KOÇ UNIVERSITY

CUDA Concepts

Didem Unat

COMP 429/529

Parallel Programming

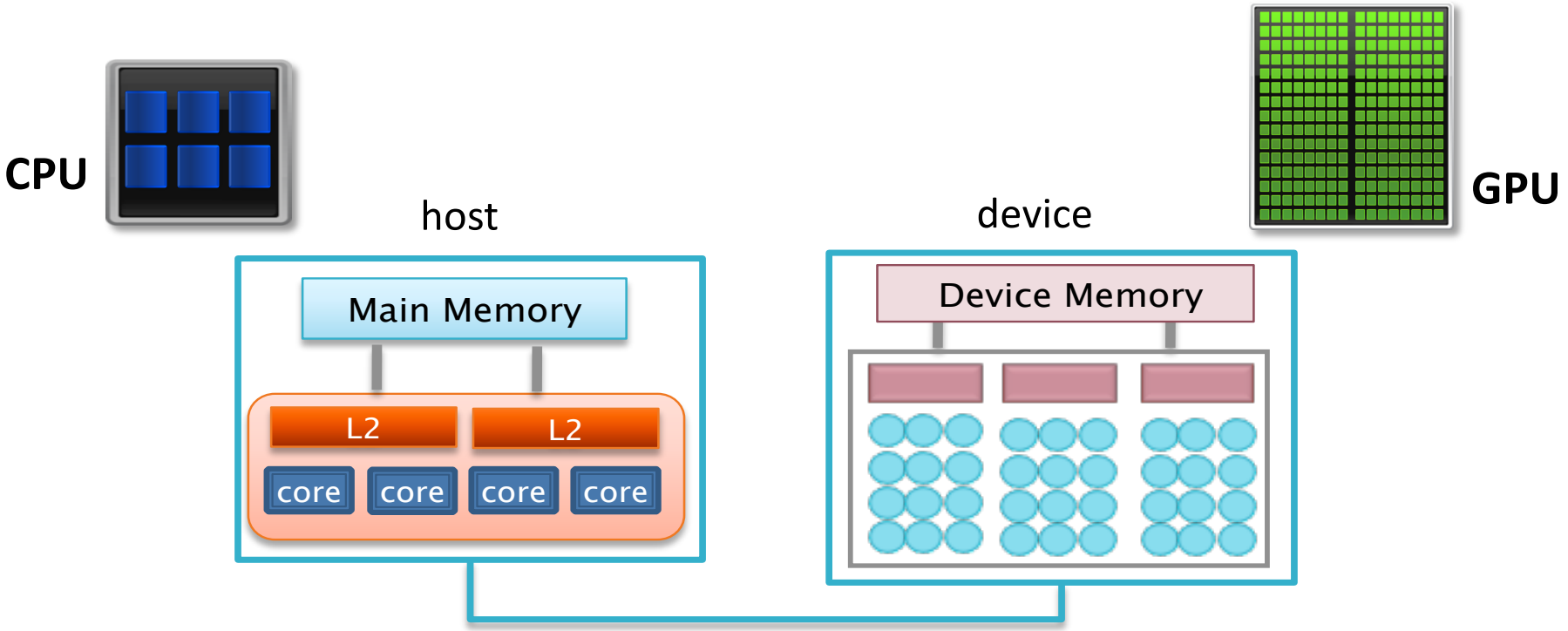
- Heterogeneous Computing
- Kernels
- Memory Management
- Threads
- Thread Blocks
- Memory Hierarchy
- Shared Memory
- Handling errors
- Managing devices

CUDA

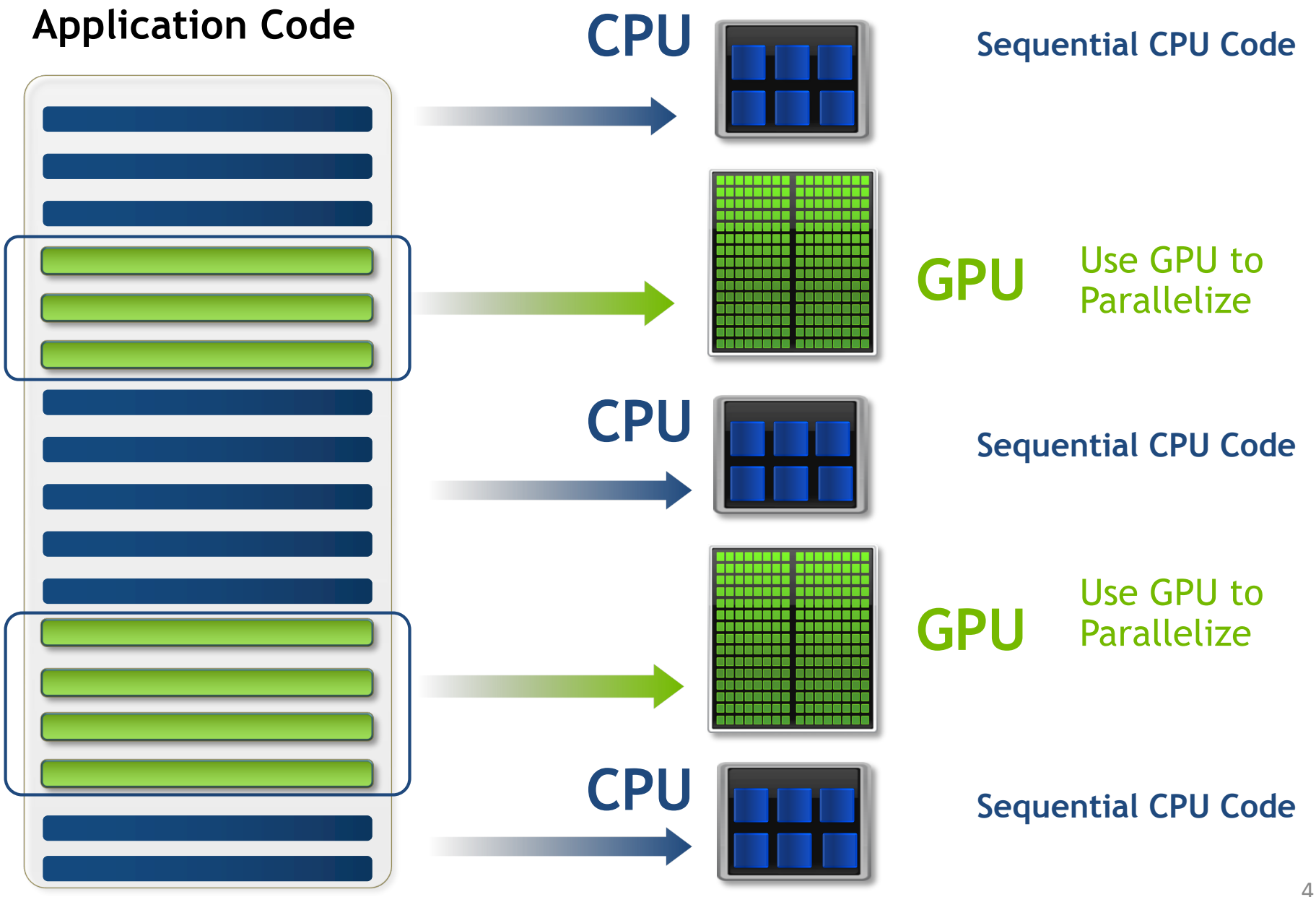
- CUDA Architecture
  - Expose GPU parallelism for general-purpose computing
  - Retain performance
- CUDA C/C++
  - Based on industry-standard C/C++
  - Small set of extensions to enable heterogeneous programming
  - Straightforward APIs to manage devices, memory etc.

Heterogeneous Computing

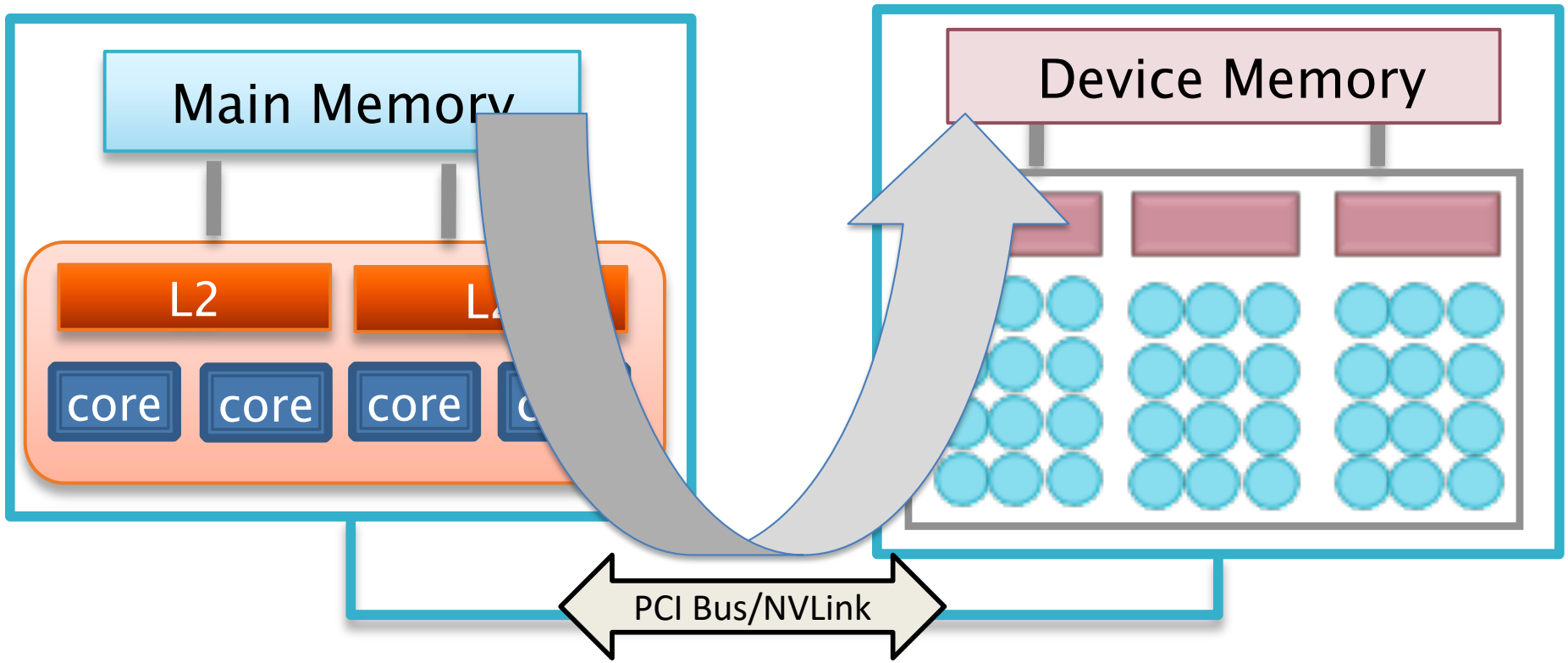
- Terminology:
  - *Host* The CPU and its memory (host memory)
  - *Device* The GPU and its memory (device memory)



Heterogeneous Programming

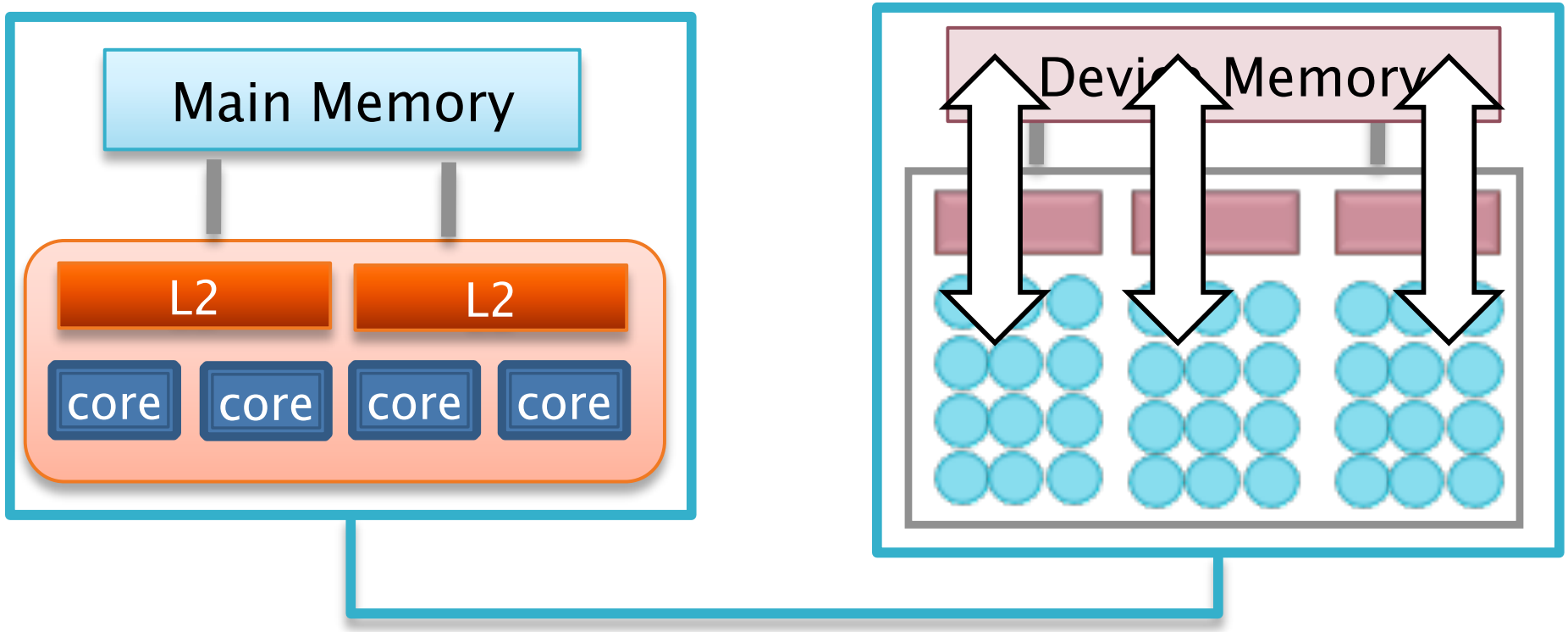


Simple Processing Flow



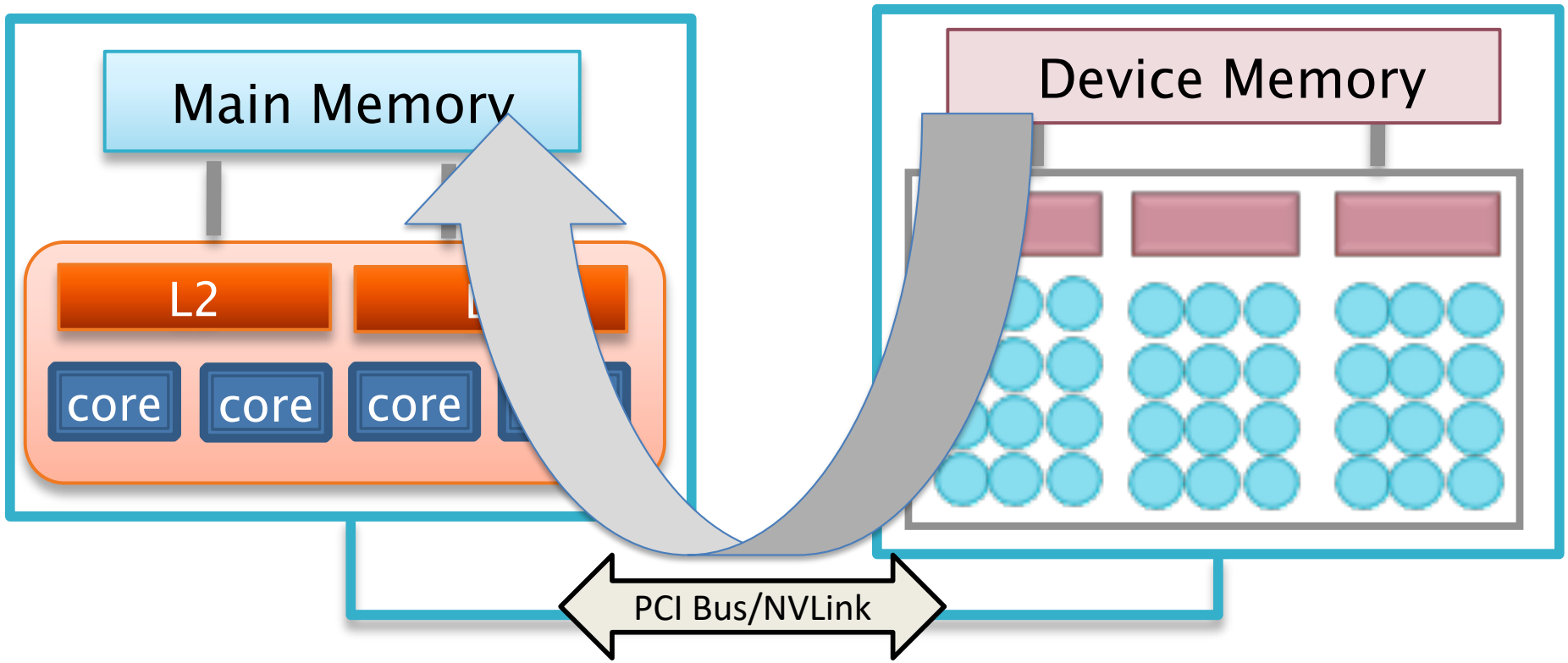
1. Copy input data from CPU memory to GPU memory

Simple Processing Flow



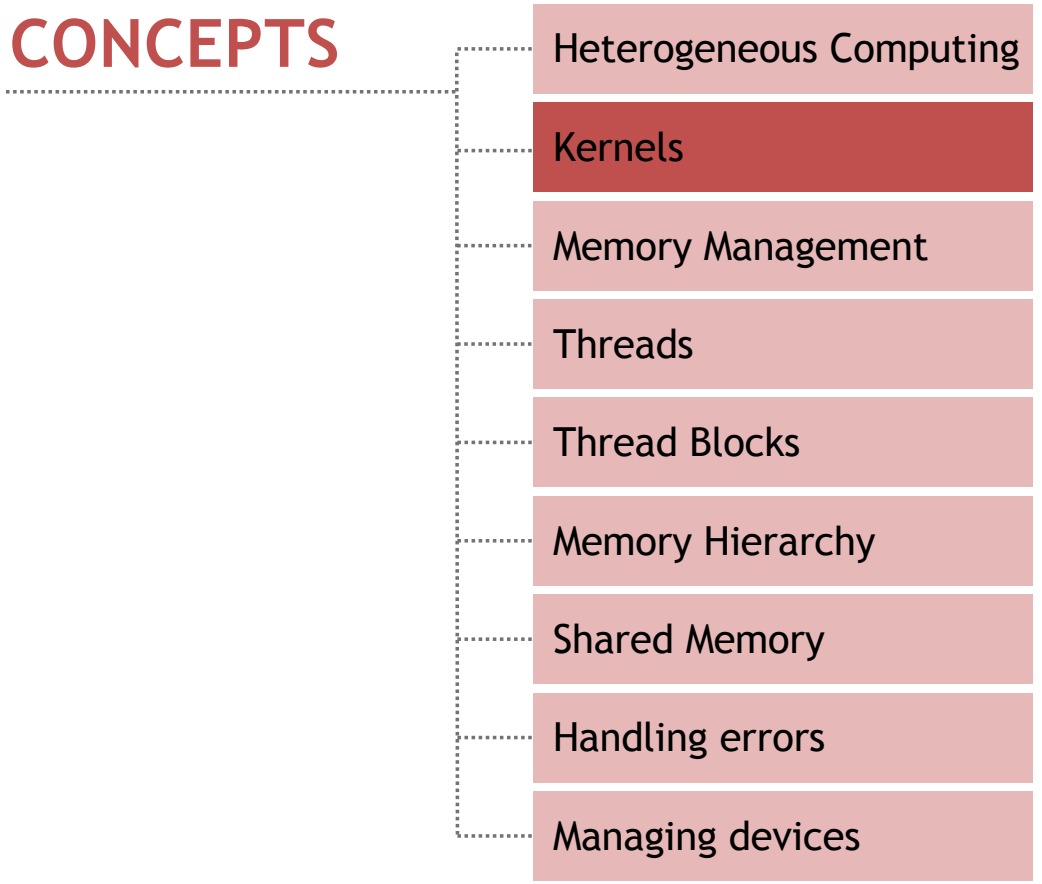
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute on GPU's streaming multiprocessors

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute on GPU's streaming multiprocessors
3. Copy results from GPU memory to CPU memory

CONCEPTS


Heterogeneous Computing
Kernels
Memory Management
Threads
Thread Blocks
Memory Hierarchy
Shared Memory
Handling errors
Managing devices

Hello World!

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

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

Compiling:

```
$ nvcc hello.cu
```

Running:

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

Hello World with Device Code

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

- CUDA C/C++ keyword `__global__` indicates a function that:
  - Runs on the device
  - Called from host code
- `nvcc` separates source code into host and device components
  - Device functions (e.g. `hello()`) processed by NVIDIA compiler
  - Host functions (e.g. `main()`) processed by standard host compiler

Hello World with Device Code

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

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

Compiling:

```
$ nvcc hello.cu
```

Running:

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

`hello()` does nothing, somewhat anticlimactic!

Lab2- Hello

- Copy the labs from /kuacc/users/dunat/COMP429/GPU/Hands-on-Labs/
- A very simple Hello World to demonstrate threads in a GPU.
- Every thread prints one character of the message

```
__device__ const char *STR = "HELLO WORLD!";
const char STR_LENGTH = 12;

__global__ void hello()
{
    //every thread prints one character
    printf("%c\n", STR[threadIdx.x % STR_LENGTH]);
}
int main(void)
{
    hello<<< 1, 12>>>();
    cudaDeviceSynchronize();

    return 0;
}
```

Lab2-Hello (cont.)

Instructions:

1. Compile and run the code
2. Comment out the first part of the code and Comment in the second part of the code
3. Compile and Run, what do you observe?
4. Change 12 to 16, what happens?
   - hello<<< 1, 16>>>();
5. Change 1 to 2, what happens?
   - hello<<< 2, 12>>>();
6. Change both, what happens?
   - hello<<< 2, 16>>>();

To get a GPU in an interactive session KUACC

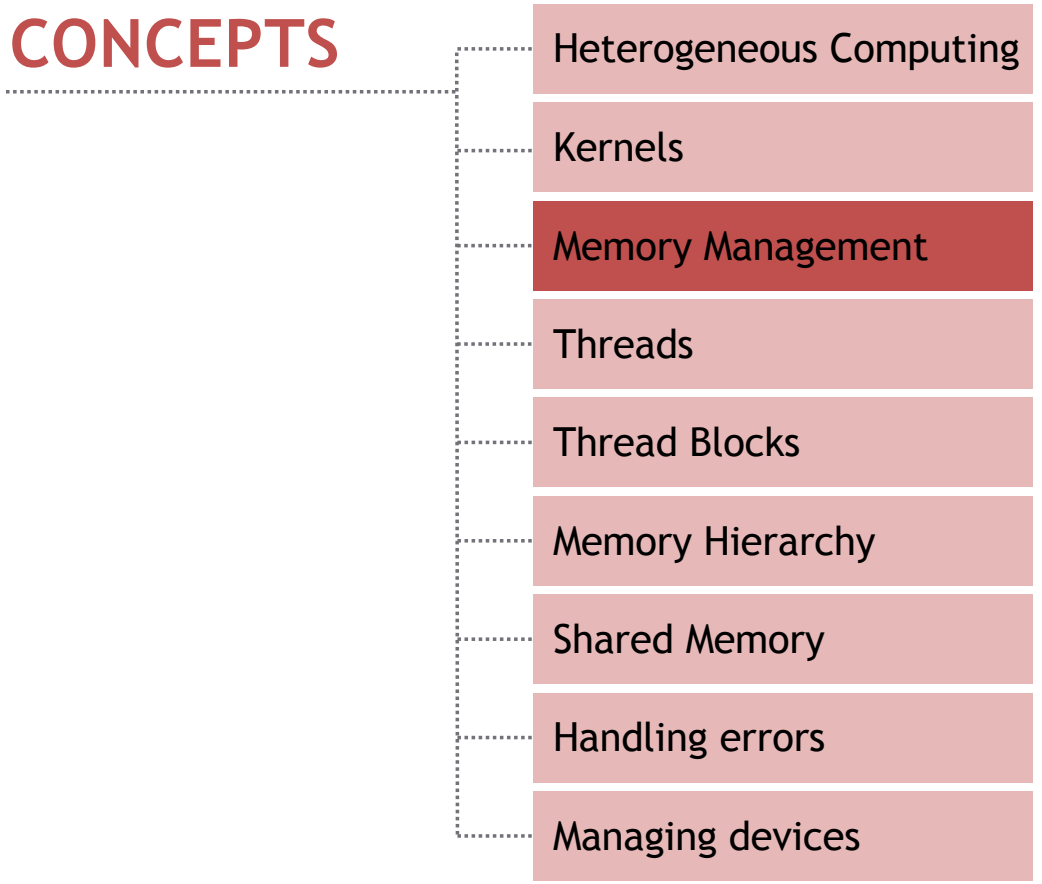
srun -N 1 -n 1 --gres=gpu:1 -p short --time=00:30:00 --pty bash

Lab2-Hello (cont.)

Observations:

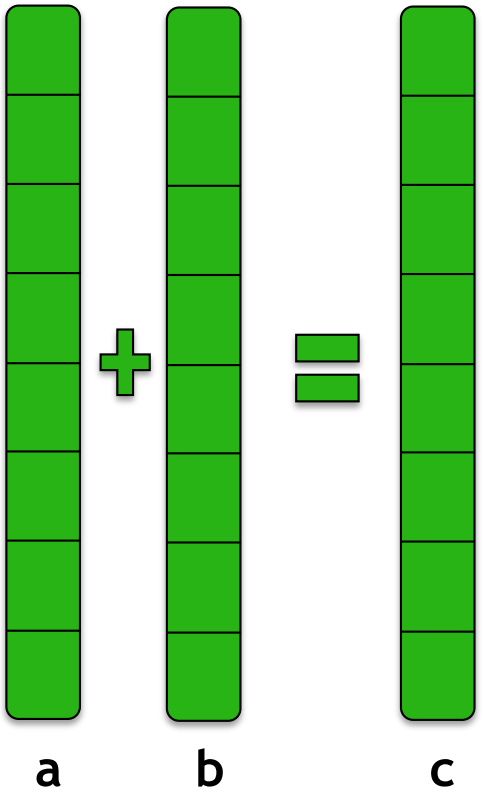
1. Compile and run the code
    1. CPU prints hello world but GPU does nothing.
2. Comment out the first part of the code and comment in the second part of the code
3. Compile and Run, what do you observe?
    1. This prints hello world – one character in each line
4. Change 12 to 16, what happens?
    1. There are more threads than characters in the string, some characters are printed twice!
5. Change 1 to 2, what happens?
    1. Two thread blocks are created, the message is printed twice
6. Change both, what happens?
    1. Both 4 and 5 occur, some characters are printed twice, and the message is printed twice.

CONCEPTS


Heterogeneous Computing
Kernels
Memory Management
Threads
Thread Blocks
Memory Hierarchy
Shared Memory
Handling errors
Managing devices

More Interesting Example

- We'll start by adding two integers and build up to vector addition
- Learn how to
  - Write a simple CUDA kernel
  - Launch kernel with parameters
  - Manage host and device memories
    - Device Memory Allocation
    - Host-Device Memory Copy



Scalar Add 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 that we use pointers for the variables

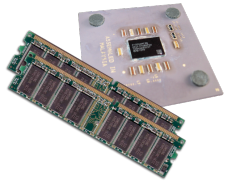
```
__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}
```

- `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()`

Memory Allocation in `main()`

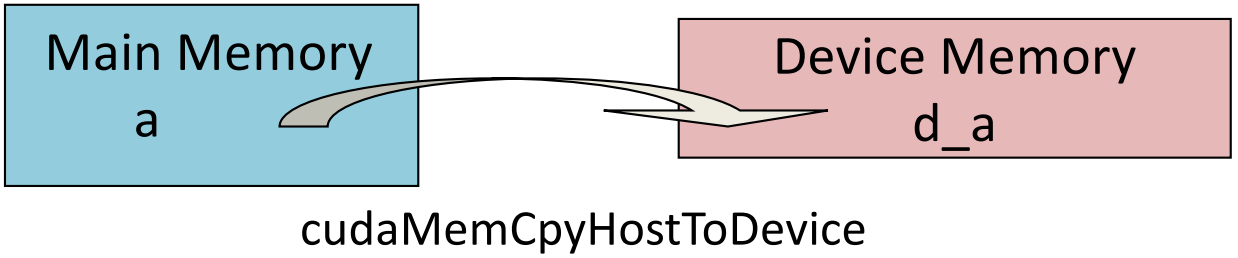
```
int main(void) {
    int a, b, c;                // host copies of a, b, c
    int *d_a, *d_b, *d_c;       // device copies of a, b, c
    int size = sizeof(int);

    // 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;
```

Main Memory
a, b, c

Device Memory
d_a, d_b, d_c

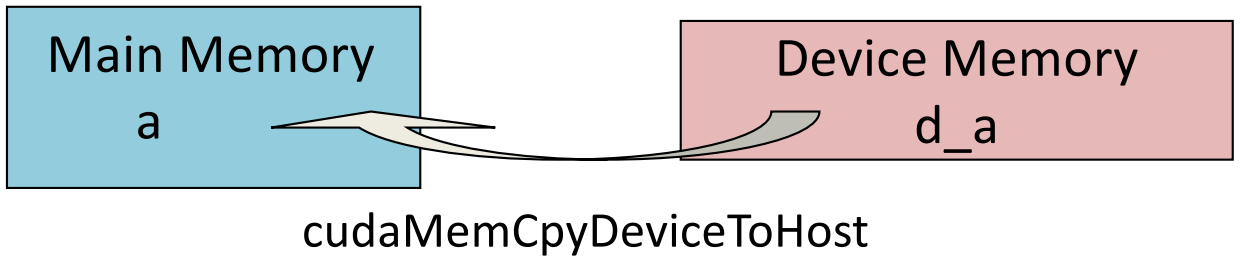
Memory Copy

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



Memory Copy

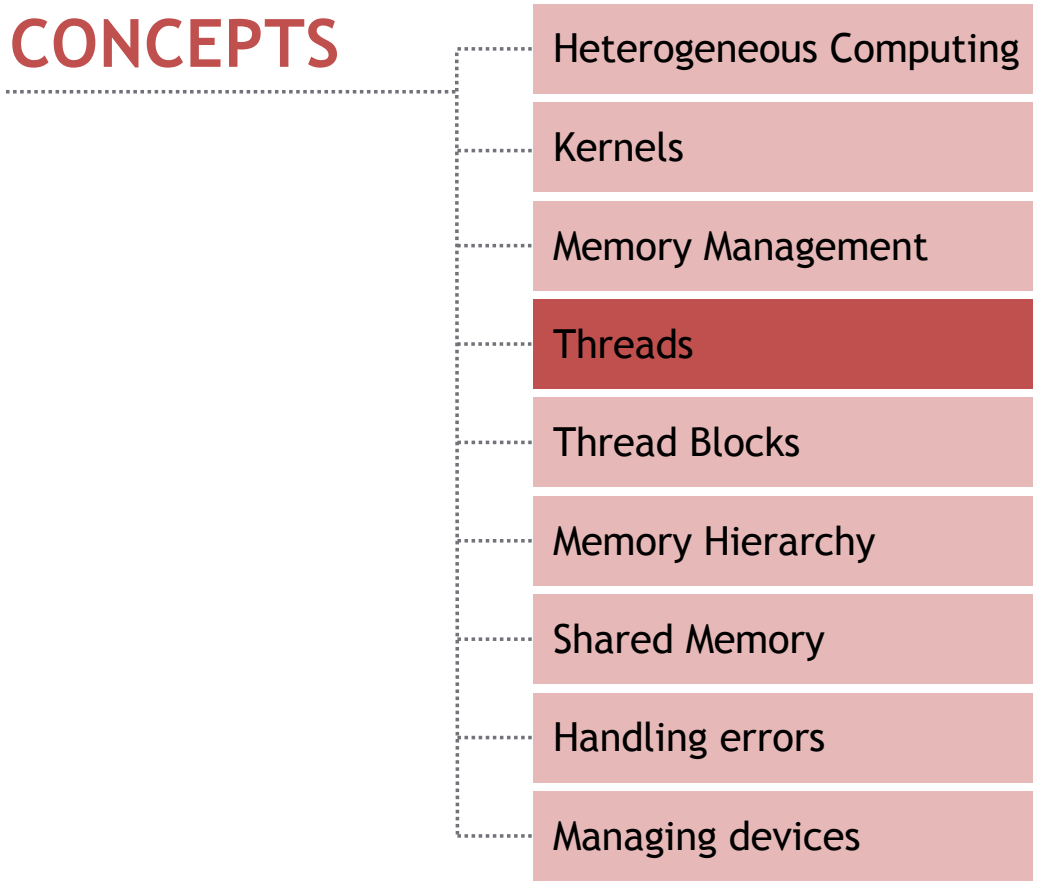
```
// Copy inputs to device
cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

// 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);
        . . .
```



Scalar Add

- Using one ALU out 64 ALUs in one SM, out of 84 SMs on a GPU
- Need to use all ALU in all SMs.

CONCEPTS


Heterogeneous Computing
Kernels
Memory Management
Threads
Thread Blocks
Memory Hierarchy
Shared Memory
Handling errors
Managing devices

Moving to Parallel

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

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

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

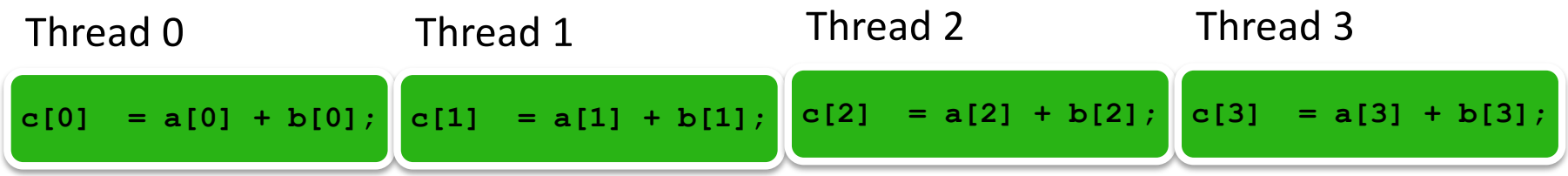
- Instead of executing `add()` once, execute N times in parallel

CUDA Threads

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

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

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



Vector Addition Using Threads

```
#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);
```

Vector Addition Using Threads

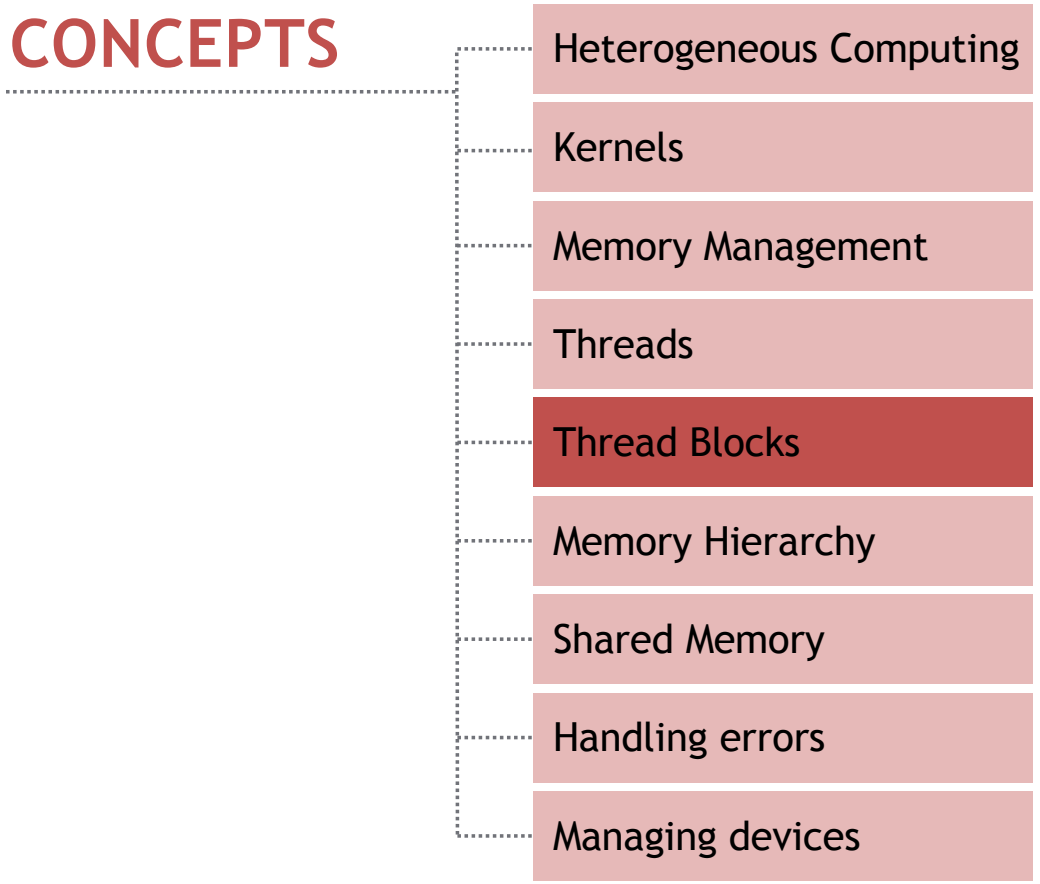
```
        // Copy inputs to device
        cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

        // 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;
    }
```

Vector Add

- Using **ALL** ALU out 64 ALUs in one SM, out of 84 SMs on a GPU
- Need to use all ALU in all SMs.

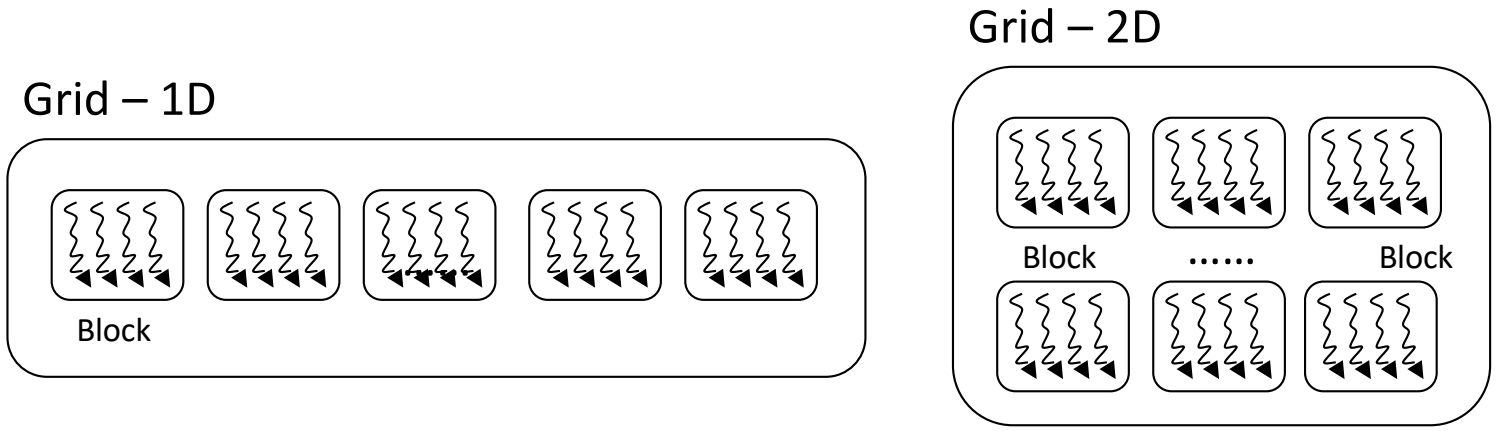
SM

CONCEPTS

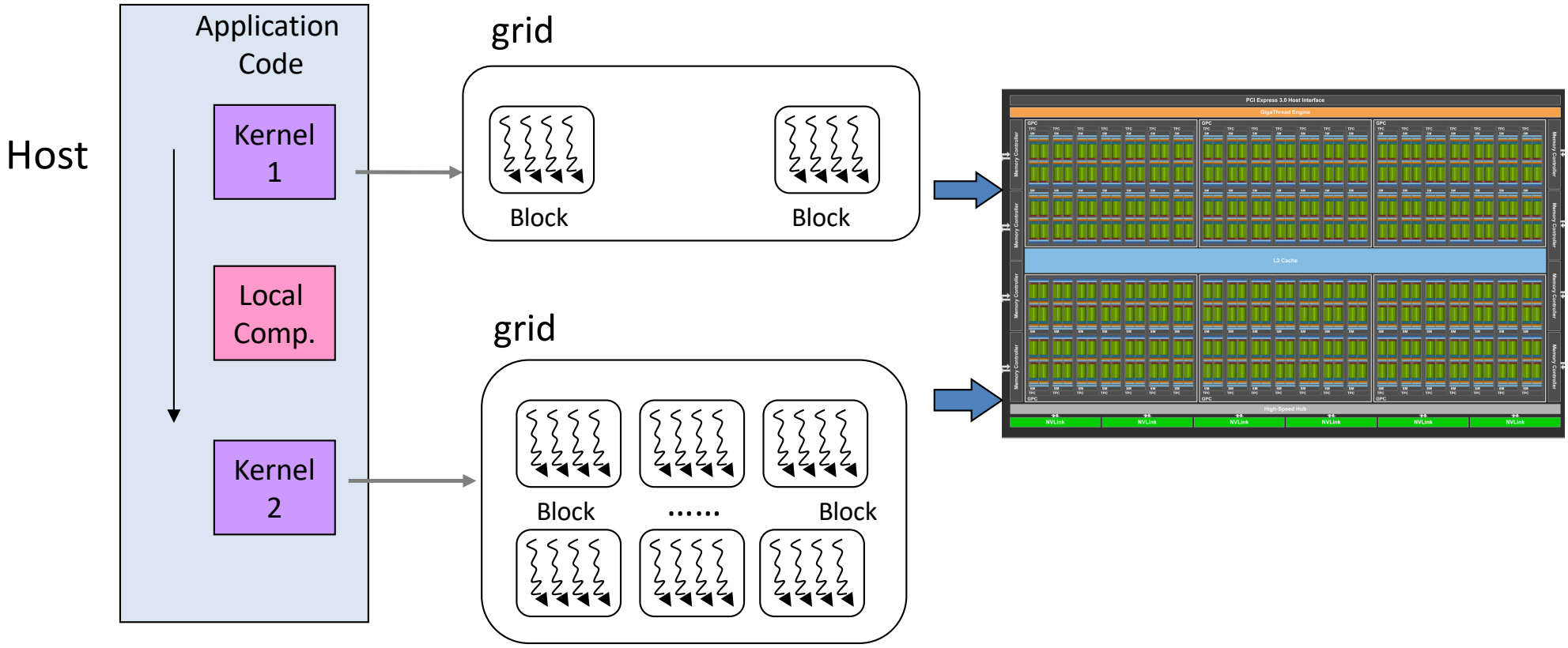

Heterogeneous Computing
Kernels
Memory Management
Threads
Thread Blocks
Memory Hierarchy
Shared Memory
Handling errors
Managing devices

Thread Blocks

- Threads can be organized into **blocks of threads** that **execute independently** from each other
- There is a hierarchy of threads
  - Each kernel has a grid containing multiple thread blocks, which contains multiple threads.
- Thread blocks can be either 1D, 2D or 3D

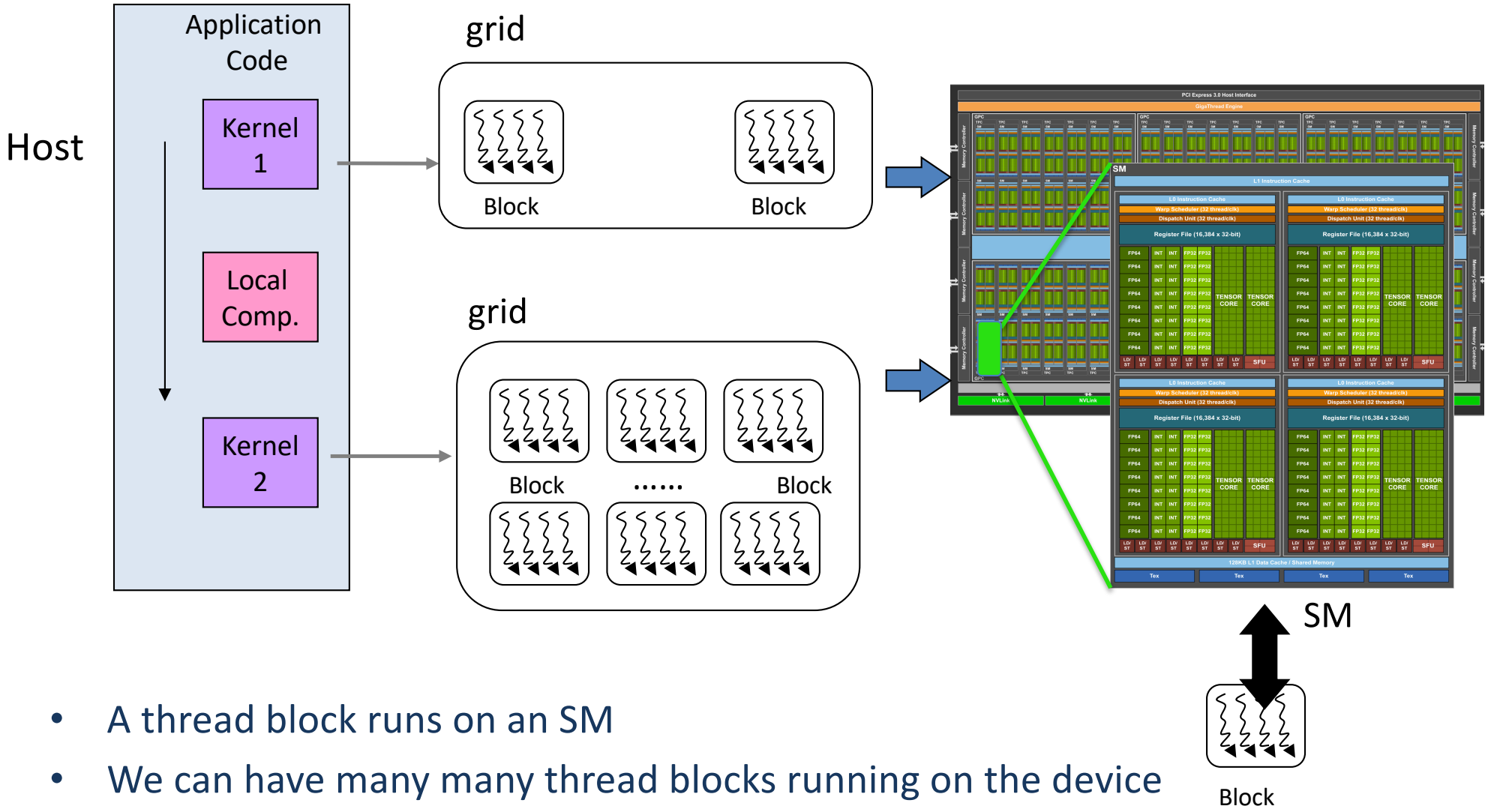


CUDA Thread Blocks



- To keep the SMs busy on the GPU, a kernel
  - Generates 1000s of threads
  - Organized as "**thread blocks**"
  - Geometry of each block can be different, depending on the problem

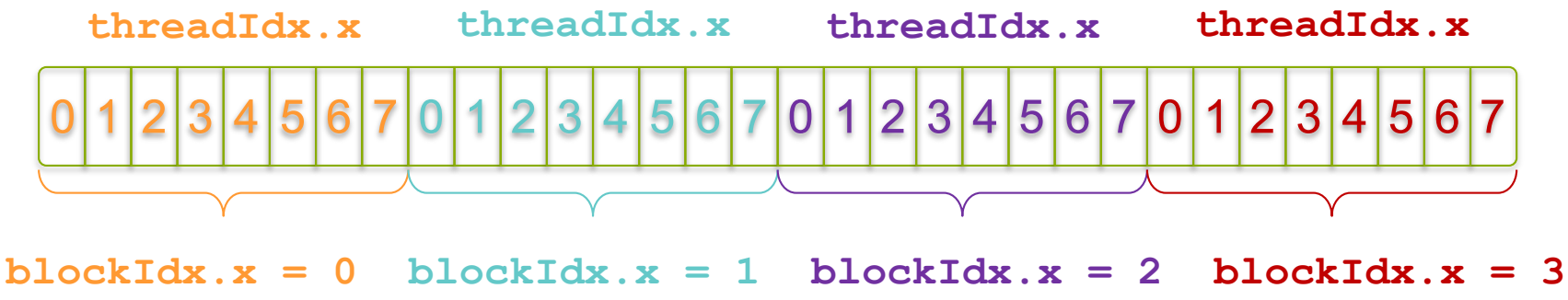
CUDA Thread Blocks



- A thread block runs on an SM
- We can have many many thread blocks running on the device simultaneously

Indexing Arrays with Thread Blocks

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

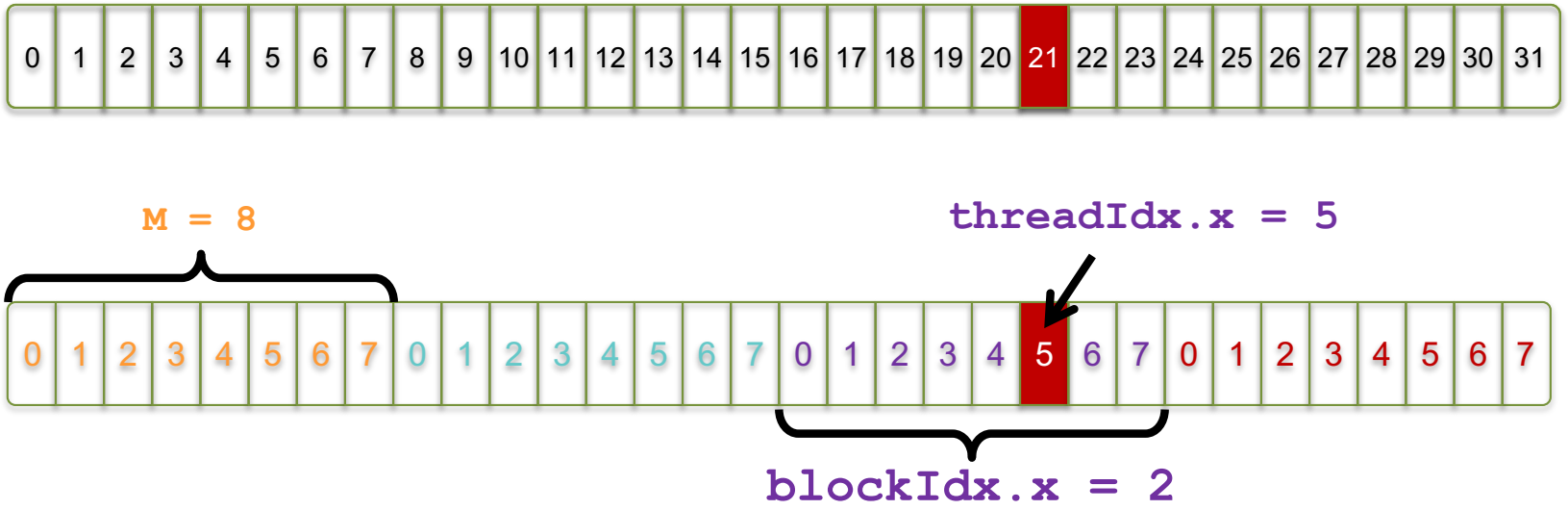


- With M threads/block a unique index for each thread is given by:

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

Indexing Arrays: Example

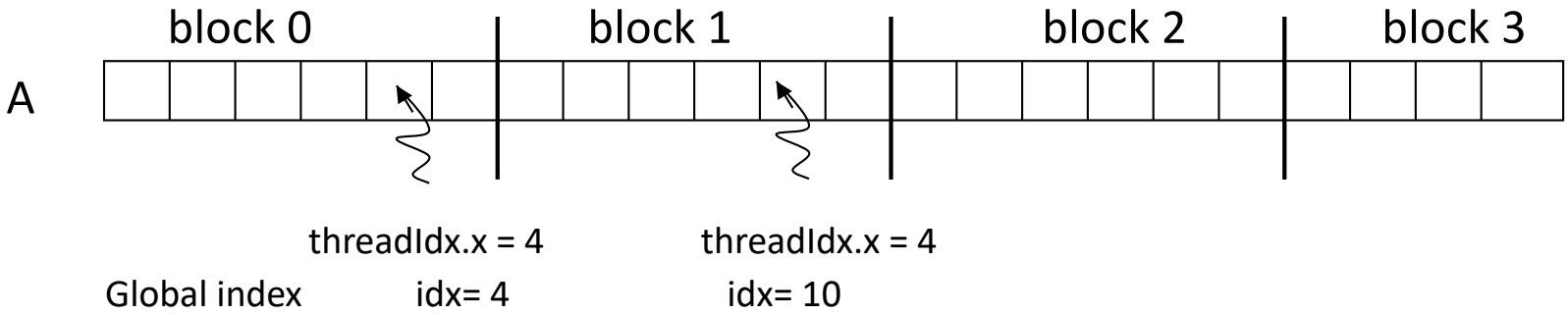
- Which thread will operate on the red element?




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

Thread Assignment

- Each thread adds one element in A and B into C
- A thread uses *block Id, block size and thread Id* parameters to determine its workload.
  - These IDs are unique provided by CUDA runtime




```
__global__ void add(int *a, int *b, int *c) {

    int index = threadIdx.x + blockIdx.x * blockDim.x;
    c[index] = a[index] + b[index];
}
```

Vector Addition with Thread Blocks

```
#define N (2048*2048)
#define nThreads 512  //threads per block
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);
```

Vector Addition with Thread Blocks

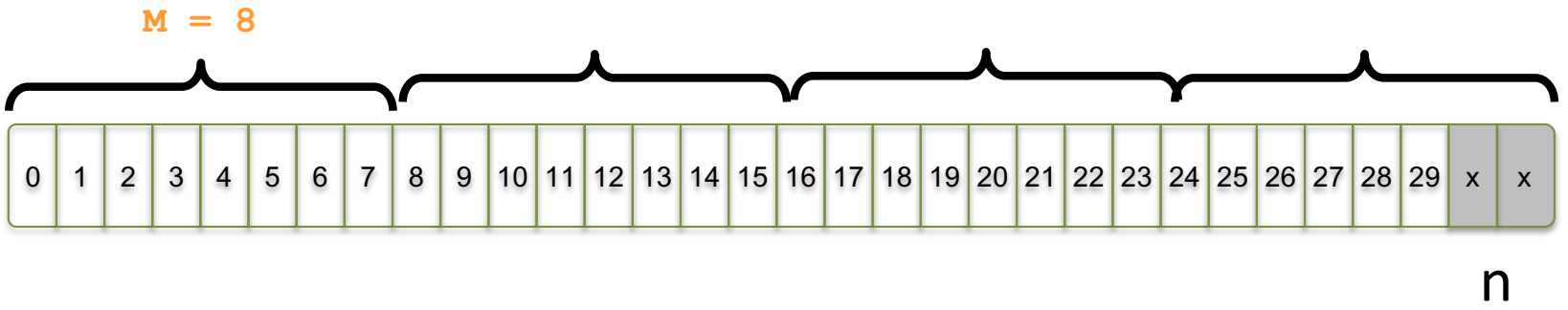
```
        // Copy inputs to device
        cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

        // Launch add() kernel on GPU with thread blocks
        add<<<N/nThreads,nThreads>>>(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 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];
}
```



Vector Addition

```
void add_on_host(int* a, int* b, int* c, int n)
{
     int i;
     for (i=0; i< n; i++)
         c[i] = a[i] + b[i];
}
```

On Host

```
__global__ void add_on_GPU(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];
}
```

On GPU

Kernel Launch

Function Call:

```
add_on_host(a, b, c, n);
```

```
void add_on_host(int* a, int* b, int* c, int n)
{
     int i;
     for (i=0; i< n; i++)
         c[i] = a[i] + b[i];
}
```

On Host

Kernel Call:

```
add_on_GPU<<<nBlocks, nThreads>>> (a_d, b_d, c_d, n);
```

```
__global__ void add_on_GPU(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];
}
```

On GPU

Kernel Launch on the Host

```
//set block size
#define nThreads  512

. . .

//compute number of blocks
int nBlocks = N/nThreads + (N % nThreads == 0 ? 0 : 1);
```

```
//kernel launch
add_on_GPU <<< nBlocks, nThreads >>> (a_d, b_d, c_d, N);
```

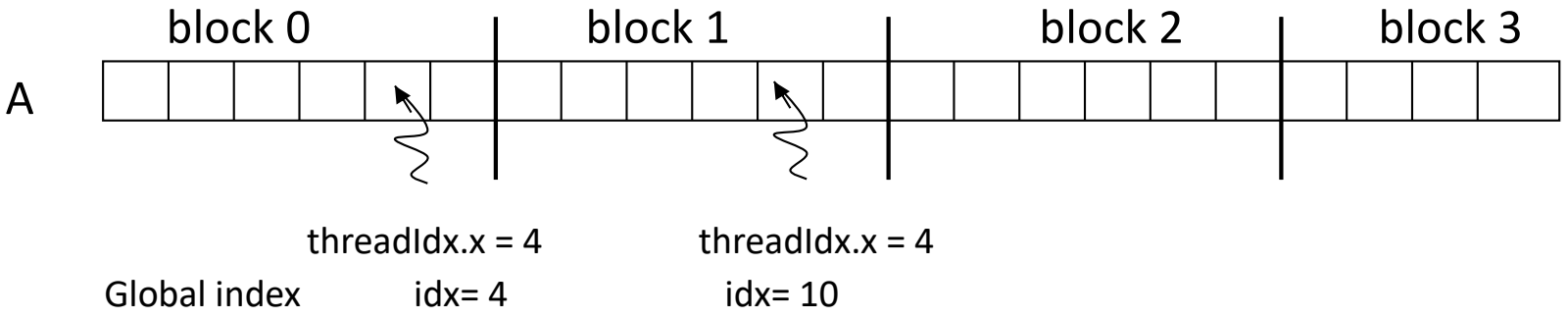
Lab3-VecAdd

- Code
  - Vector addition on the GPU
  - Allocate three arrays on the host
  - Allocate three arrays on the device
  - Copy the content of first two arrays to the device
  - Perform vector addition
  - Copy the result back to host
  - Deallocate the host and device memory

- In several places in the code, it says **FIXME.**
- You need to add the necessary code to those places

CUDA Thread Block Overview

- All threads in a block execute the same kernel program (SPMD)
- Programmer declares block:
  - Block size **1 to 1024** concurrent threads
  - Block shape 1D, 2D, or 3D
- Threads have thread id numbers within block
  - Thread program uses thread id to select work and address shared data
- Threads in different blocks cannot cooperate
  - Each block can execute in any order relative to other blocks!



Review (1 of 2)

- 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

Review (2 of 2)

- Basic device memory management
  - `cudaMalloc()`
  - `cudaMemcpy()`
  - `cudaFree()`

- Launching parallel kernels
  - Launch `N` copies of `add()` with `add<<<nBlocks,nThreads>>>(…);`
  - Use `blockIdx.x` to access block index
  - Use `threadIdx.x` to access thread index

Compiling CUDA

- Any source file containing CUDA language code must be compiled with `nvcc`
  - It separates code running on the host from the code running on the GPU
  - CUDA is compiled into PTX instructions, and the graphics driver contains a compiler which translates the PTX instructions into a binary code
  - PTX (Parallel Thread eXecution)
    - a low-level *parallel thread execution* virtual machine and ISA

Acknowledgments

- These slides are inspired and partly adapted from
  - Programming Massively Parallel Processors: A Hands On Approach, available from *http: David Kirk and Wen-mei Hwu, February 2010, Morgan Kaufmann Publishers, ISBN 0123814723.*
  - NVidia, *CUDA Programming Guide,* available from *http://docs.nvidia.com/cuda/cuda-c-programming-guide*
  - Why GPU Computing? By Mark Ebersole – Nvidia
    - https://developer.nvidia.com/cuda-education
  - CS193g Spring 2010 GPU Computing Course at Stanford
    - https://github.com/jaredhoberock/stanford-cs193g-sp2010
  - CUDA Education
    - https://developer.nvidia.com/cuda-education