CUDA C++ BASICS

NVIDIA Corporation

WHAT IS CUDA?

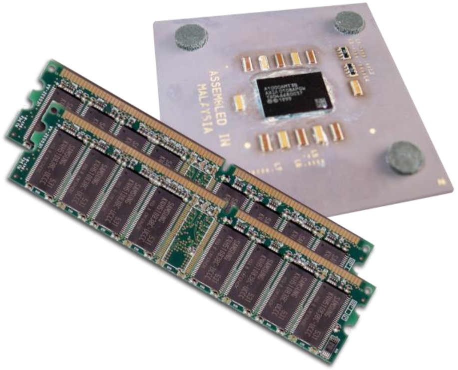
- CUDA Architecture
  - Expose GPU parallelism for general-purpose computing
  - Expose/Enable performance
- CUDA C++
  - Based on industry-standard C++
  - Set of extensions to enable heterogeneous programming
  - Straightforward APIs to manage devices, memory etc.
- This session introduces CUDA C++
  - Other languages/bindings available: Fortran, Python, Matlab, etc.

INTRODUCTION TO CUDA C++

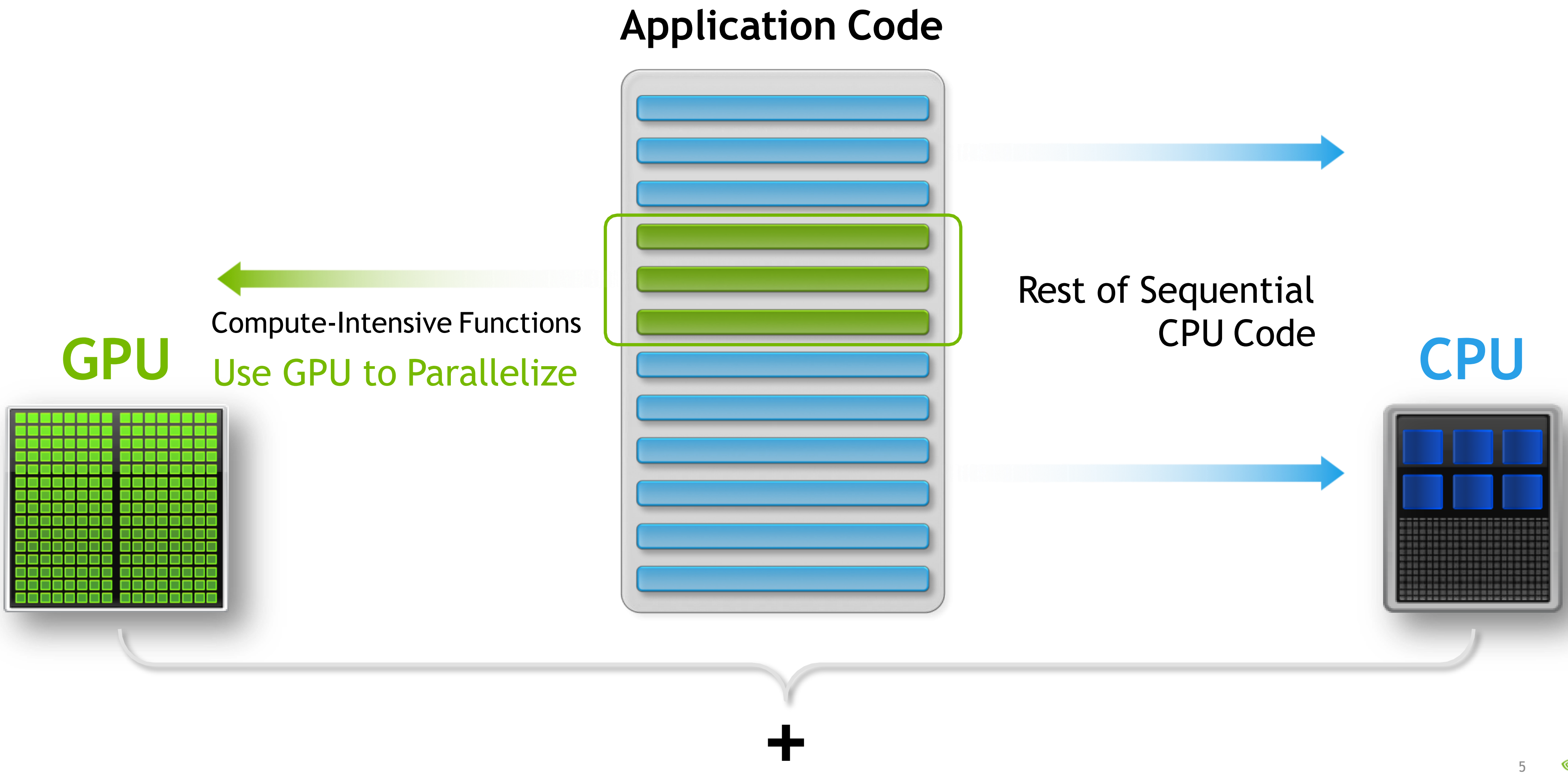
- What will you learn in this session?
  - Start with vector addition
  - Write and launch CUDA C++ kernels
  - Manage GPU memory
  - (Manage communication and synchronization)-> next session
- (Some knowledge of C or C++ programming is assumed.)

HETEROGENEOUS COMPUTING

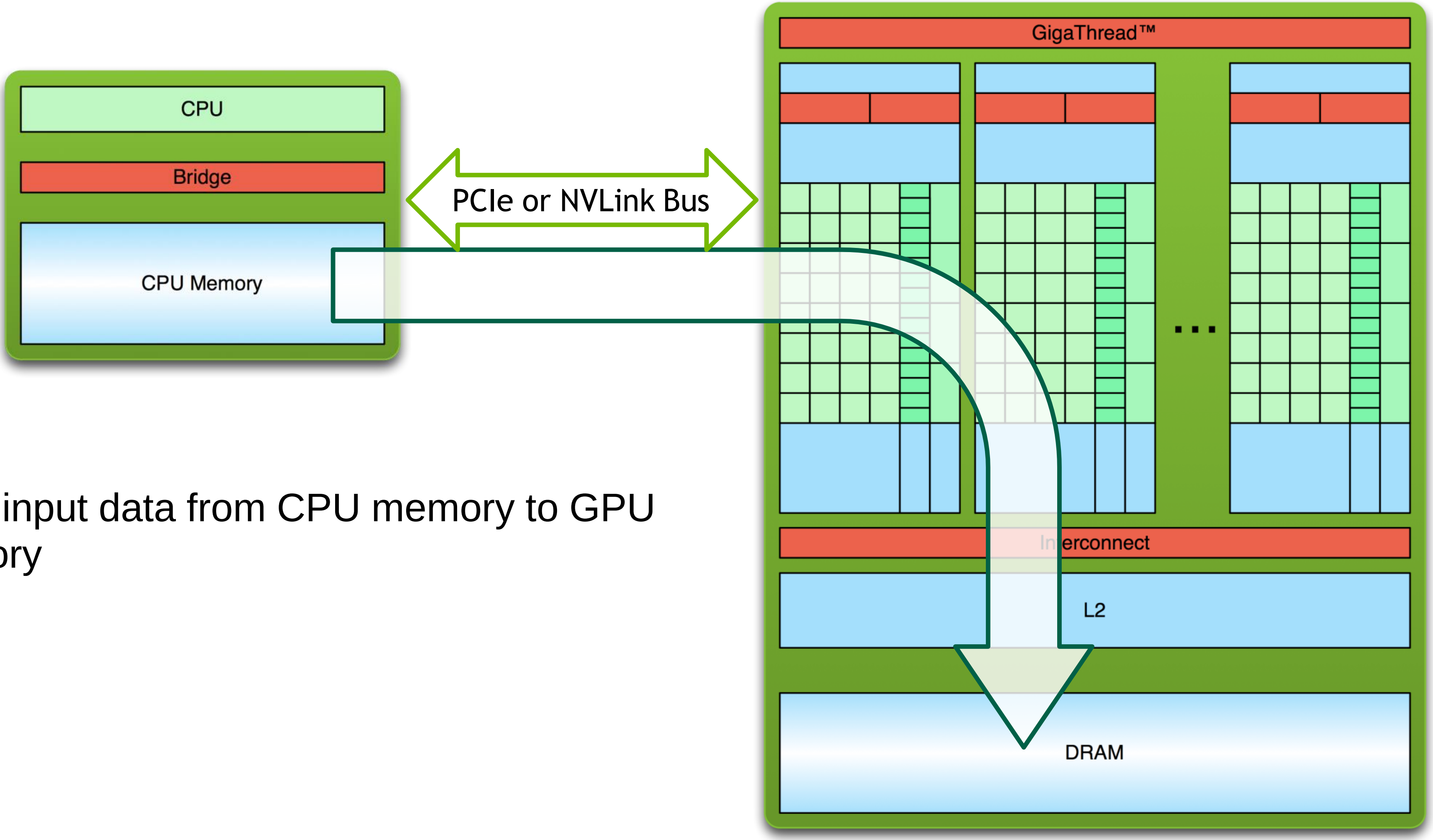
- *Host* The CPU and its memory (host memory)
- *Device* The GPU and its memory (device memory)



PORTING TO CUDA

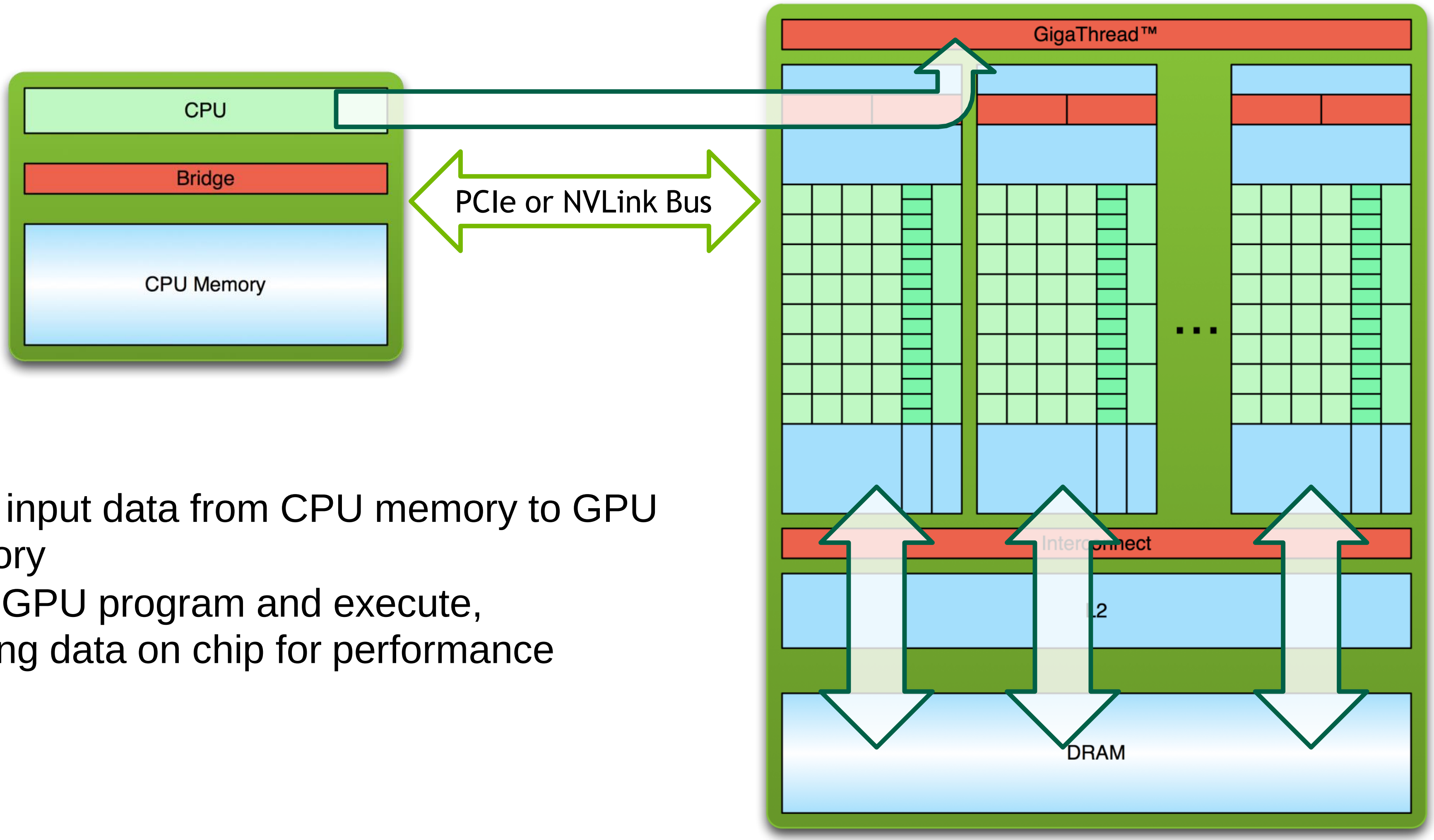


SIMPLE PROCESSING FLOW



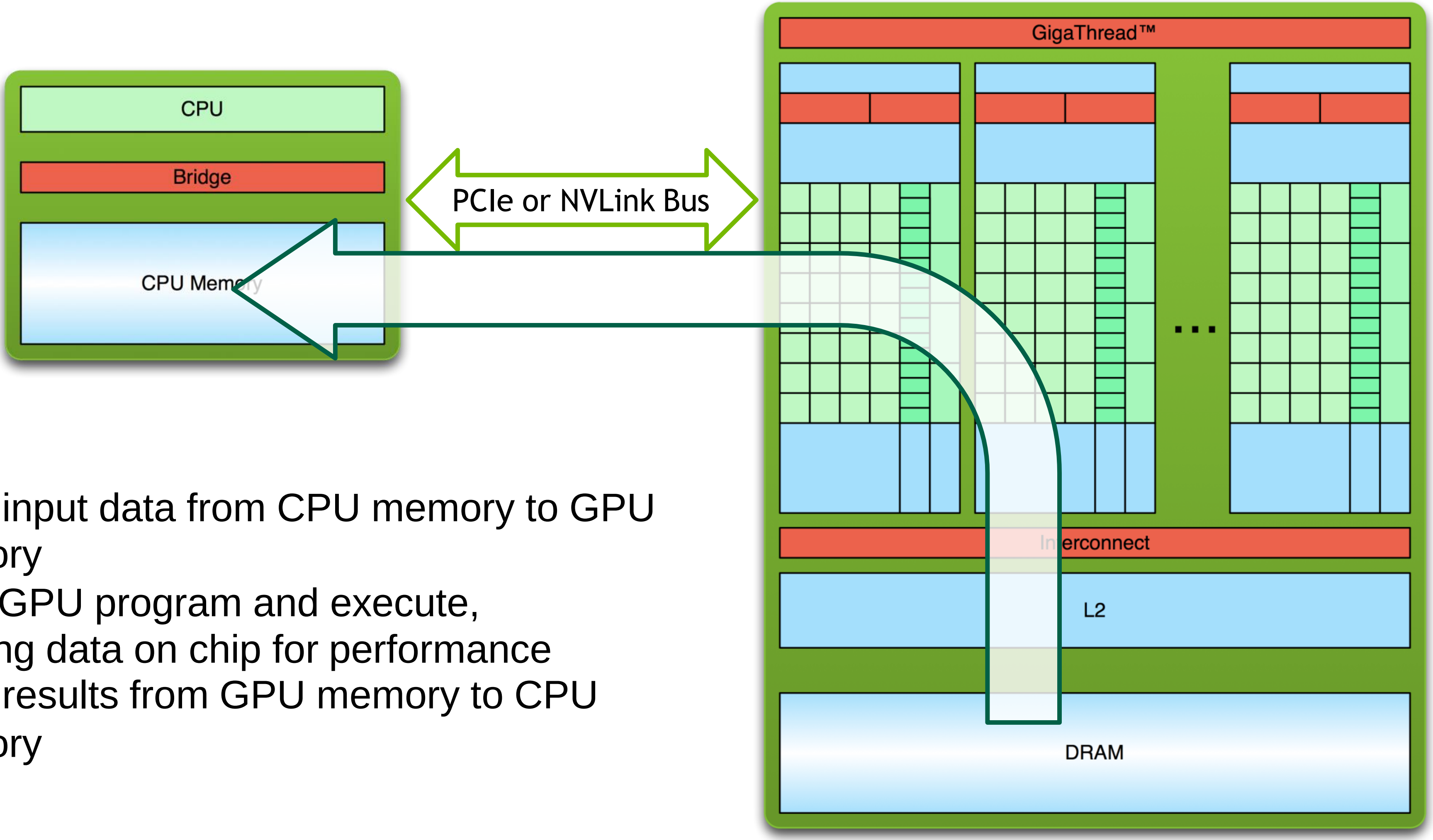
1. Copy input data from CPU memory to GPU memory

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

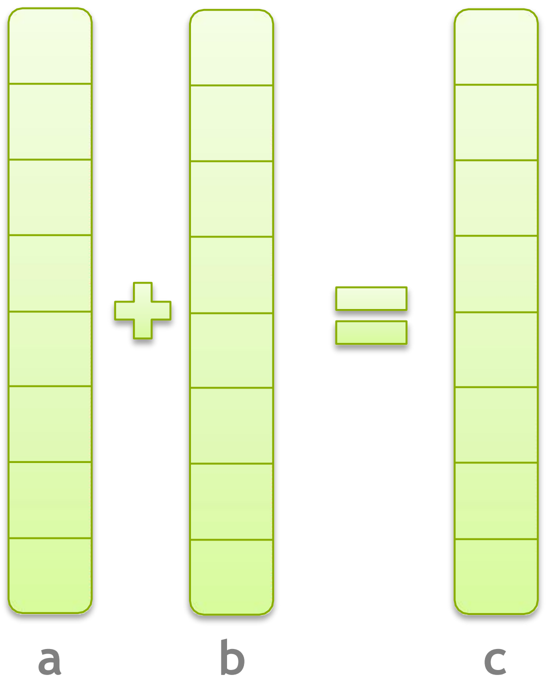
SIMPLE PROCESSING FLOW



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

PARALLEL PROGRAMMING IN CUDA C++

- GPU computing is about massive parallelism!
- We need an interesting example…
- We'll start with vector addition



GPU KERNELS: DEVICE CODE

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

- CUDA C++ keyword `__global__` indicates a function that:
  - Runs on the device
  - Is called from host code (can also be called from other device code)
- `nvcc` separates source code into host and device components
  - Device functions (e.g. `mykernel()`) processed by NVIDIA compiler
  - Host functions (e.g. `main()`) processed by standard host compiler:
    - `gcc`, `cl.exe`

GPU KERNELS: DEVICE CODE

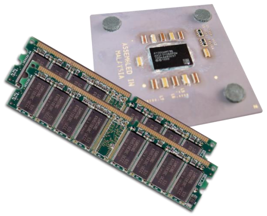
```
mykernel<<<1,1>>>();
```

- Triple angle brackets mark a call to *device* code
    - Also called a “kernel launch”
    - We’ll return to the parameters (1,1) in a moment
    - The parameters inside the triple angle brackets are the CUDA kernel **execution configuration**
- That’s all that is required to execute a function on the GPU!

MEMORY MANAGEMENT

- Host and device memory are separate entities
- Device pointers point to GPU memory
    - Typically passed to device code
    - Typically not dereferenced in host code
- Host pointers point to CPU memory
    - Typically not passed to device code
    - Typically not dereferenced in device code

- Simple CUDA API for handling device memory
    - cudaMalloc(), cudaFree(), cudaMemcpy()
    - Similar to the C equivalents malloc(), free(), memcpy()



RUNNING CODE IN 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 all 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
- Built-in variables like `blockIdx.x` are zero-indexed (C/C++ style), 0..`N`-1, where `N` is from the kernel execution configuration indicated at the kernel launch

VECTOR ADDITION ON THE DEVICE

```
#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

```
    // 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 (or possibly from other device code)
- 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

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()`:

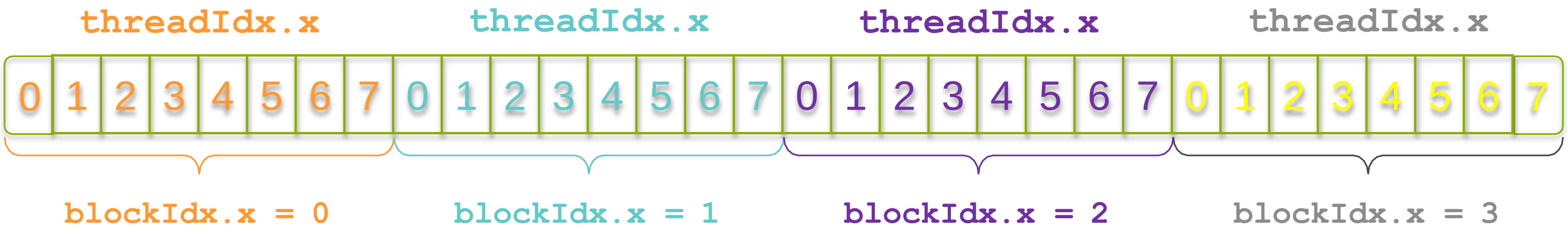
```
add<<< 1, N >>>();
```

COMBINING BLOCKS *AND* THREADS

- We’ve 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…

INDEXING ARRAYS WITH BLOCKS AND THREADS

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

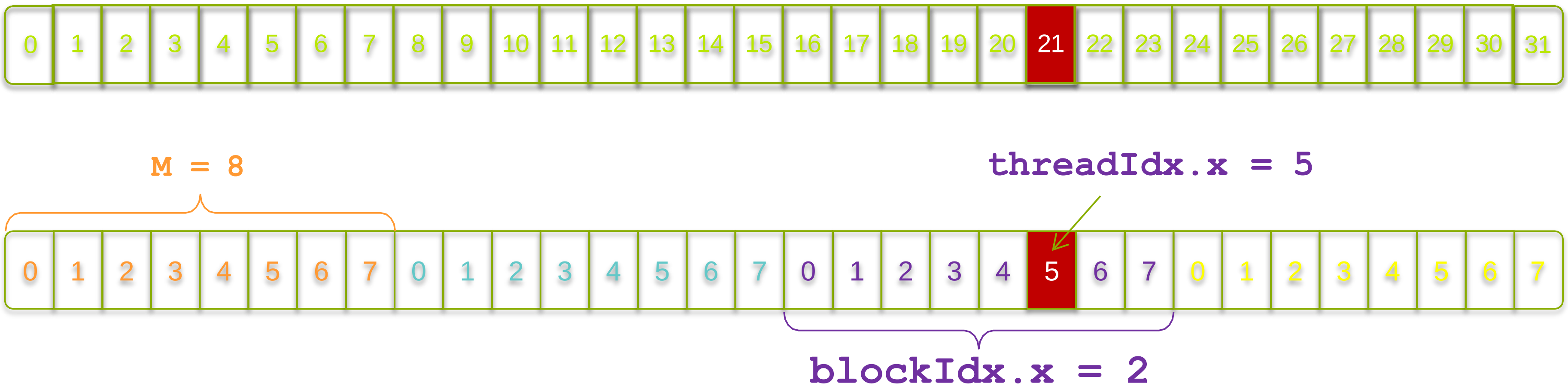


- With M threads/block a unique index for each thread is given by:

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

INDEXING ARRAYS: EXAMPLE

- Which thread will operate on the red element?




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

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

```
#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 and setup input values
    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

```
    // 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 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 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… (next session)

REVIEW

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

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

FUTURE SESSIONS

- CUDA Shared Memory
- CUDA GPU architecture and basic optimizations
- Atomics, Reductions, Warp Shuffle
- Using Managed Memory
- Concurrency (streams, copy/compute overlap, multi-GPU)
- Analysis Driven Optimization
- Cooperative Groups

FURTHER STUDY

- An introduction to CUDA:
  - https://devblogs.nvidia.com/easy-introduction-cuda-c-and-c/
- Another introduction to CUDA:
  - https://devblogs.nvidia.com/even-easier-introduction-cuda/
- CUDA Programming Guide:
  - https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
- CUDA Documentation:
  - https://docs.nvidia.com/cuda/index.html
  - https://docs.nvidia.com/cuda/cuda-runtime-api/index.html (runtime API)