EBU7501: Cloud Computing
Week 2, Day 3: Advanced GPU Programming using CUDA

Dr. Gokop Goteng

Queen Mary University of London

Lecture Aim and Outcome

- Aim
  - The aim of this lecture is to teach students how to write parallel CUDA programmes that run on graphics processing unit (GPU)
- Outcome
  - At the end of this lecture students should be able to:
    - Write simple parallel CUDA application for GPUs
    - Know the concept of heterogeneous programming in GPUs using CUDA model

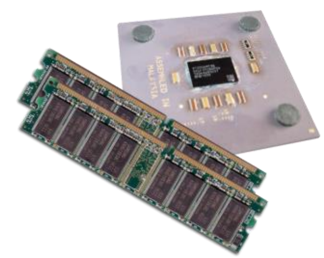
Lecture Outline

- CUDA and Heterogeneous Computing
- CUDA and Parallel Computing
- CUDA and Parallel Programming
- CUDA Parallel Programming using Vectors
- CUDA Heterogeneous Programming

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: `add()`

- Returning to our `add()` kernel

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

- Let's take a look at main()…

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;
```

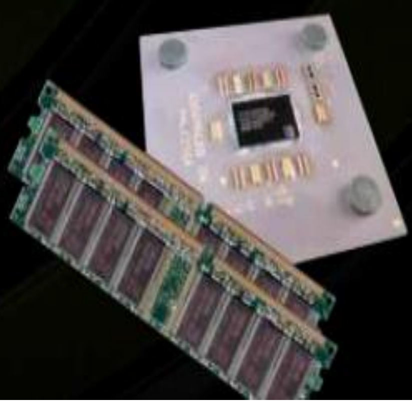
Queen Mary University of London

Addition on the Device: `main()`

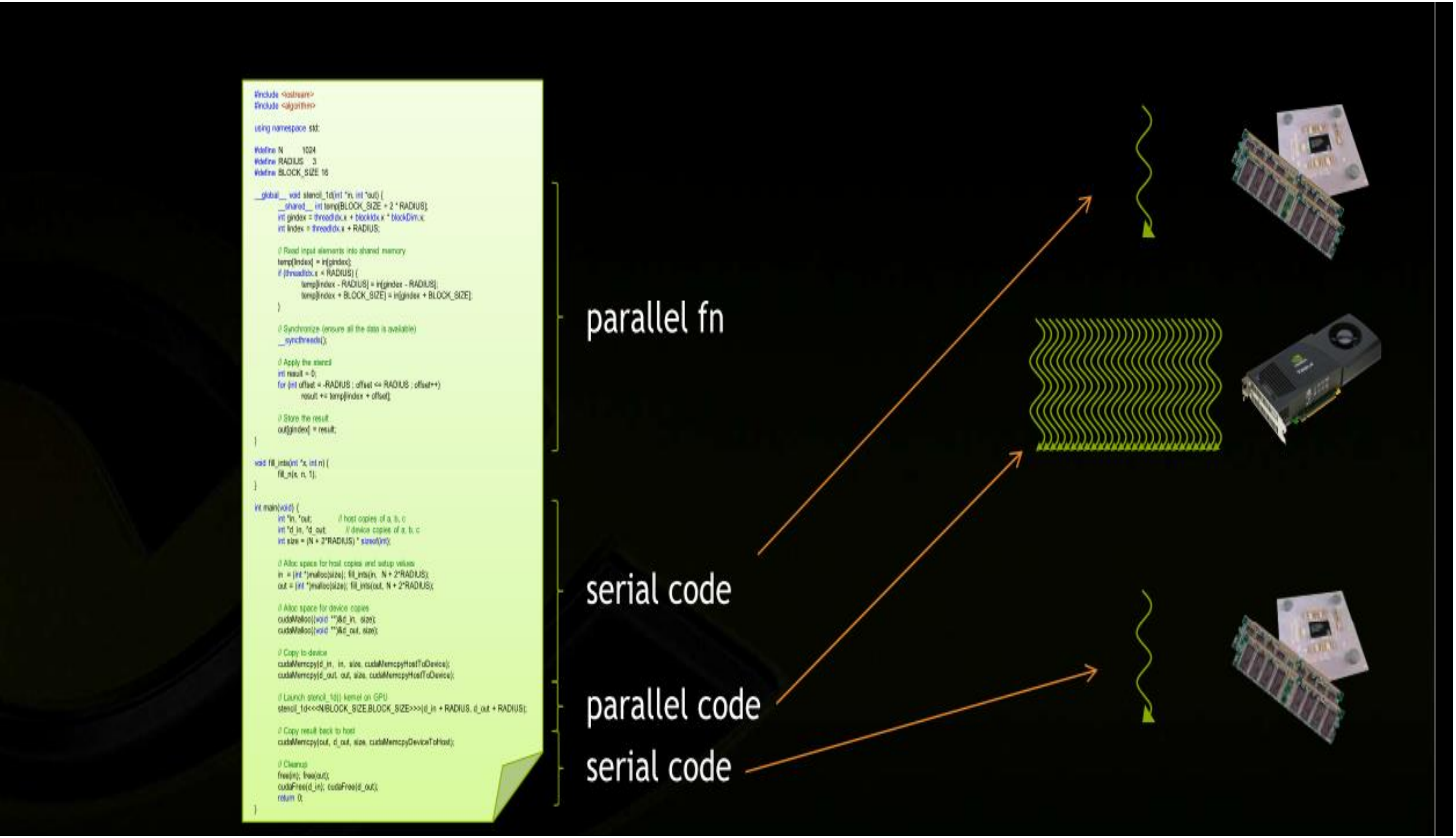
```
        // Copy inputs to device
        cudaMemcpy(d_a, &a, size,cudaMemcpyHostToDevice);
        cudaMemcpy(d_b, &b, size,cudaMemcpyHostToDevice);

        // 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;
}
```

CUDA and Heterogeneous Computing

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

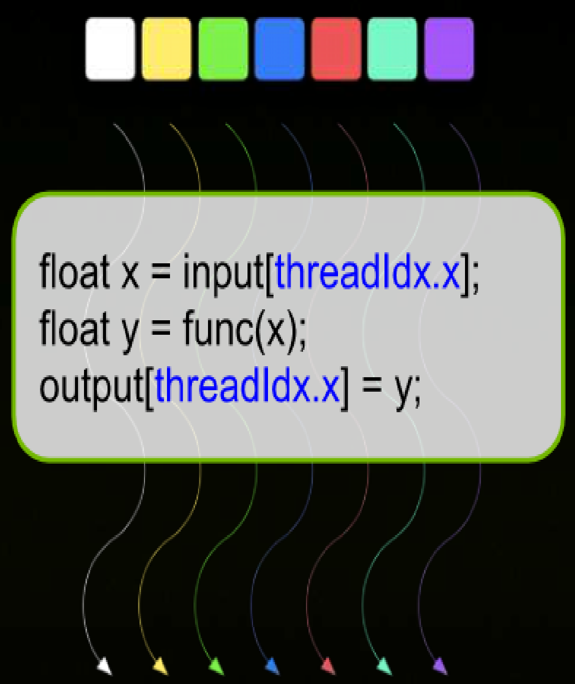


CUDA and Heterogeneous Computing



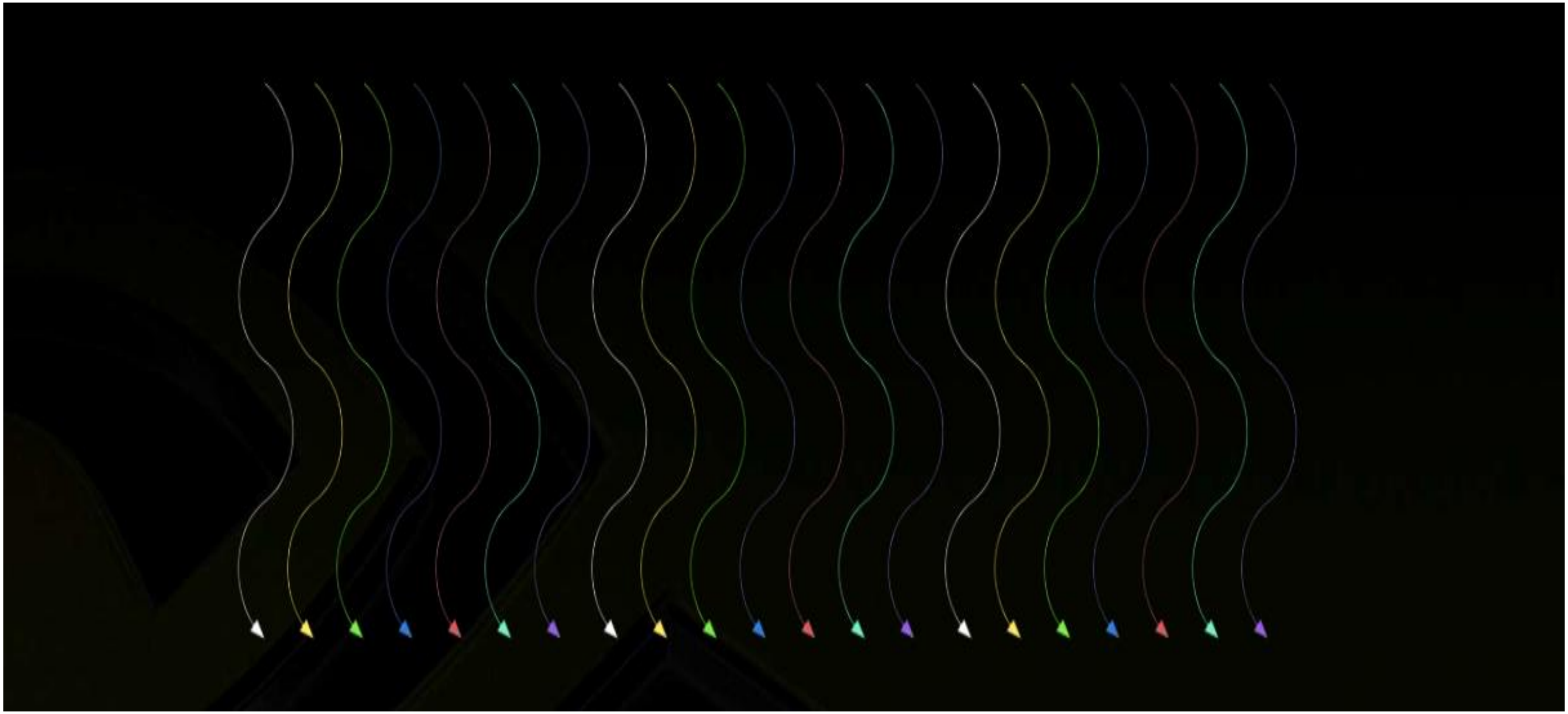
CUDA and Parallel Computing

- Parallel Threads
  - A kernel is a function executed on the GPU as an array of threads in parallel
  - All threads execute the same code, can take different paths
  - Each thread has an ID
    - Select input/output data
    - Control decisions



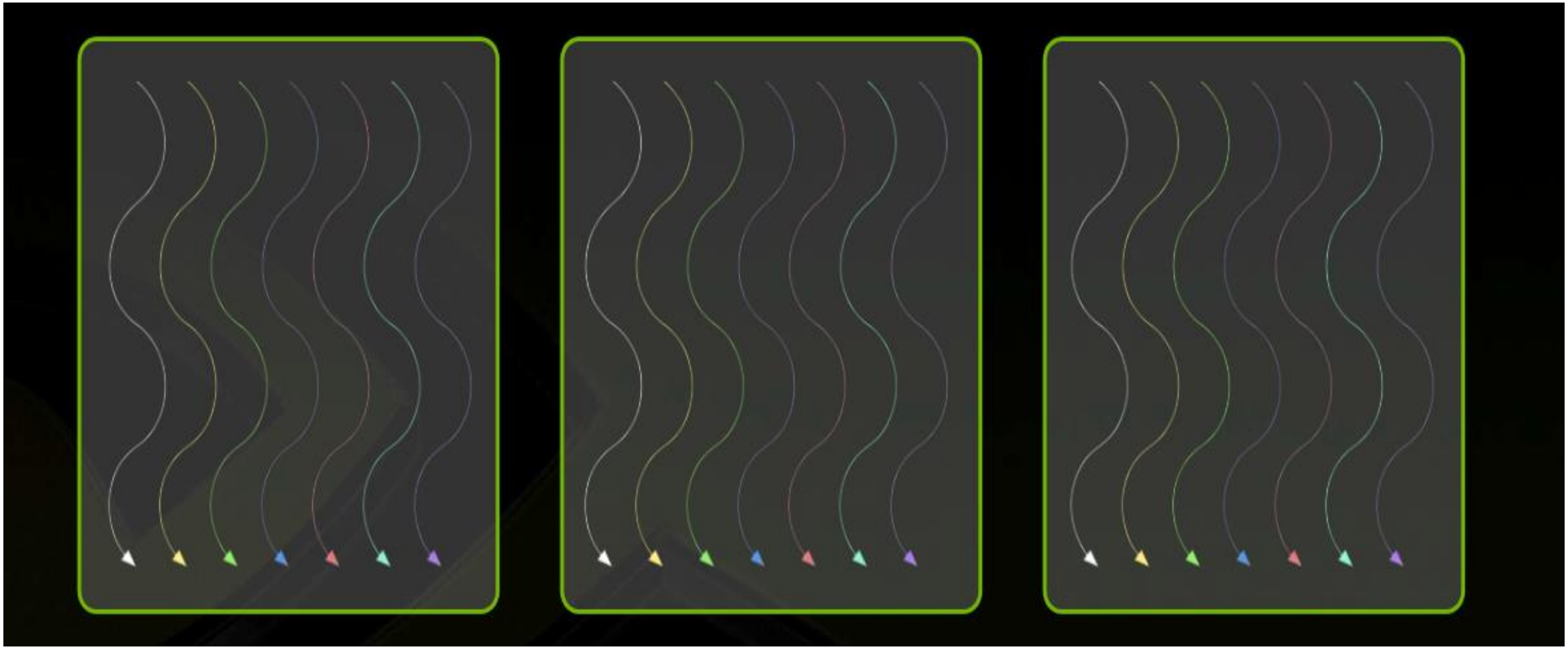
CUDA and Parallel Computing

- Sub-divide these threads into blocks



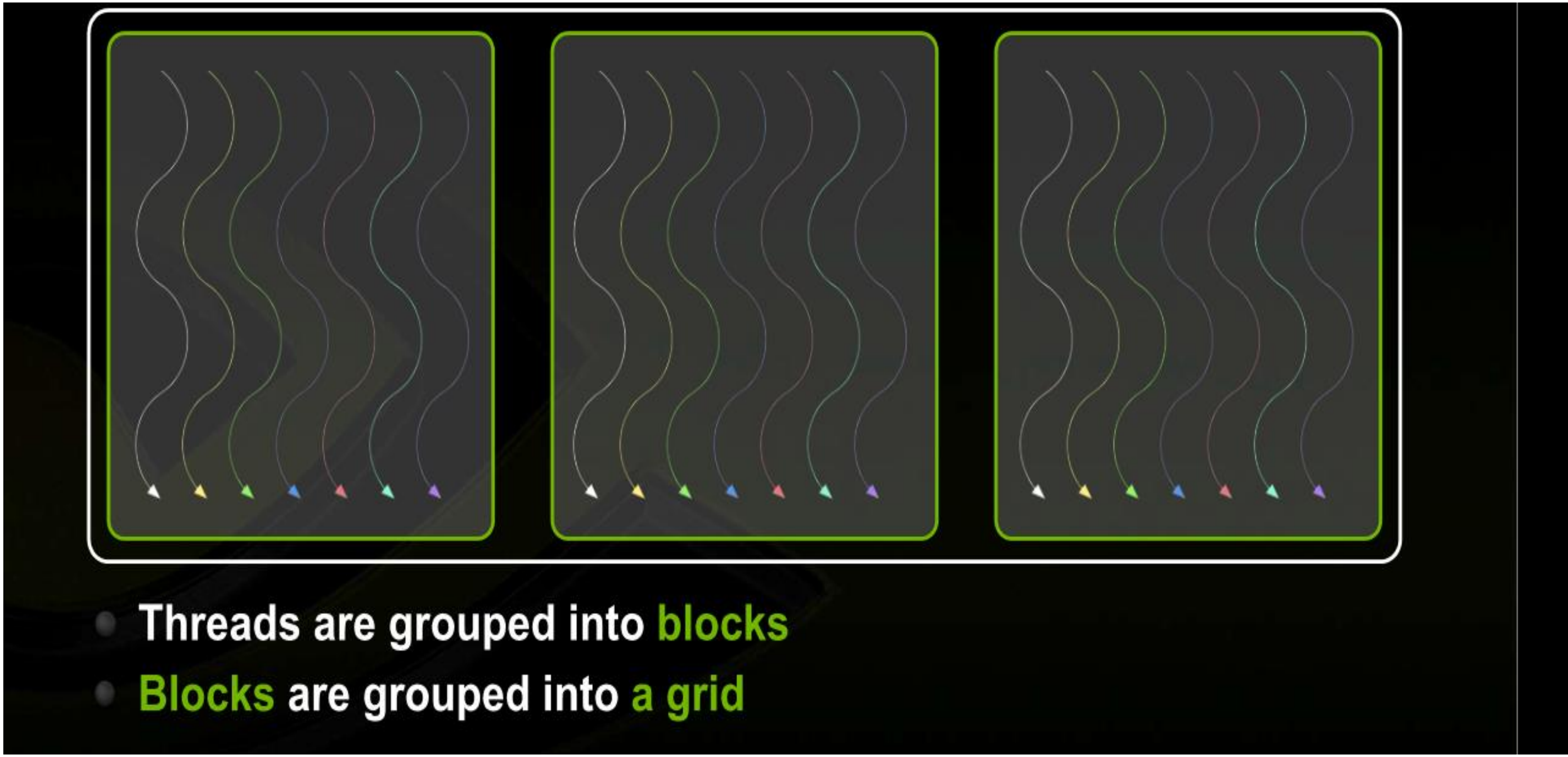
CUDA and Parallel Computing

- Threads are grouped into blocks



CUDA and Parallel Computing

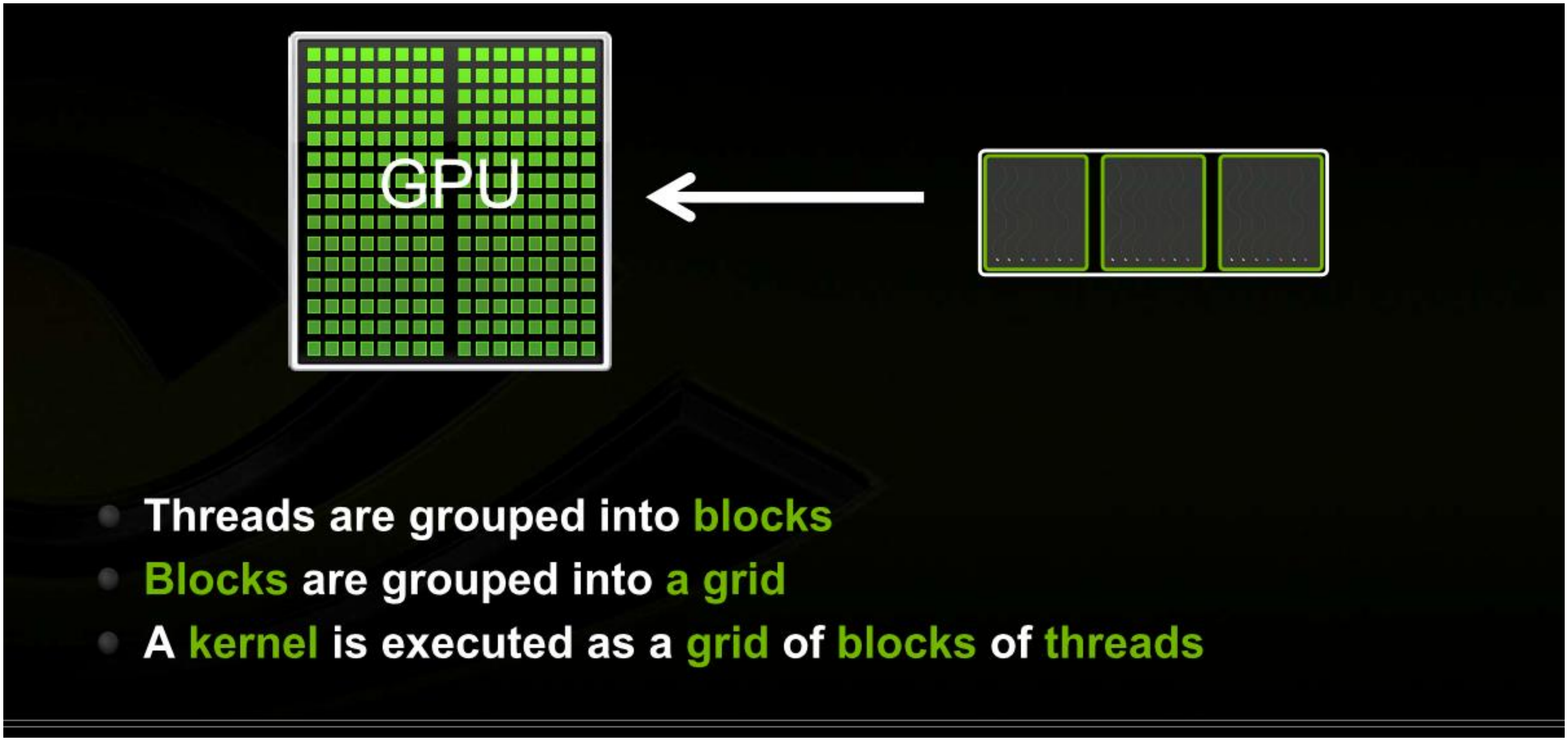
- Blocks are grouped into grids



Queen Mary University of London

CUDA and Parallel Computing

- CUDA Kernels are sub-divided into blocks, blocks into grid and kernel is executed



CUDA and Parallel Programming

- A simple example of adding two variables a and b and assigning the result to a variable c

```
__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

CUDA and Parallel Programming

- Adding to the device using “pointers”

```
__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

CUDA and Parallel Programming

- Managing memory allocations between host and device entities

- 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()`

CUDA and Parallel Programming

- Adding on the device using the defined “add()” function and “main()” function

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

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

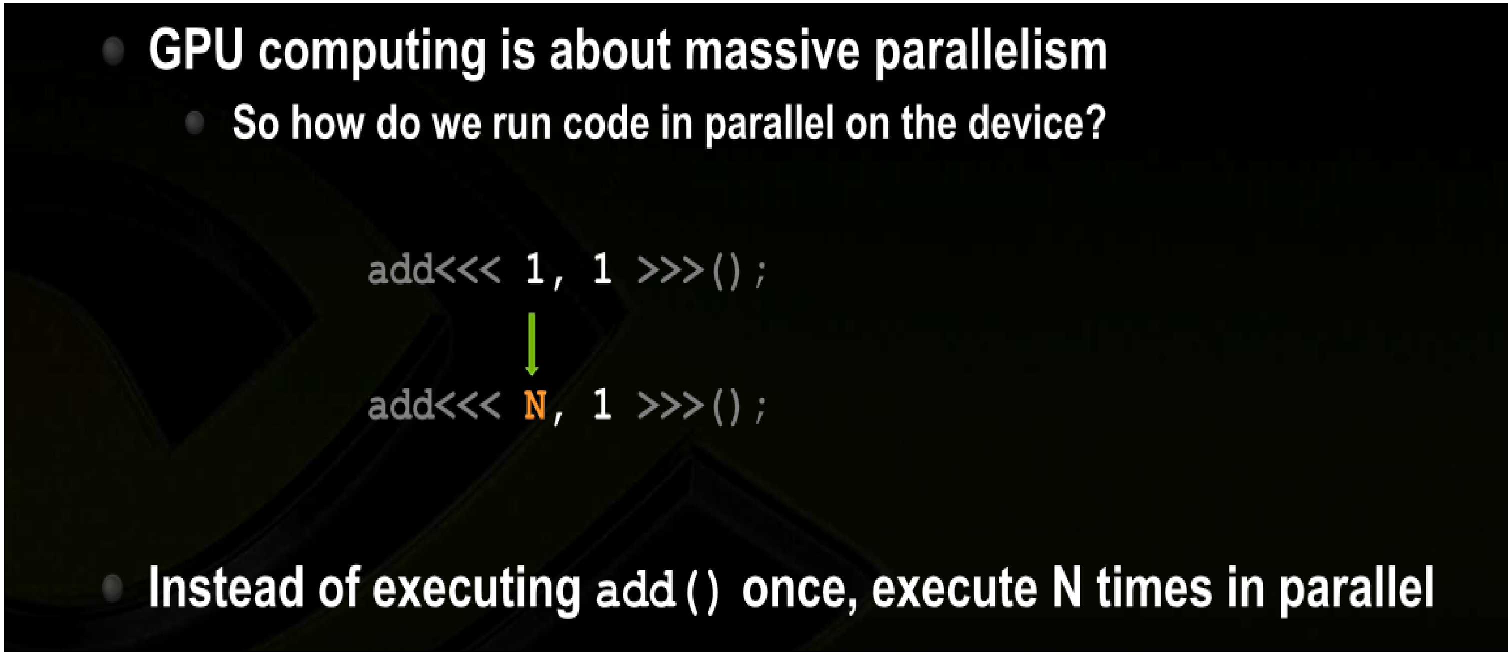
CUDA and Parallel Programming

- Adding on the device using the defined "add()" function and "main()" function
  - Still under the "main()" function

```
// Allocate memory space for device variables
 cudaMalloc((void **) &d_a, size);
 cudaMalloc((void **) &d_b, size);
  cudaMalloc((void **) &d_c, size);
// Assign input values to the host variables
   a = 10;
   b = 6;
 // Copy the input from host to the device
   cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
   cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);
 //Launch "add() kernel to the GPU
   add<<<1,1>>>(d_a, d_b, d_c);
 // Copy the result from the device back to the host
  cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);
 // Free the memory to clean up the space
 cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
  return 0;
} // End main()
```

CUDA and Parallel Programming

- Class Task
  - What is the problem with this program in terms of parallel programming with the solution given for adding a and b and assigning the result to c?
  - Correct the problem
- Solution:



CUDA Parallel Programming using Vectors

- Vector addition on the device

- 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

CUDA Parallel Programming using Vectors

- 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 parallel:

Block 0	Block 1	Block 2	Block 3
`c[0] = a[0] + b[0];`	`c[1] = a[1] + b[1];`	`c[2] = a[2] + b[2];`	`c[3] = a[3] + b[3];`

CUDA Parallel Programming using Vectors

- Vector addition on the device using the “add()” function and “main()”

Queen Mary University of London

CUDA Parallel Programming using Vectors

```
__global__ void add(int *a, int *b, int *c) {
        c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
#define N 512
int main(void) {
    int *a, *b, *c;   // host variables
    int *d_a, *d_b, *d_c; // device variables
    int size = N * sizeof(int);
    // Allocate space for device variables
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);
```

CUDA Parallel Programming using Vectors

- Vector addition on the device using the "add()" function and "main()" continued…

```
// Allocate space for host variables 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 blocks
 add<<<N,1>>>(d_a, d_b, d_c);
```

CUDA Parallel Programming using Vectors

- Vector addition on the device using the “add()” function and “main()” continued…

```
 // Copy result back to host
  cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
  // Cleanup memory
  free(a); free(b); free(c);
  cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
  return 0;
}
```

CUDA By Example: CUDA Code-1

```
__global__ void add(int *a, int *b, int *c) {
        int bid=blockIdx.x;
        if (bid < 6 ) { // The students can also use a "for loop here"
          c[bid]=a[bid]+b[bid];
        }
}

int main(void) {
        int a[6], b[6], c[6];
        int *dev_a, *dev_c, *dev_c;
        // allocate memory to device
        cudaMalloc((void**)&dev_a, 6*sizeof(int));
        cudaMalloc((void**)&dev_b, 6*sizeof(int));
        cudaMalloc((void**)&dev_c, 6*sizeof(int));
        // Fill arrays "a" and "b" with values on the host
        for (int i=0; i<6; i++) {
          a[i]=i;
          b[i]=i*i;
        }
```

Queen Mary
University of London

CUDA By Example: CUDA Code-2

```
// Copy arrays "a" and "b" to the device

        cudaMemcpy(dev_a, a, 6*sizeof(int), cudaMemcpHostToDevice));
        cudaMemcpy(dev_b, b, 6*sizeof(int), cudaMemcpHostToDevice));
        // Launch the kernel
        add<<<2,1>>>(dev_a, dev_b, dev_c);
        //Copy the array "c" from the device to the host
        cudaMemcpy(c,dev_c, 6*sizeof(int), cudaMemcpyDeviceToHost));
        //Print the array "c"
        for (int i=0;i<6;i++) {
          printf("%d\n", c[i]);
        }
        //Free memory allocated to the device
        cudaFree(dev_a);
        cudaFree(dev_b);
        cudaFree(dev_c);

        return 0;
} // End main
```

Some Explanations on the Workings of CUDA

- Using the last CUDA code as an example
- Important differences between GPU and CPU CUDA programs

```
__global__ void add(int *a, int *b, int *c) {
        int bid=blockIdx.x;
        if (bid < 6 ) { // The students can also use a "for loop here"
          c[bid]=a[bid]+b[bid];
        }
}
```

And

```
add<<<2,1>>>(dev_a, dev_b, dev_c);
```

Some Explanations on the Workings of CUDA

```
__global__ void add(int *a, int *b, int *c) {
        int bid=blockIdx.x;
        if (bid < 6 ) { // The students can also use a "for loop here"
            c[bid]=a[bid]+b[bid];
        }
}
.......
.........

add<<<2,1>>>(dev_a, dev_b, dev_c);
.................
................

}
```

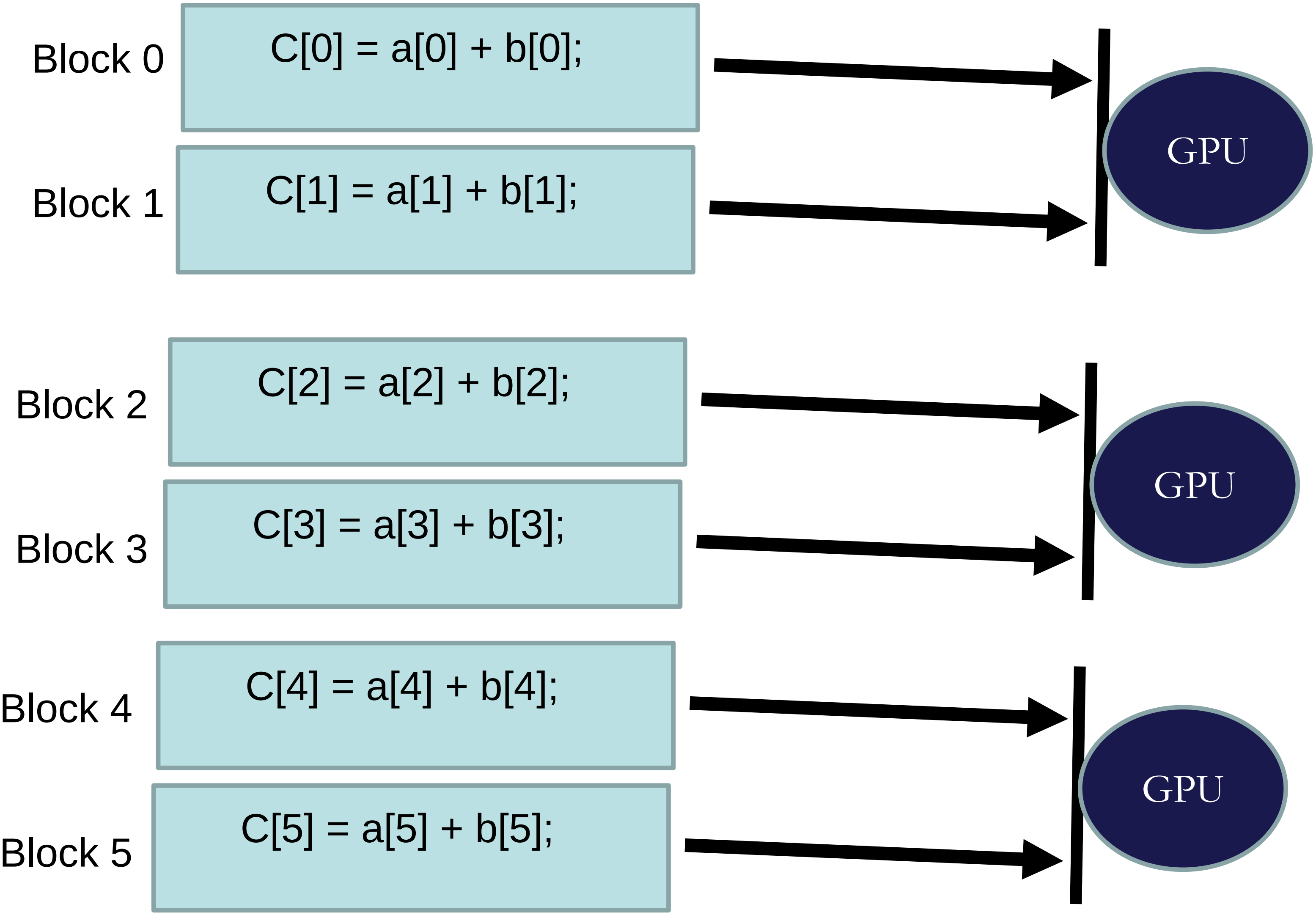
Queen Mary
University of London

Some Explanations on the Workings of CUDA

What happens?

- Because of the 2 blocks in add<<<2,1>>>(dev_a,dev_b,dev_c), 2 copies of __global__ void add(int *a, int *b, int *c) are created
- The two copies take each one thread as indicated in add<<<2,1>>>(dev_a,dev_b,dev_c)
- The two copies run in parallel, taken the first and second runs in the loop, thus calculating c[0] and c[1] at the same time
- This means that c[2] and c[3] will be calculated next followed by c[4] and c[5] finally.

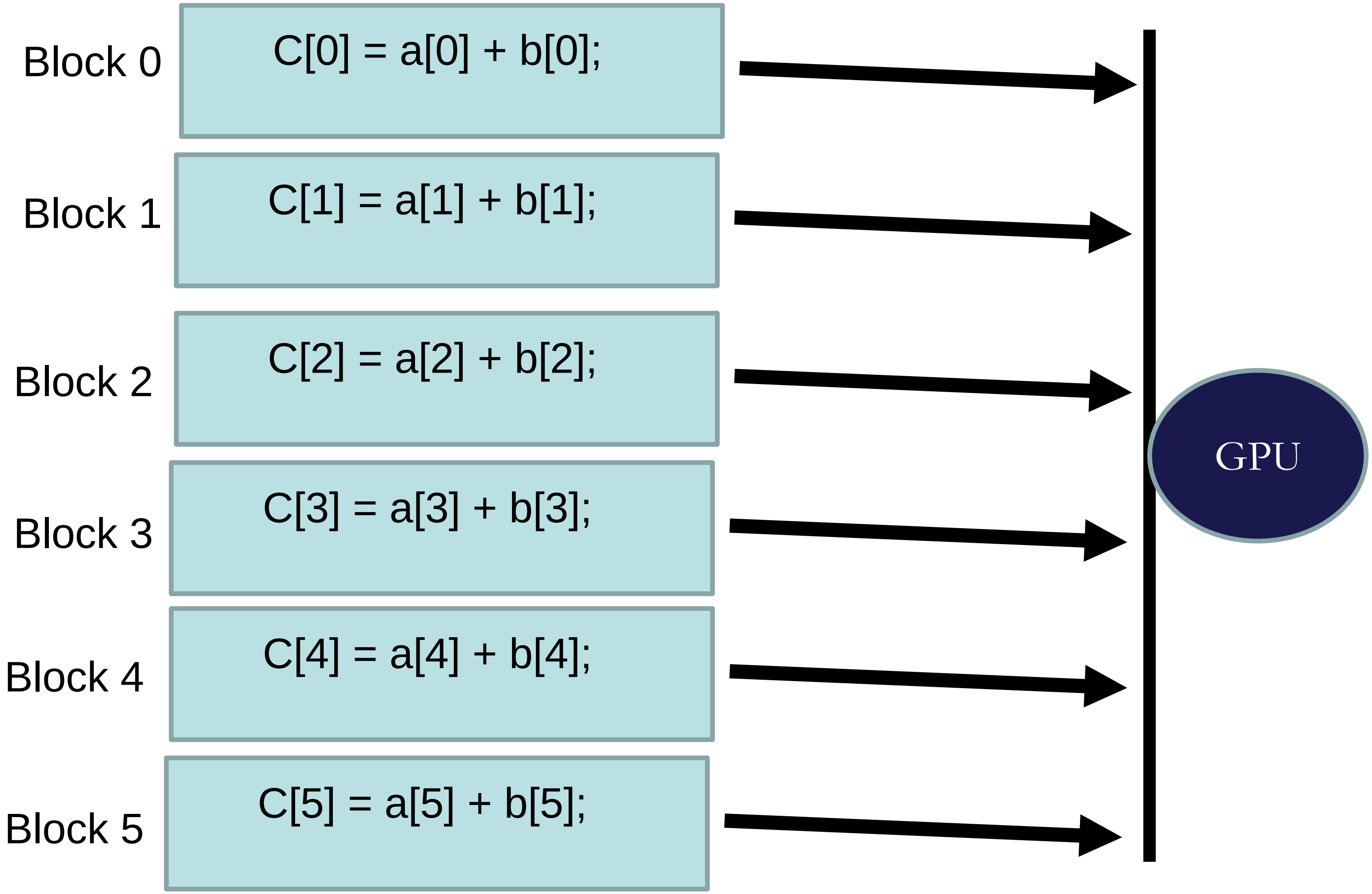
Some Explanations on the Workings of CUDA



There are a total of 3 iterations, instead of 6 if done on a single CPU machine

Some Explanations on the Workings of CUDA

If we now have add<<<6,1>>>(dev_a,dev_b,dev_c), then we will have only one iteration



Queen Mary University of London

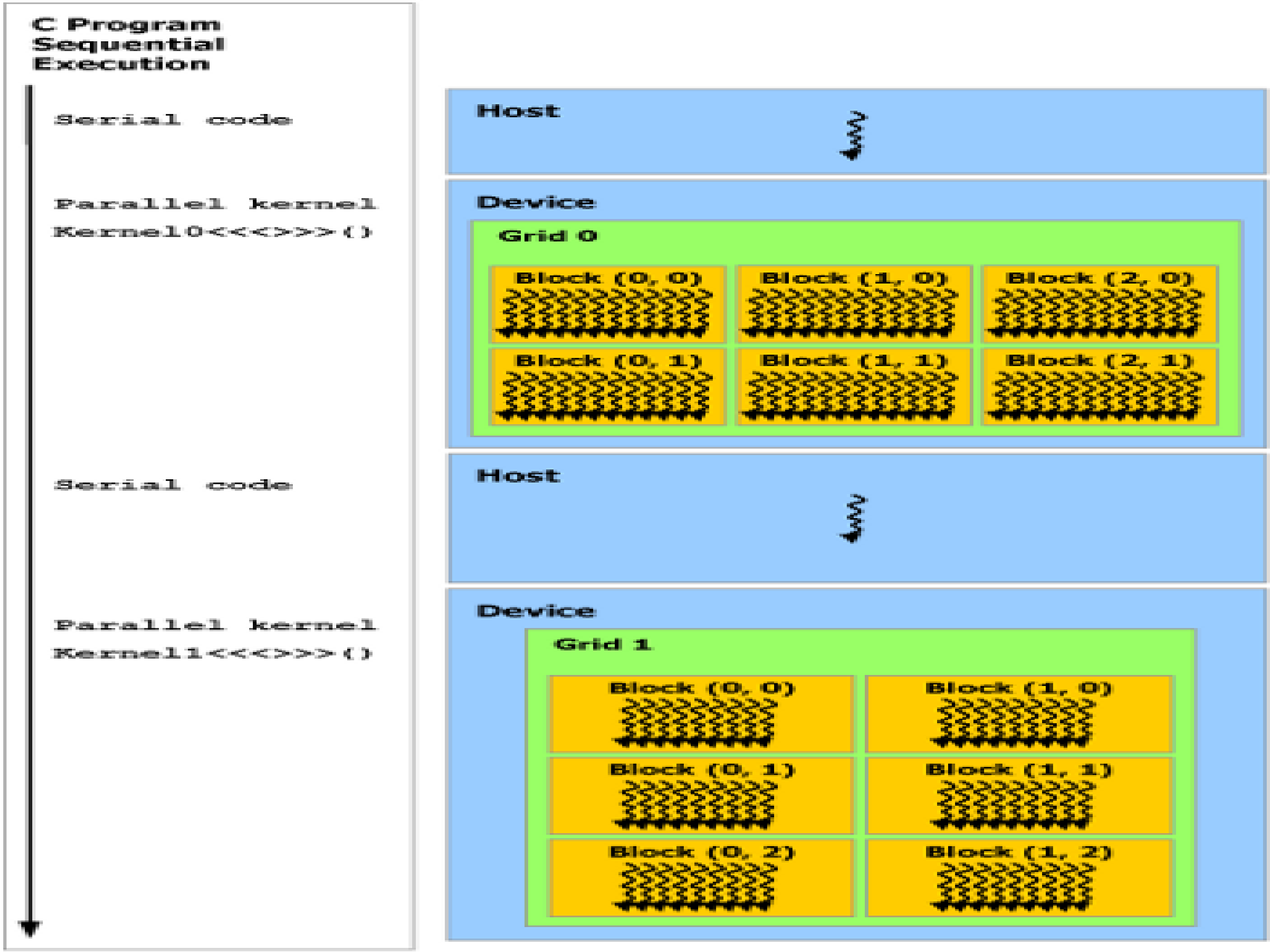
Class Discussions

- Discuss the different scenarios in using blocks and threads
- Examples
  - `add<<<10,1>>>()`, `add<<<1,10>>>()`
- Which is better,uUsing more blocks or more threads?

CUDA Heterogeneous Programming

- The CUDA programming model assumes that the CUDA threads execute on a physically separate *device* that operates as a coprocessor to the host running the C program
  - The kernels execute on a GPU and the rest of the C program executes on a CPU
- The CUDA programming model also assumes that both the *host* and the device maintain their own separate memory spaces in DRAM
  - Referred to as host memory and device memory
- CUDA provides a programming interface for C, C++ and other languages to simplify development of codes on the CUDA model for CUDA-enabled GPUs

CUDA Heterogeneous Programming



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 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

```
__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 parallel:

Block 0	Block 1	Block 2	Block 3
`c[0] = a[0] + b[0];`	`c[1] = a[1] + b[1];`	`c[2] = a[2] + b[2];`	`c[3] = a[3] + b[3];`

Vector Addition on the Device: `add()`

- Returning to our parallelized **`add()`** kernel

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

- Let's take a look at main()…

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 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 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 (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<<<N,1>>>(…);`**
  - Use **`blockIdx.x`** to access block index

Class Task

- Write simple CUDA C programme to multiply two numbers a, b and save the results in c
  - Use the “threadIdx” built-in variable
- What do you understand by heterogeneous programming in GPUs?