUs at Comet

- Nvidia Tesla Kepler (K80)
  - 4992 GPU Cores (Stream Processors)
  - 24Gb Ram
  - 2.91 teraflops – 64-bit
- Nvidia Tesla P100
  - 3584 GPU Cores
  - 16Gb Ram
  - 4.7 teraflops – 64 bit



3000+ cores?


SMX
Instruction Cache
Warp Scheduler
Dispatch
Register File (65,536 x 32-bit)
Core
DP Unit
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
48 KB Read-Only Data Cache
Tex

What is CUDA?

- Compute Unified Device Architecture
- Exposes GPU architecture for general purpose computing
- Does so using standard c/c++ library
- Relatively small set of extensions
- Wrappers for other languages (ex: python, java)

Running CUDA on Comet

- Separate job queue (gpudev)

```
srun --partition=gpu
  --gres=gpu:k80:4
  --pty --nodes=1
  --ntasks-per-node=24 -t 00:30:00
  --wait=0 --export=ALL /bin/bash
```

- Load kernel module

```
module load cuda
```

- Use nvcc compiler

```
nvcc hello_world.cu
```

Heterogeneous Computing

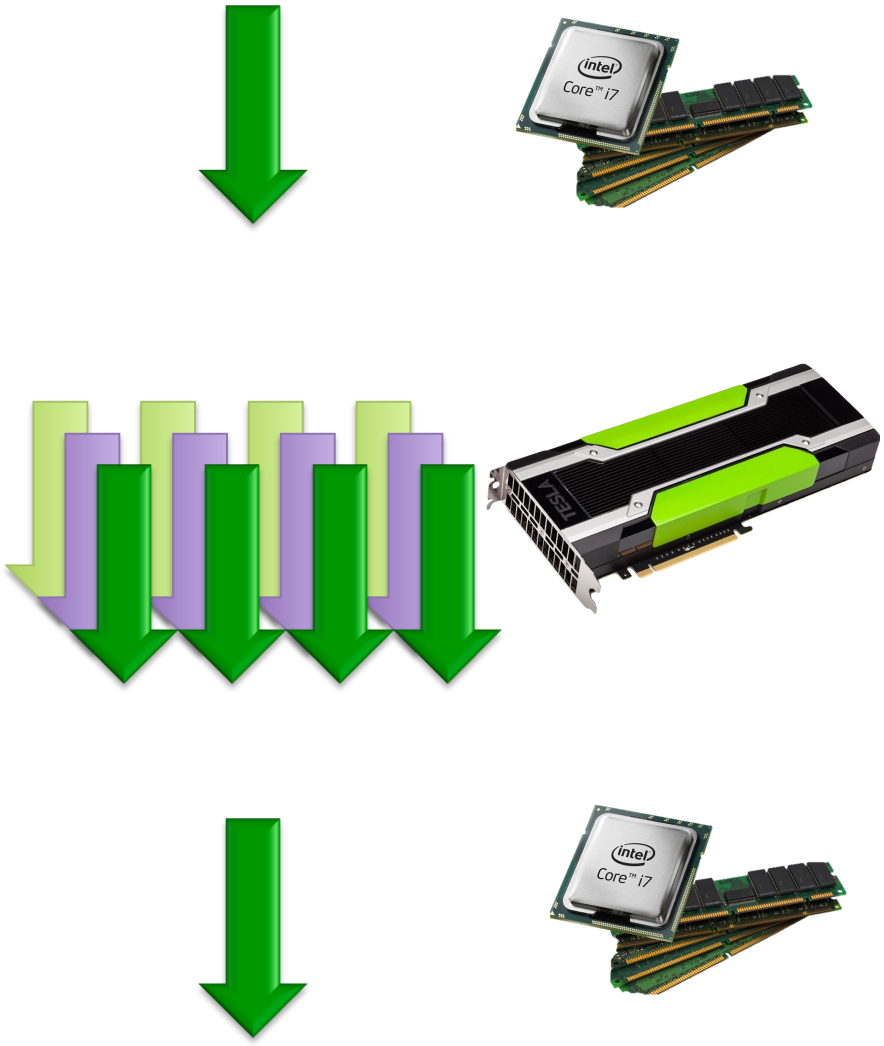
- Host
  - CPU and Memory
- Device
  - GPU and it's Memory

Host

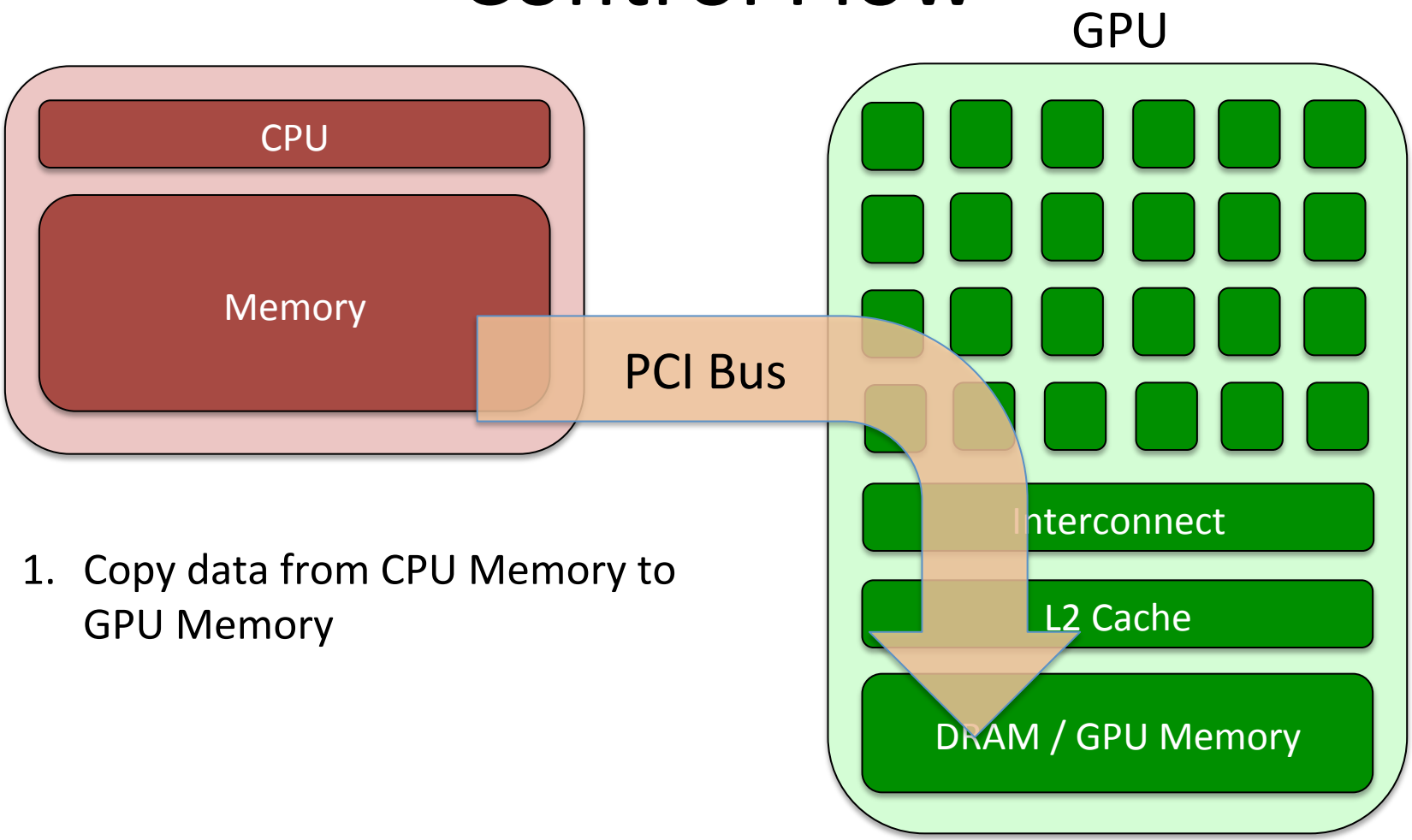
Device

Heterogeneous Computing

```
#include <iostream>
#include <math.h>
#include <stdio.h>

__global__ void add(int n, float *x, float *y) {
  int index = threadIdx.x;
  int stride = blockDim.x;
  for (int i = index; i < n; i += stride)
      y[i] = x[i] + y[i];
}

void FillWithData(int n, float* x, float* y) {
  for (int i = 0; i < n; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }
}

int main(void) {
  int N = 1<<20;
  float *x, *y;
  float *d_x, *d_y;
  int size = N * sizeof(float);

  x = (float*) malloc(size);
  y = (float*) malloc(size);
  FillWithData(N, x, y);

  cudaMalloc(&d_x, size);
  cudaMalloc(&d_y, size);
  cudaMemcpy(d_x, x, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, size, cudaMemcpyHostToDevice);

  add<<<2, 256>>>(N, d_x, d_y);

  cudaMemcpy(x, d_x, size, cudaMemcpyDeviceToHost);
  cudaMemcpy(y, d_y, size, cudaMemcpyDeviceToHost);

  int i = 0;
  int sample_rate = N / 10;
  for (i = 0; i < N; i=i+sample_rate) {
    printf("Value y[%d] = %f\n" , i, y[i]);
  }

  // Free memory
  free(x); free(y);
  cudaFree(d_x); cudaFree(d_y);

  return 0;
}
```

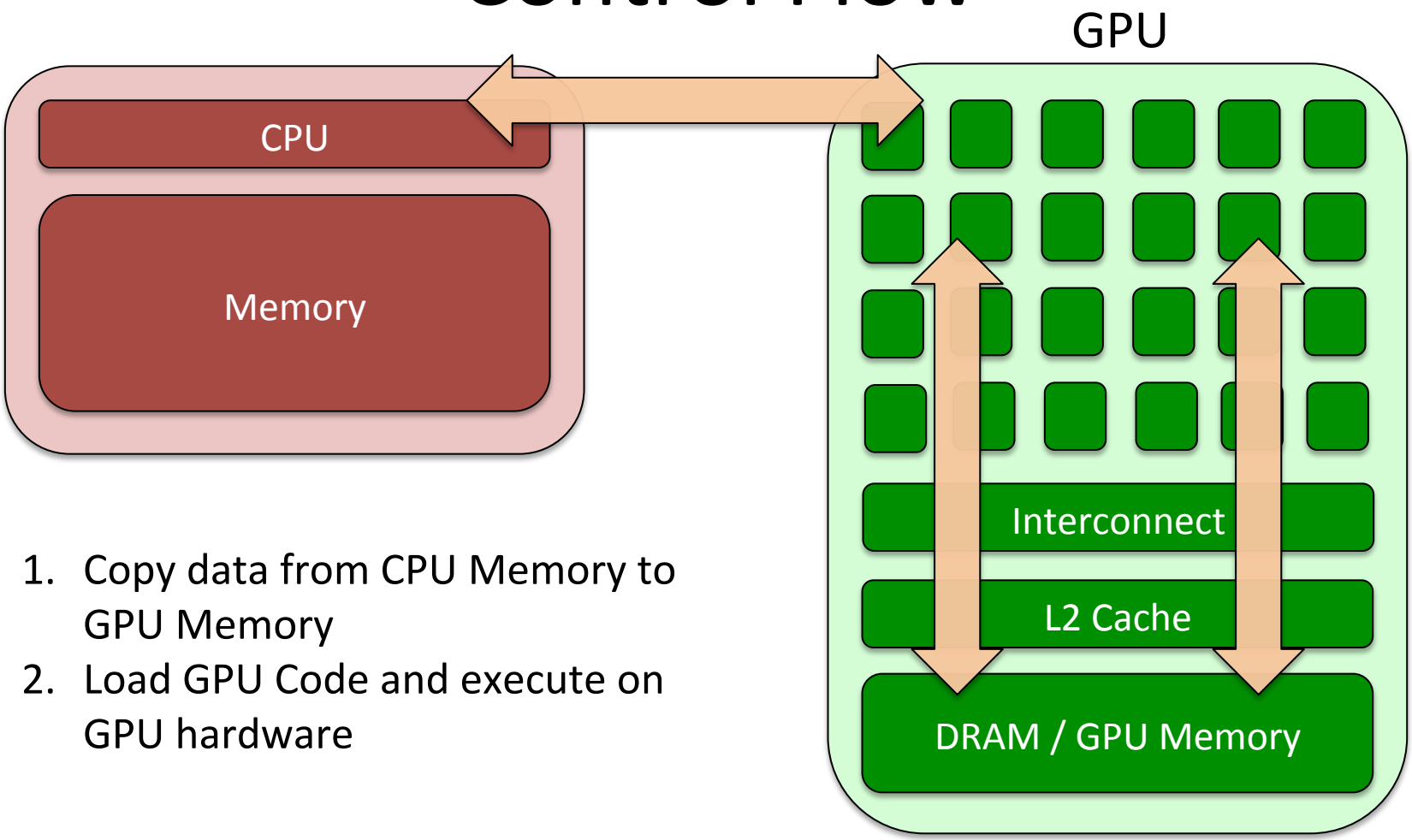


Control Flow



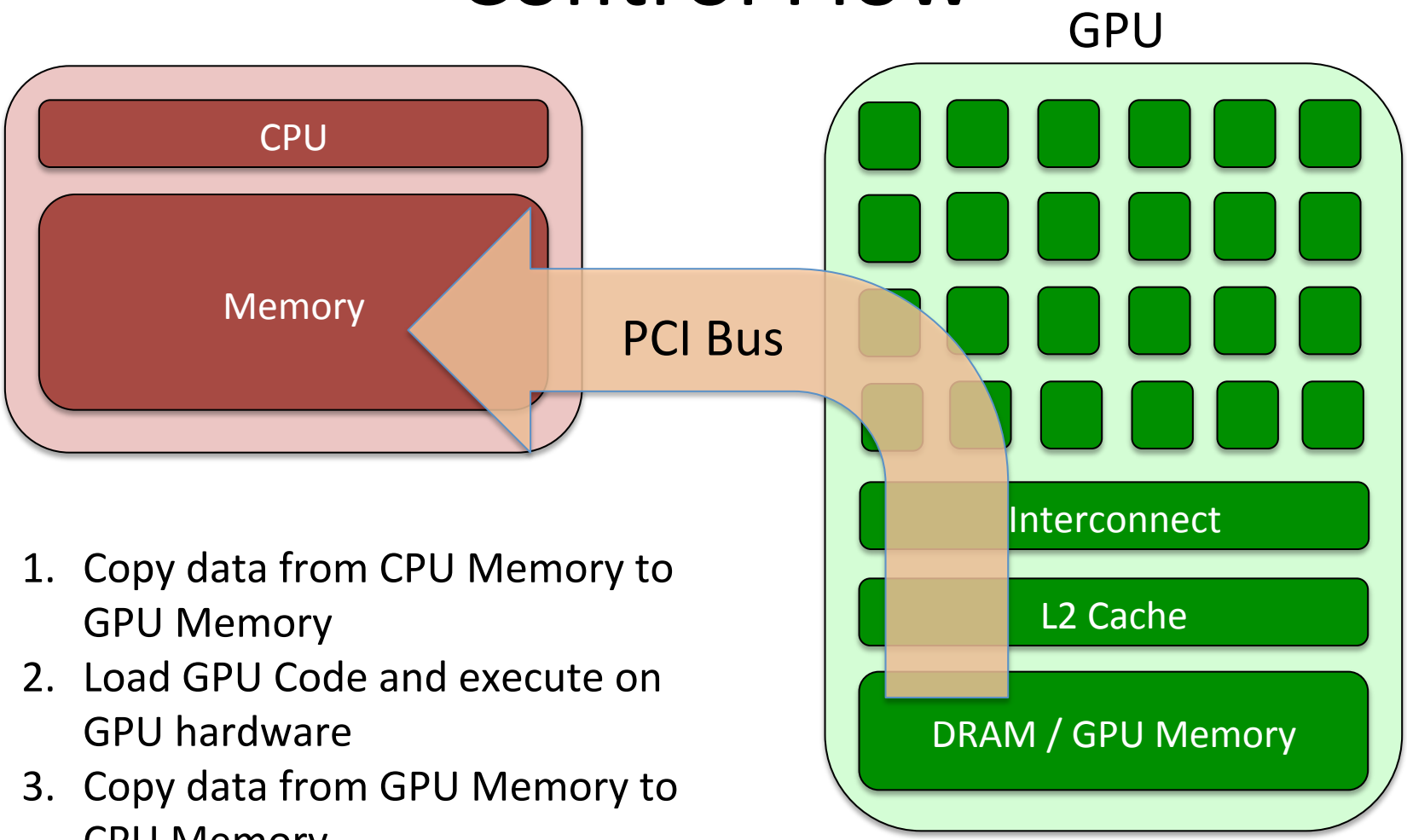
1. Copy data from CPU Memory to GPU Memory

Control Flow



1. Copy data from CPU Memory to GPU Memory
2. Load GPU Code and execute on GPU hardware

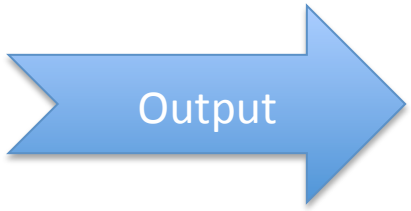
Control Flow



1. Copy data from CPU Memory to GPU Memory
2. Load GPU Code and execute on GPU hardware
3. Copy data from GPU Memory to CPU Memory

Hello World

```
__global__ void mykernel(void) {
    printf("Hello World from GPU!\n");
}

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




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

Syntactic Elements

```
__global__ void mykernel(void) {
    printf("Hello World from GPU!\n");
}
```

- `__global__` indicates function that is:
  - Executed on GPU
  - Called from CPU
- `nvcc` seperates GPU and CPU code
  - Code for GPU are compiled to that device
  - Main and other function compiled for host system

Syntactic Elements

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

- Triple brackets mark a call from host code to device
  - Also called a “kernel” launch
  - Ignore parameters for the moment `<<<1, 1>>>`
- Wow, that was easy.

Hello World

```
__global__ void mykernel(void) {
    printf("Hello World from GPU!\n");
}

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

printf doesn't always work from GPU, depends on CUDA version and device.

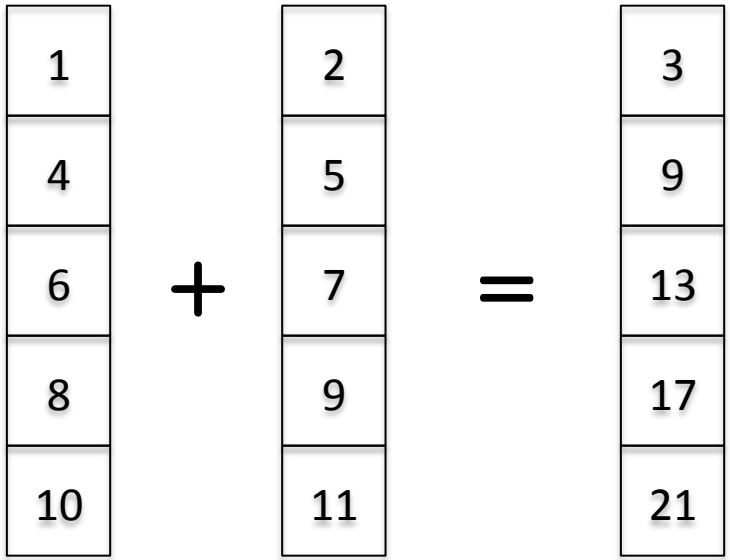
Let's do something more!

Output

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

Vector Addition

- Start easy, just add two vectors.



Add Kernel

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

- Add runs on the device.
  - So a, b, and c must be pointers in device memory, not the CPU
- We need to allocate memory on the GPU and copy the data.

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()

Add Kernel

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

- Who copies these pointers?

Main function for Add (1)

```
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;
```

Main function for Add (2)

```
    // 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;
}
```

Parallelism?

- 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

Addition on Device

- Running in parallel but how can we actually use it?
  - Modify the kernel function

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

- Each parallel execution of add is referred to as a block.
  - A set of blocks is referred to a grid
  - Each execution will have a `blockIdx.x`

Main function for Parallel Add (1)

```
#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);

    // 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 values on host.
    a = (int *) malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

Main function for Add (2)

```
    // 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 location:
  - Device
  - Host
- Use __global__ to define device code
- Must pass data between Device and Host
- Can launch parallel kernels with blocks.
  - Modify kernel function to look at blockIdx.x

Cuda Threads

- Each block can be divided in threads
- Instead of using blocks, use threads

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

Note: threadIdx instead of blockIdx

Main - Parallel Add Threads (1)

```
#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);

    // 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 values on host.
    a = (int *) malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

Main - Parallel Add Threads (2)

```
    // 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;
}
```

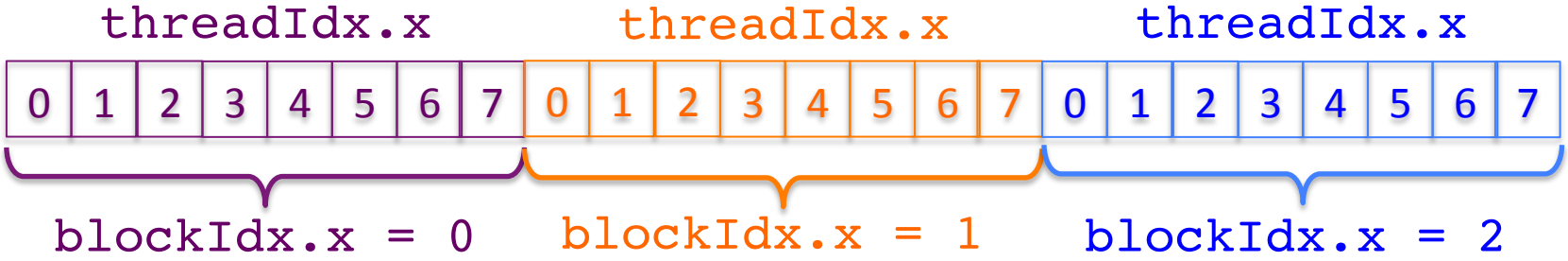
Combining Blocks and Threads

- Parallel Vector Addition with…
  - Blocks
  - Threads
- Why not use both?

- Why would be use both?
  - Good question… we'll cover this in a bit

Indexing using Blocks + Threads

- No longer as simple as just using blockIdx.x and threadIdx.x
  - Consider if we have 8 threads per block

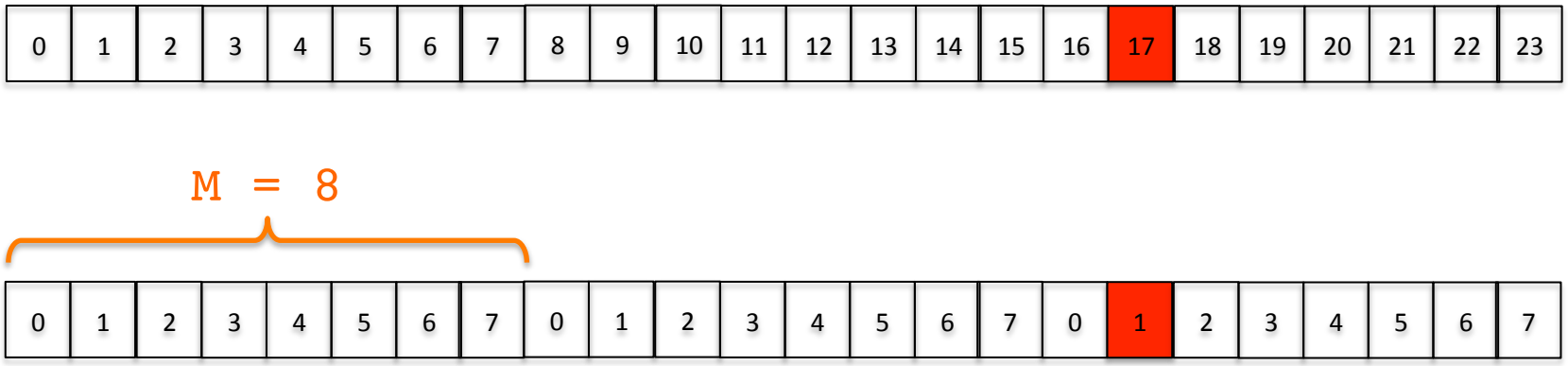


- With M threads per block.

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

Indexing using Blocks + Threads

- Which thread will operate on red element?




```
int index = threadIdx.x + blockIdx.x * M
          =       1     +      2      * 8
          = 17
```

Indexing using Blocks + 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()?

Main - Parallel Add Threads (1)

```
#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);

    // 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 values on host.
    a = (int *) malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

Main - Parallel Add Threads (2)

```
    // 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<<<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;
}
```

Arbitrary Sizes

- 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);
```

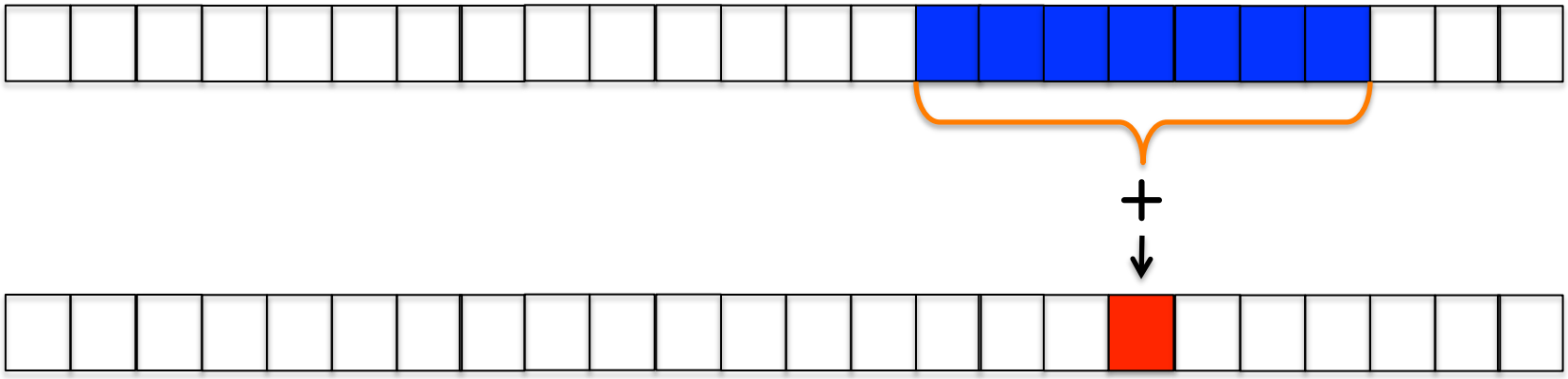
Why Threads?

- Threads seem unnecessary
  - They add a level of complexity
  - What do we gain?
- Unlike parallel blocks, threads have mechanisms to efficiently:
  - Communicate
  - Synchronize

Cooperating Threads

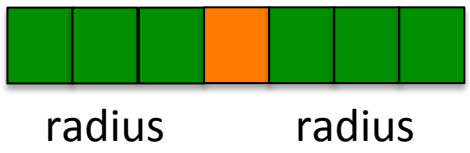
1D Stencil

- Consider applying a 1D stencil to a 1D array of elements
  - Each output element is the sum of input elements within a radius
- If radius is 3, then each output element is the sum of 7 input elements:



- Each thread process one output element
  - blockDim.x elements per block

- With radius of 3, each element is read 7 times!

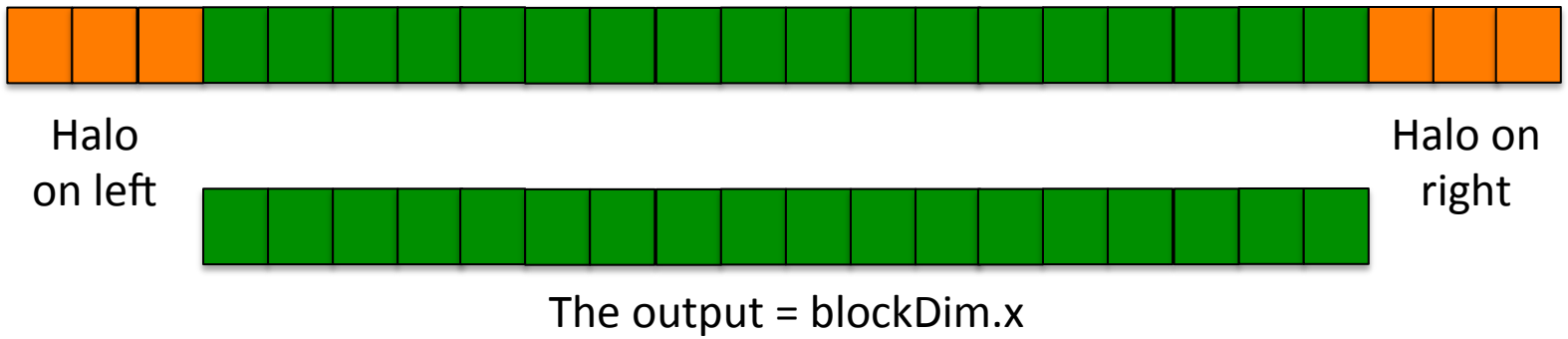


Sharing Data Between Threads

- Terminology: within a block, threads share data via shared memory
- Extremely fast on-chip memory
  - By opposition to device memory, referred to as global memory
  - Like a user-managed cache
- Declare using __shared__, allocated per block
- Data is not visible to threads in other blocks

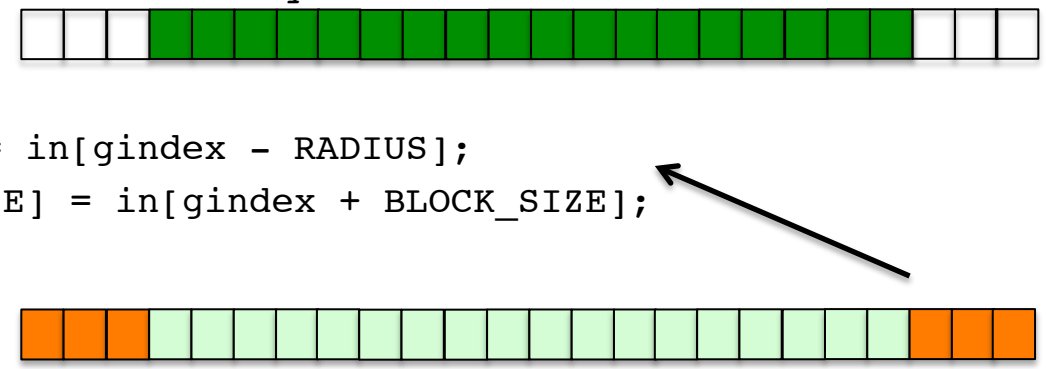
Implementing Sharing

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



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];
    }
 …… function continues
```



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

Race To Data!

- The stencil example will not work…
- Suppose thread 15 reads the halo before thread 0 has fetched it...

__syncthreads()

- void __syncthreads();

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

Stencil Kernel with sync

```
__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];
    }

    __syncthreads();

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

Managing Device

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

Reporting Errors

- All CUDA API calls return an error code (`cudaError_t`)
  - Error in the API call itself
  - OR
  - Error in an earlier asynchronous operation (e.g. kernel)
- Get the error code for the last error:

```
cudaError_t cudaGetLastError(void)
```

- Get a string to describe the error:

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

Error Checking

```
void CheckCudaError() {
  cudaError_t err = cudaGetLastError();
  if (err != cudaSuccess) {
    printf("Error: %s\n", cudaGetErrorString(err));
    exit(-1);
  }
}
```

Limits

- Block Size limited to $(2^{31} - 1)$
  - Older versions < 3.0 limited to 65535
  - As of writing this default nvcc is not 3.0+, so you need to compile with:
    `nvcc add.cu –o add` **`--gpu-architecture=sm_30`**
- Max threads per block 1024.
- If you go over these limits very strange things occur ☺