Multicore Computing
Lecture21 – GPU

남 범 석

bnam@skku.edu

SIMD vs. SPMD

- SIMD
  - Single Instruction Multiple Data
  - Designed for tightly-coupled, synchronous hardware
  - i.e., vector units
- SPMD
  - Single Program Multiple Data
  - Designed for clusters
  - Too expensive to do synchronization on each statement, need a looser model

SIMD vs. SPMD

- SIMD example

```
if (factor == 0)
  factor = 1.0
parfor (i = 1..N)
  A[i] = B[i]*factor;
j += factor;
```

- Single thread of control
  - Global synchronization at each program instruction
  - Need hardware support

- SPMD example

```
if (factor == 0)
  factor = 1.0
A[myid] = B[myid]*factor;
j += factor;
```

- Multiple threads of control
  - Asynchronous
  - Programmer-specified synchronization

SIMD vs. SPMD

- SIMD example

```
if (factor == 0)
   factor = 1.0
parfor (i = 1..N)
   A[i] = B[i]*factor;
j += factor;
```

- Only the master thread executes the sequential part.

- SPMD example

```
if (factor == 0)
   factor = 1.0
A[myid] = B[myid]*factor;
j += factor;
```

- All threads execute the sequential part.
- Why?
  - Often cheaper to replicate computation in parallel than compute in one place.

SPMD and MPI

- SPMD is an adaption of SIMD for coarse-grain, distributed parallel machines.

- MPI is the most popular way to write SPMD programs
  - Every thread has a unique id
  - Threads can send/receive messages
  - Synchronization primitives
  - High communication overheads

A GPU is a SIMD (SIMT) Machine

- Except it is **not** programmed using SIMD instructions

- Instead, It is programmed using threads (SPMD model)
  - Each thread executes the same code but operates a different piece of data
  - Each thread has its own context (i.e., can be treated/restarted/executed independently)

- A set of threads executing the same instruction are dynamically grouped into a **warp** by the hardware
  - A warp is essentially a SIMD operation formed by hardware!

GPU Programming

- GPU – graphics processing unit

- Originally designed as a graphics processor
- Nvidia's GeForce 256 (1999) – first GPU
  - Up until 1999, the term “GPU” didn’t exist
- Now, GPUs are present in
  - Embedded systems
  - Game consoles
  - Mobile phones
  - etc

GPGPU

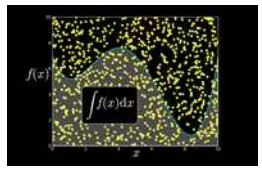
- 1999-2000 computer scientists from various fields started using GPUs to accelerate a range of scientific applications.

- GPU-accelerated libraries
  - powerful library of parallel algorithms and data structures
  - cuFFT, cuBLAS, Thrust, NPP, IMSL, CULA, cuRAND, etc
  - thrust::sort algorithm delivers **100**x faster sorting performance than STL and TBB

cuSPARSE

NPP

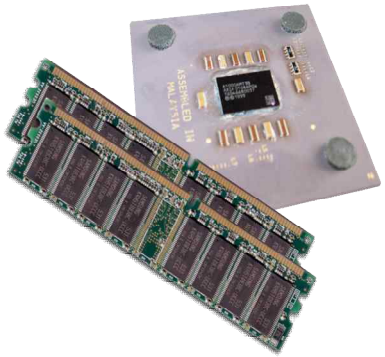
cuFFT



cuRAND

Heterogeneous Computing

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



Host

Device

Heterogeneous Computing

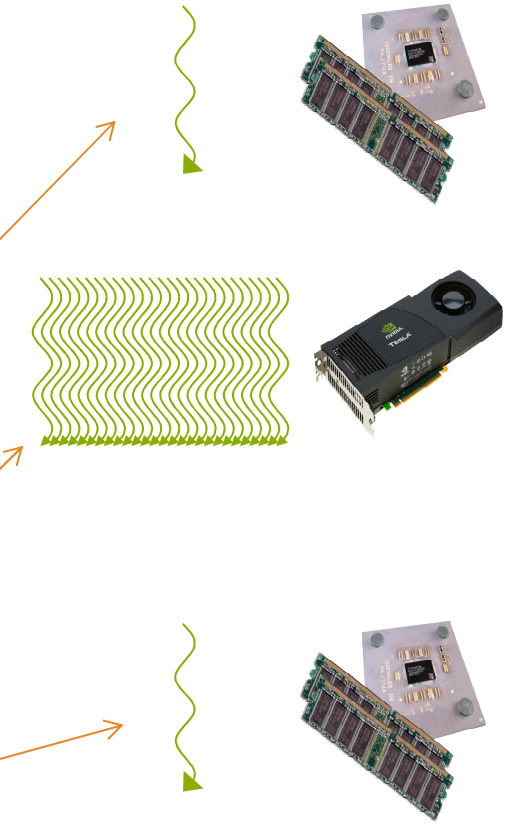
```
#include <iostream>
#include <algorithm>

using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__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];
        }

        // Synchronize (ensure all the data is available)
        __syncthreads();

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

        // Store the result
        out[gindex] = result;
}

void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}

int main(void) {
        int *in, *out;          // host copies of a, b, c
        int *d_in, *d_out;      // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS,
d_out + RADIUS);

        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```

parallel fn

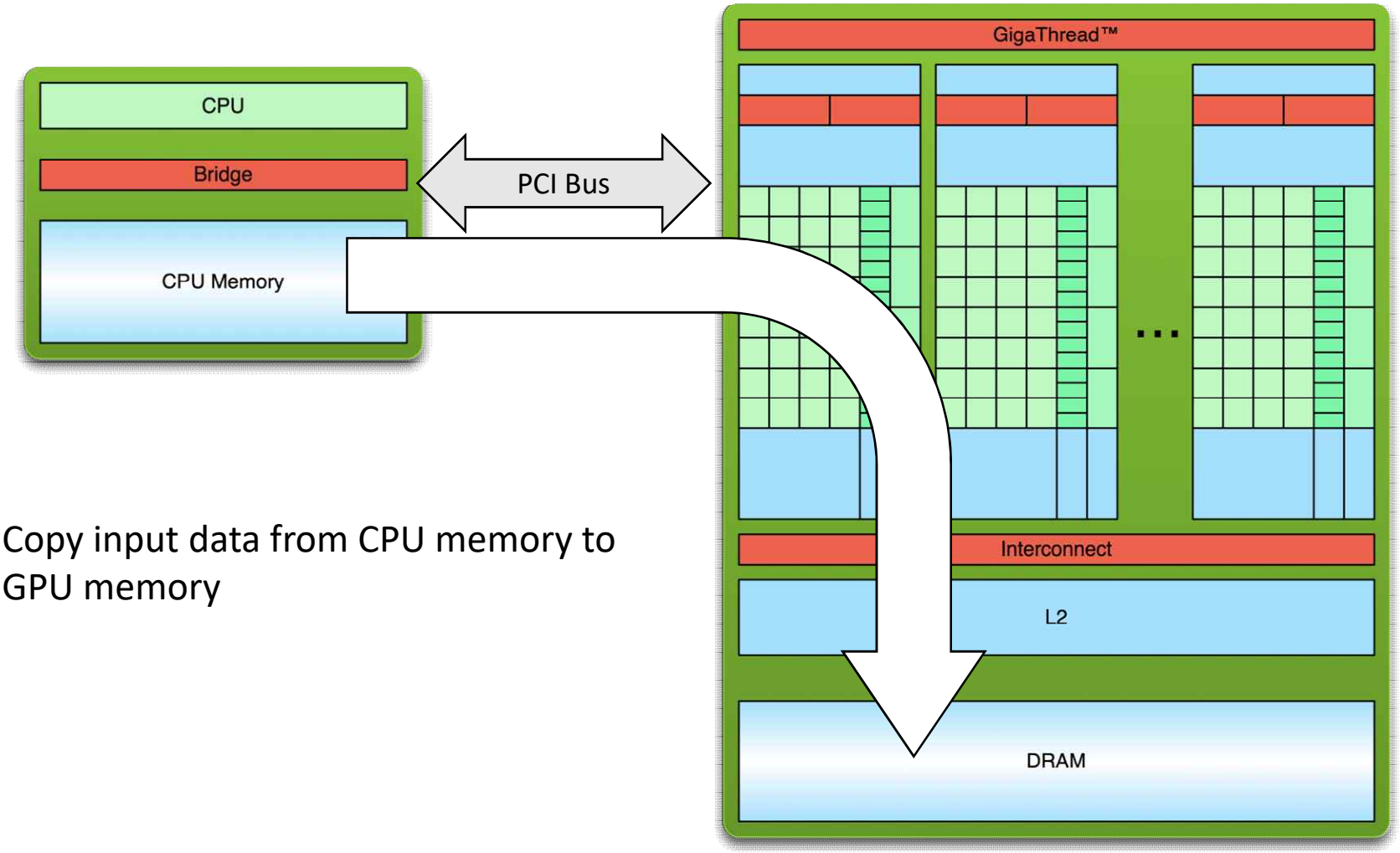
serial code

parallel code

serial code

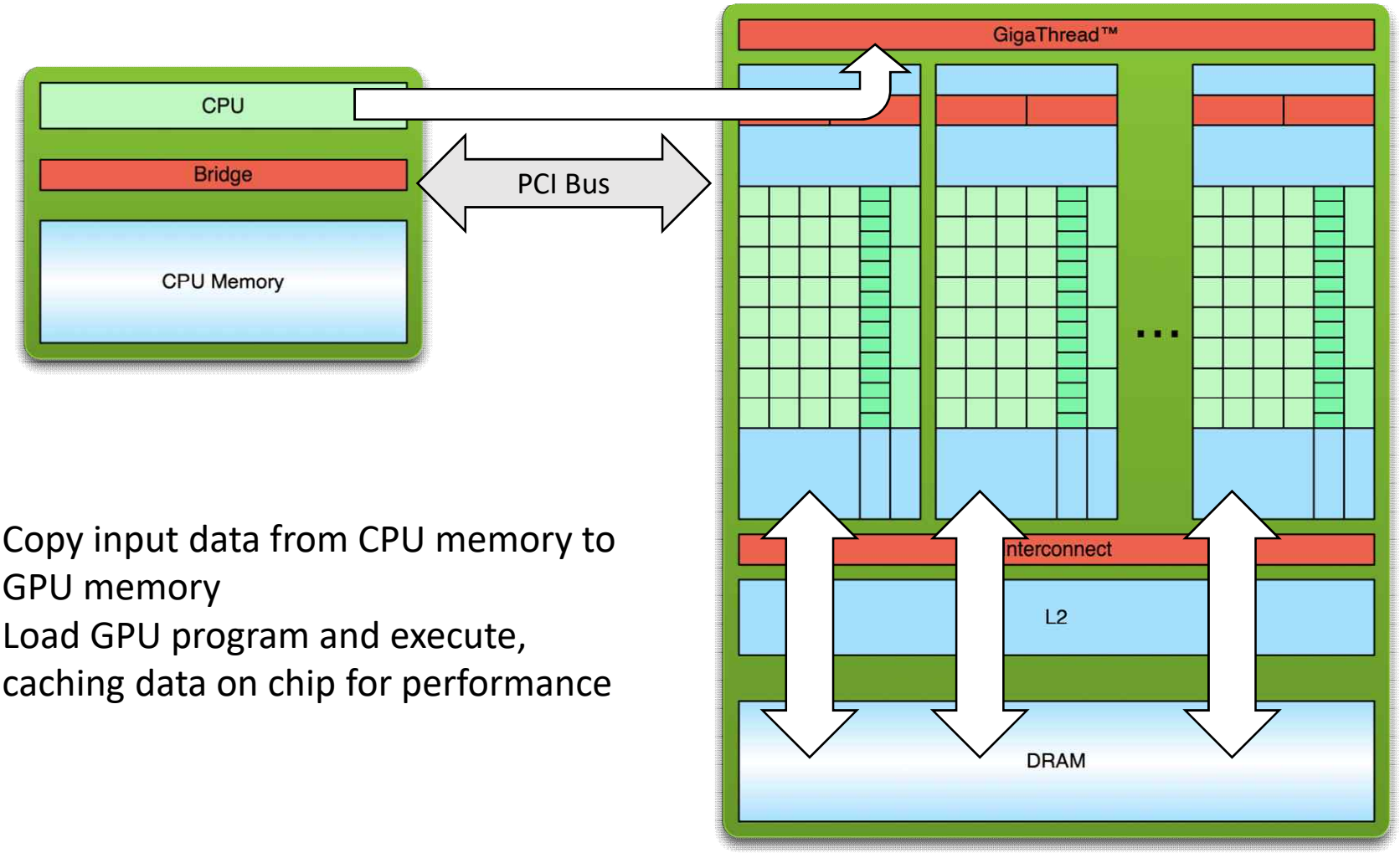


Simple Processing Flow



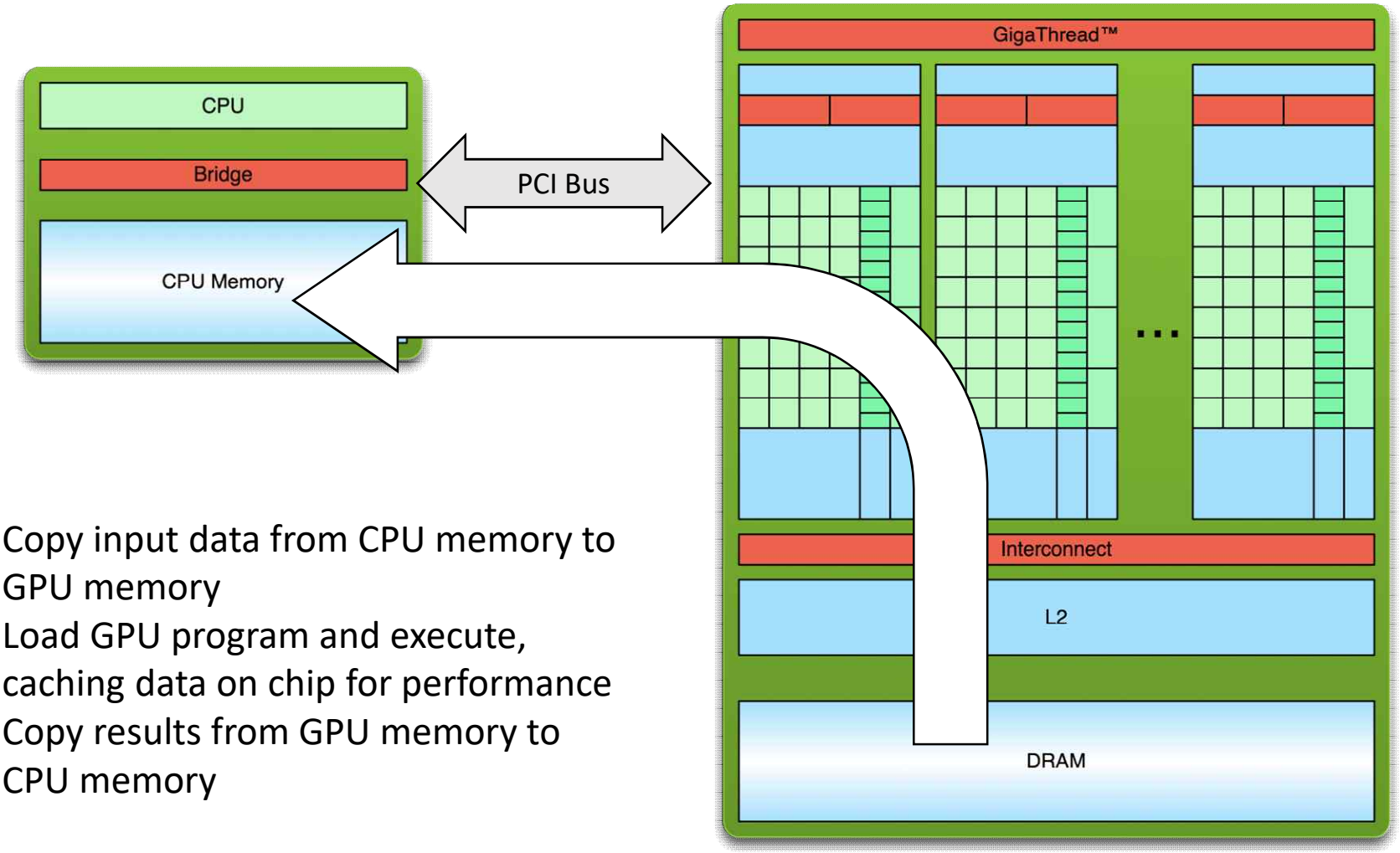
1. Copy input data from CPU memory to GPU memory

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

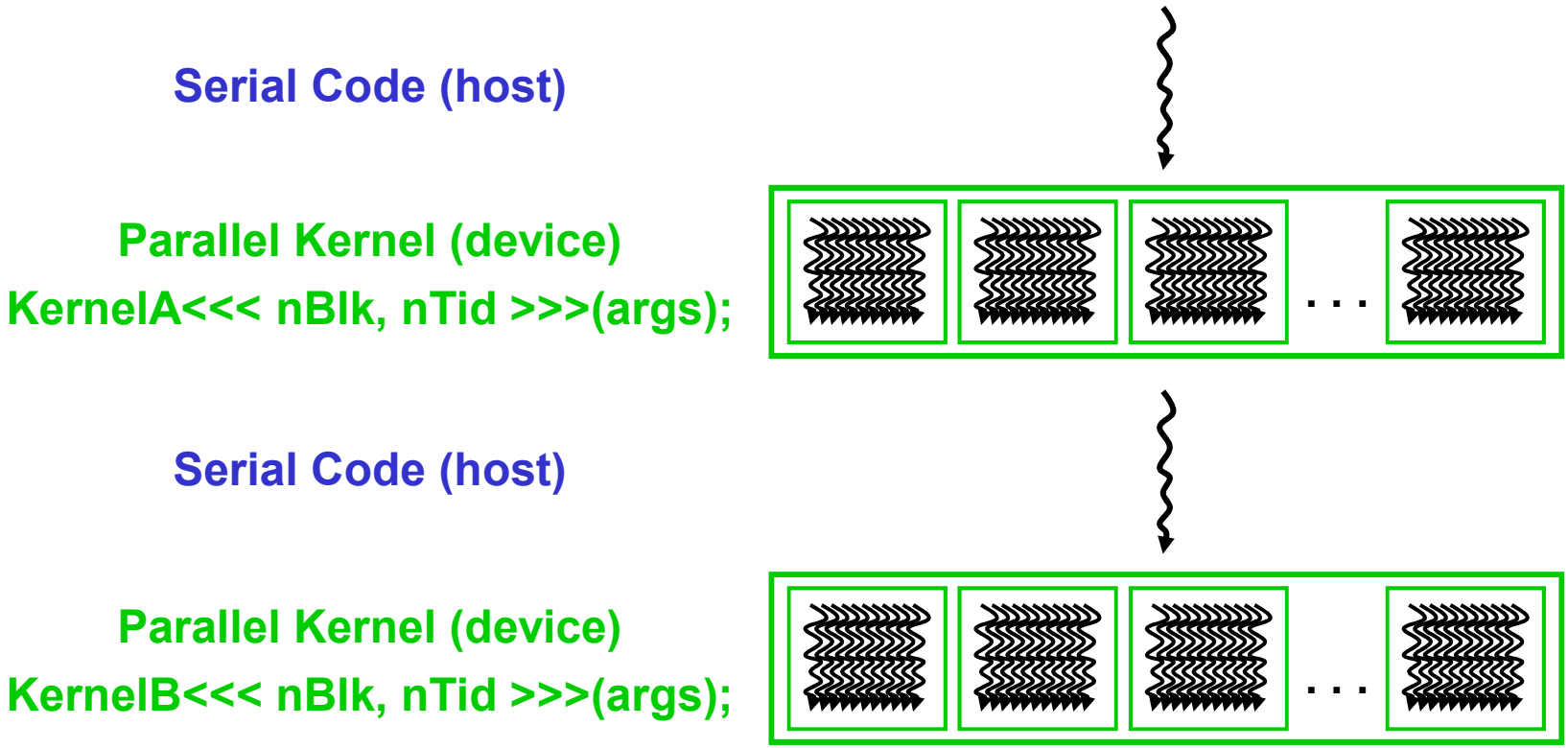
Simple Processing Flow



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

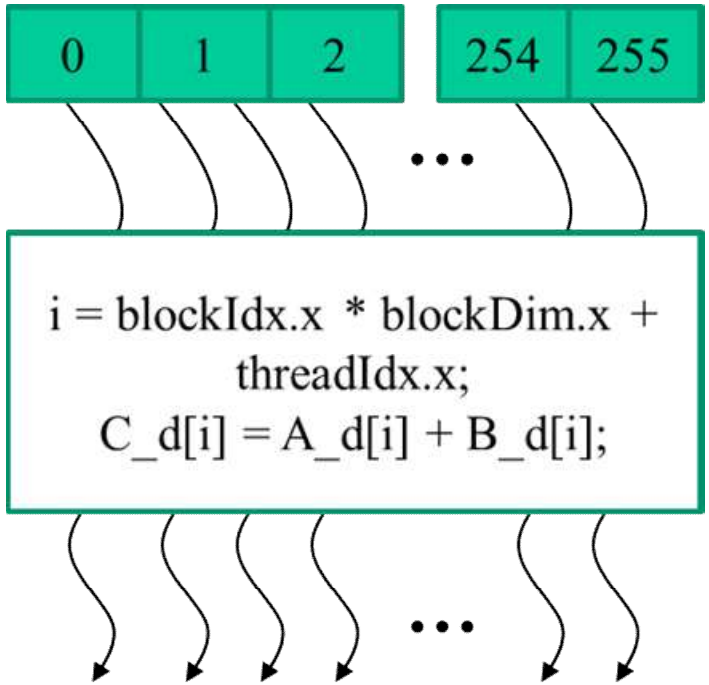
CUDA – Execution Model

- Integrated host+device app C program
  - Serial or modestly parallel parts in host C code
  - Highly parallel parts in device SPMD kernel C code



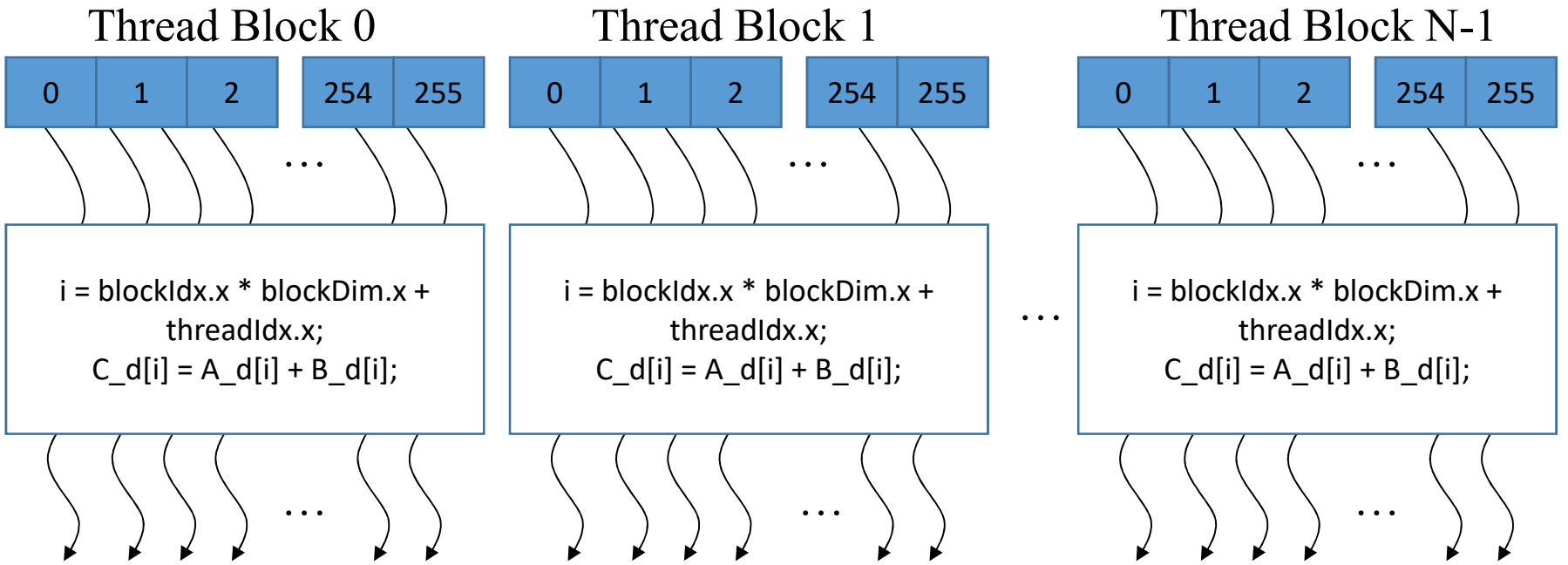
Arrays of Parallel Threads

- A CUDA kernel is executed by a grid (array) of threads
  - All threads in a grid run the same kernel code (SPMD)
  - Each thread has an index that it uses to compute memory addresses and make control decisions



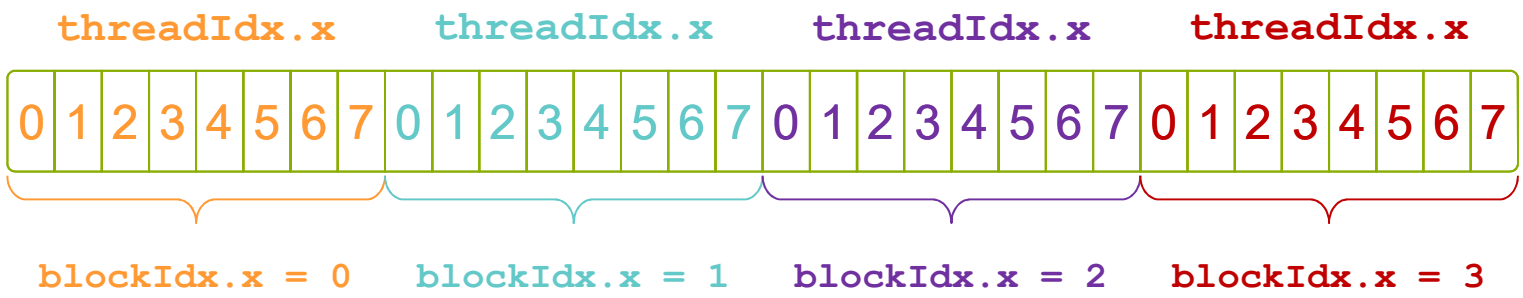
Thread Blocks: Scalable Cooperation

- Divide thread array into multiple blocks
  - Threads within a block cooperate via shared memory, atomic operations and barrier synchronization
  - Threads in different blocks cannot cooperate



Indexing Arrays with Blocks and Threads

- using `blockIdx.x` and `threadIdx.x`
  - Consider indexing an array with one element per thread (8 threads/block)

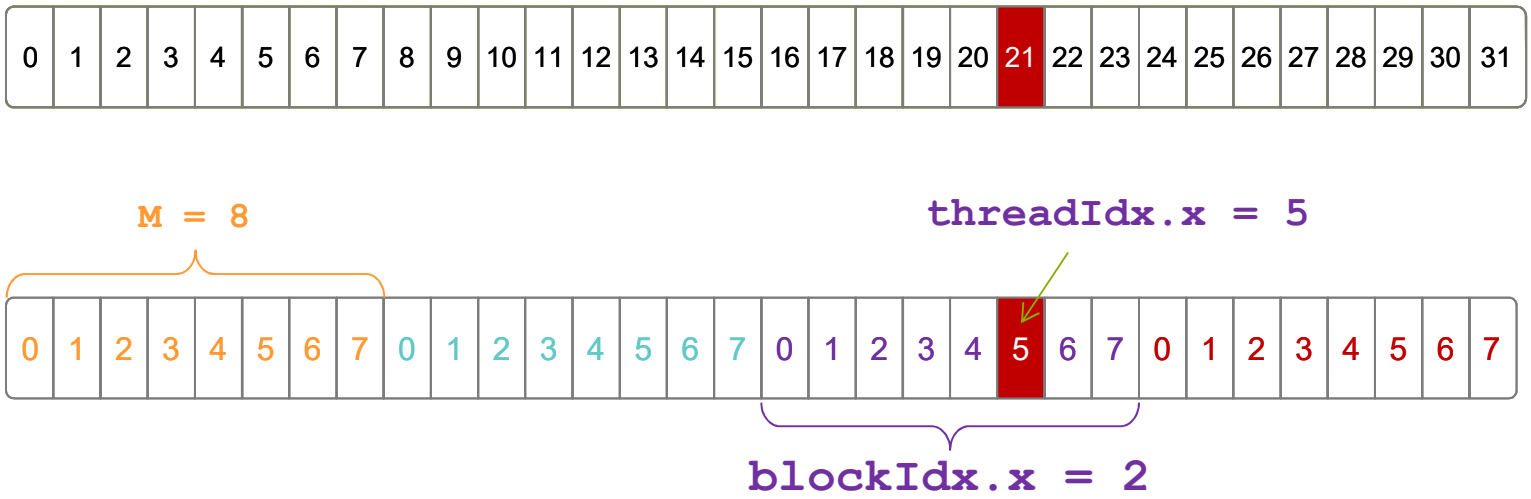


- With M threads/block a unique index for each thread is given by:

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


Indexing Arrays: Example

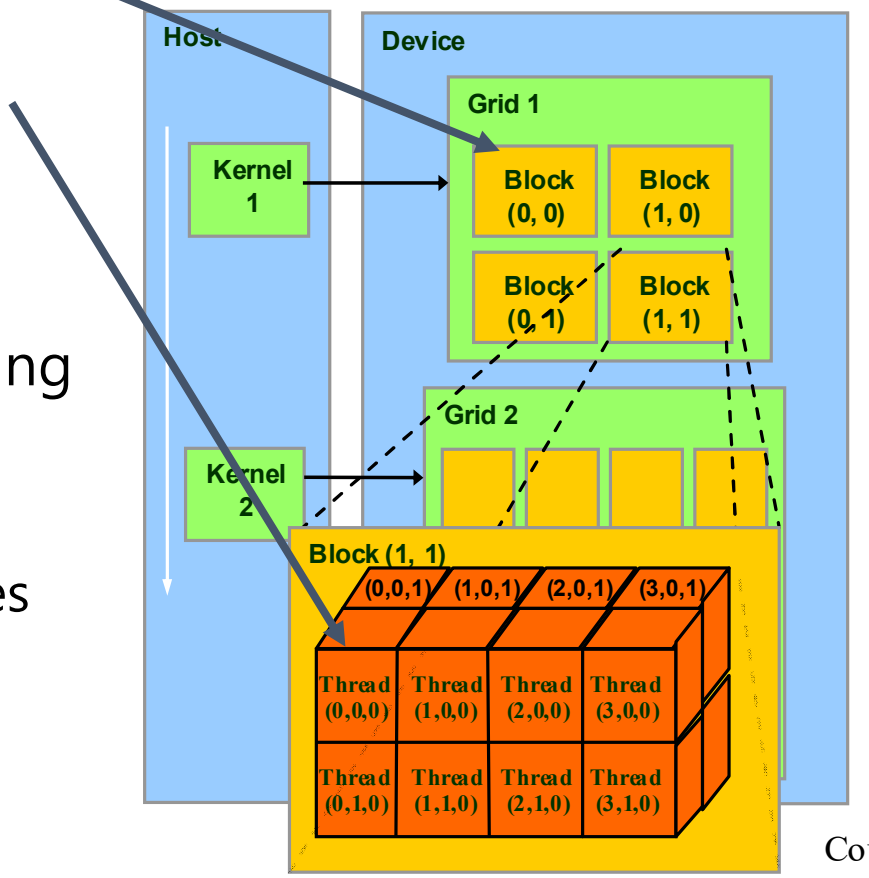
- Which thread will operate on the red element?




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

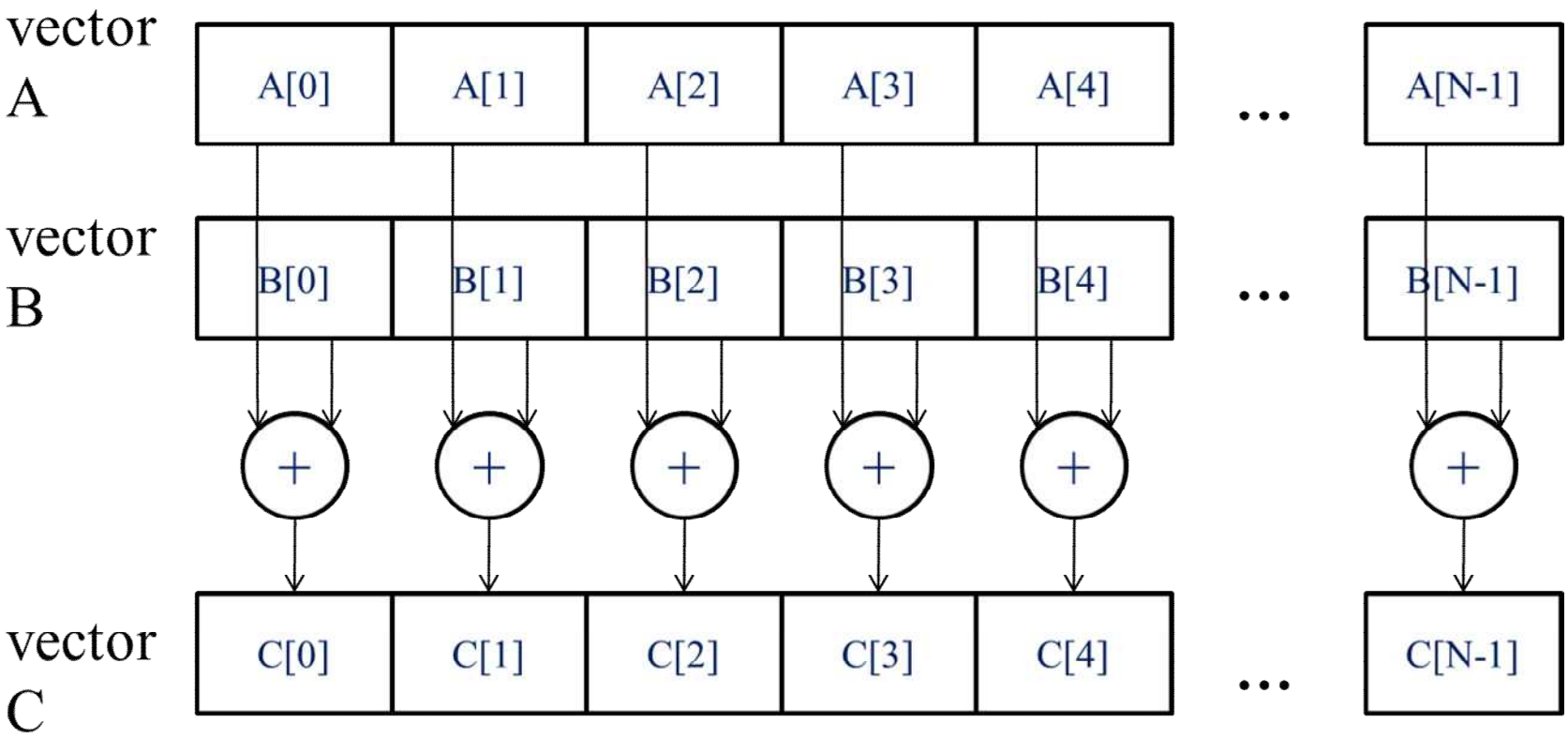

blockIdx and threadIdx

- Each thread uses indices to decide what data to work on
  - blockIdx: 1D, 2D, or 3D (CUDA 4.0)
  - threadIdx: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



Courtesy: NDVIA

Vector Addition: Conceptual View

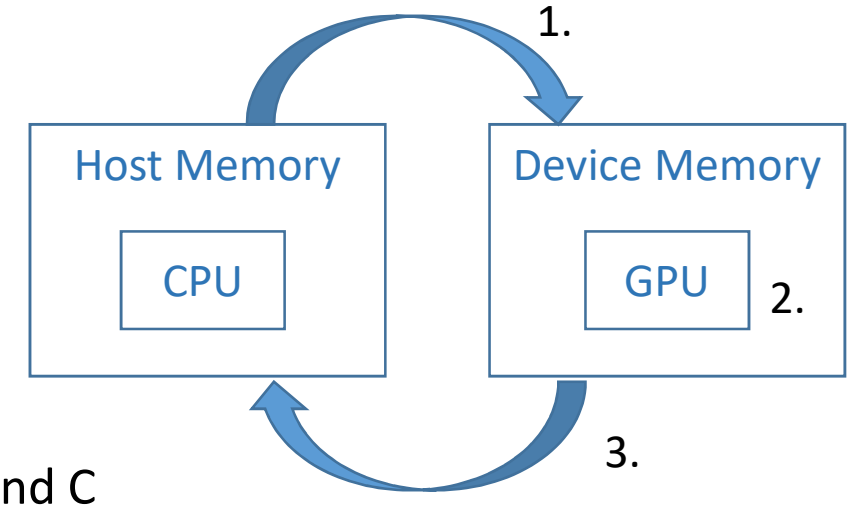


Vector Addition – Traditional C Code

```
// Compute vector sum C = A+B
void vecAdd(float* A, float* B, float* C, int n)
{
  for (i = 0, i < n, i++)
    C[i] = A[i] + B[i];
}

int main()
{
    // Memory allocation for A_h, B_h, and C_h
    // I/O to read A_h and B_h, N elements
    …
    vecAdd(A_h, B_h, C_h, N);
}
```

Heterogeneous Computing vecAdd Host Code

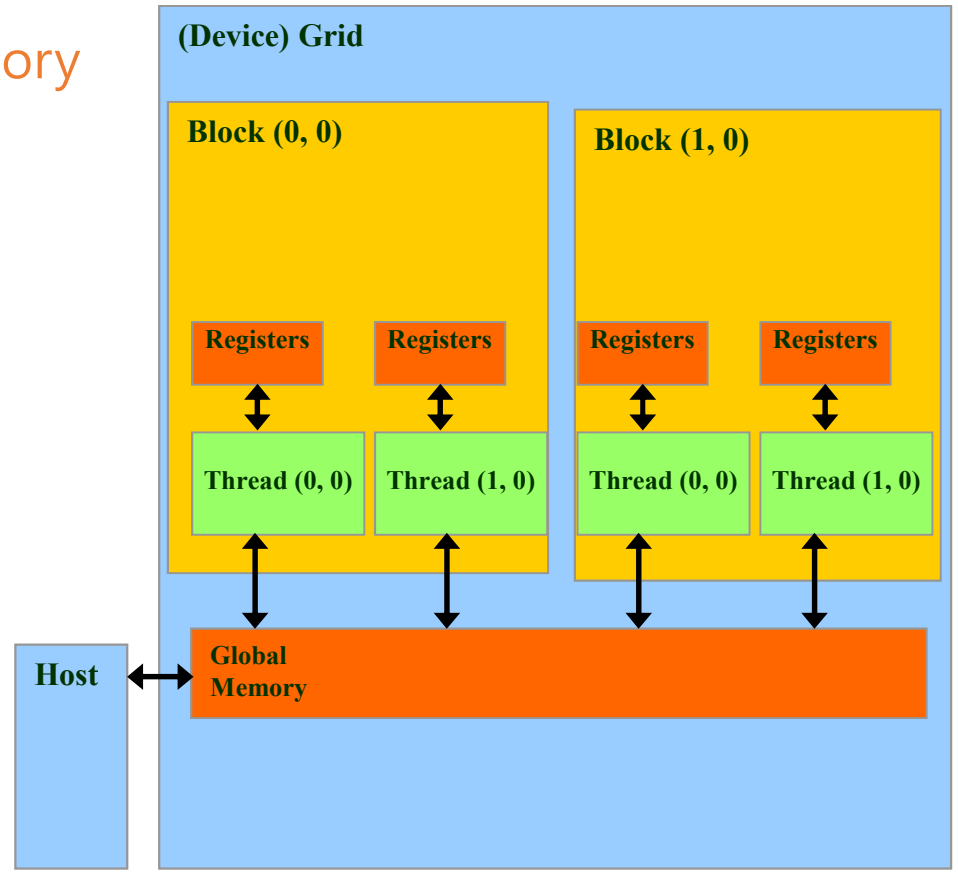
```
#include <cuda.h>
void vecAdd(float* A, float* B, int n)
{
    int size = n*sizeof(float);
    float *A_d, B_d, C_d;

    1. // Allocate device memory for A, B, and C
       // copy A and B to device memory

    2. // Kernel launch code – to have the device
       // to perform the actual vector addition

    3. // copy C from the device memory
       // Free device vectors
}
```



Partial Overview of CUDA Memories

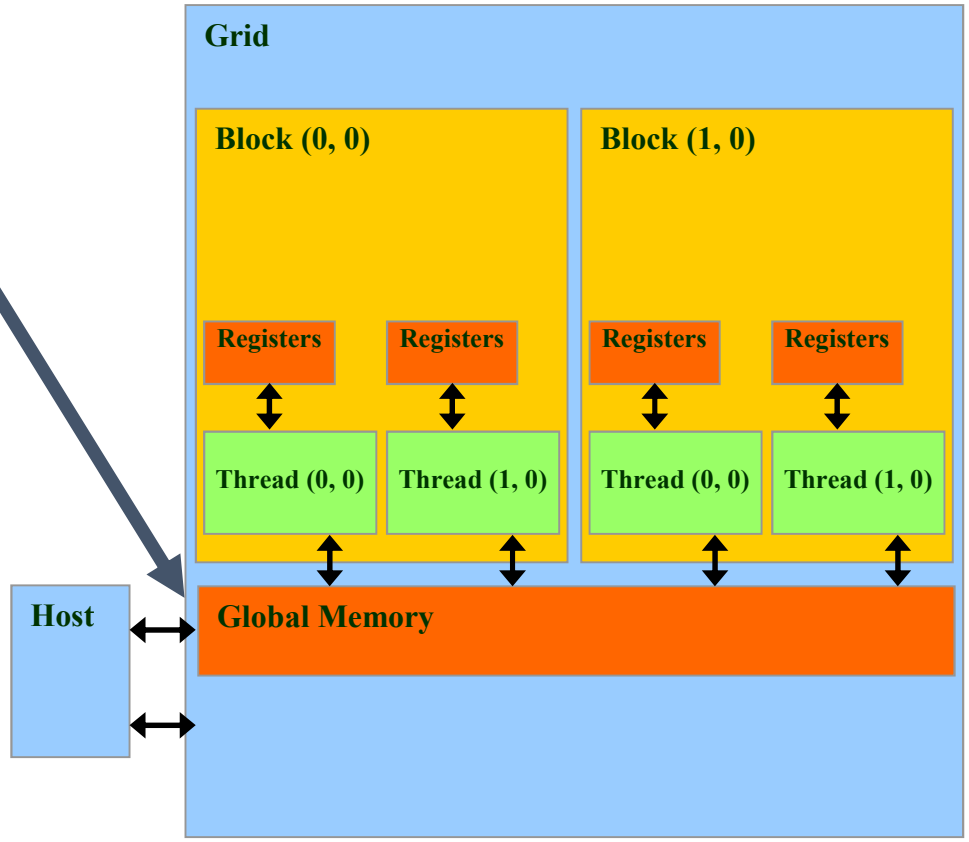
- Device code can:
  - R/W per-thread registers
  - R/W per-grid global memory

- Host code can
  - Transfer data to/from per grid global memory



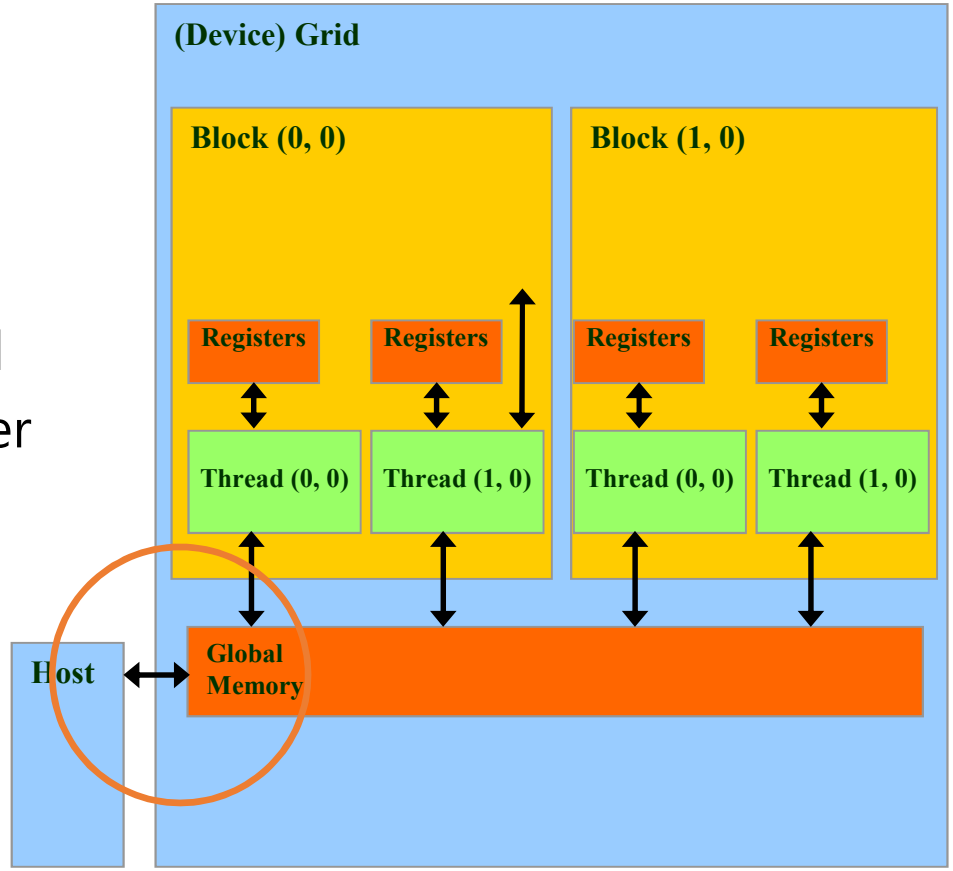
CUDA Device Memory Management API Functions

- cudaMalloc()
  - Allocates object in the device global memory
  - Two parameters
    - Address of a pointer to the allocated object
    - Size of allocated object in terms of bytes
- cudaFree()
  - Frees object from device global memory
    - Pointer to freed object



Host-Device Data Transfer API functions

- cudaMemcpy()
  - memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type/Direction of transfer
  - Transfer to device is asynchronous



```
void vecAdd(float* A, float* B, float* C, int n)
{
   int size = n * sizeof(float);
   float* A_d, B_d, C_d;

1. // Transfer A and B to device memory
    cudaMalloc((void **) &A_d, size);
    cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &B_d, size);
    cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);

     // Allocate device memory for
     cudaMalloc((void **) &C_d, size);

2.   // Kernel invocation code – to be shown later
   …
3.    // Transfer C from device to host
     cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);
       // Free device memory for A, B, C
     cudaFree(A_d); cudaFree(B_d); cudaFree (C_d);
}
```

Example: Vector Addition Kernel

Device Code

```
// Compute vector sum C = A + B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel ( float *A_d, float * B_d, float * C_d, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i<n) C_d[i] = A_d[i] + B_d[i];
}
```

```
int vecAdd(float *A, float* B, float* C, int n)
{
    // A_d, B_d, C_d allocations and copies omitted
    // Run ceil (n/256) blocks of 256 threads each
    vecAddKernel <<<ceil (n/256), 256 >>> (A_d, B_d, C_d, n);
}
```

Example: Vector Addition Kernel

```
// Compute vector sum C = A + B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel ( float *A_d, float * B_d, float * C_d, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i<n) C_d[i] = A_d[i] + B_d[i];
}
```

Host Code

```
int vecAdd(float *A, float* B, float* C, int n)
{
    // A_d, B_d, C_d allocations and copies omitted
    // Run ceil (n/256) blocks of 256 threads each
    vecAddKernel <<<ceil (n/256), 256 >>> (A_d, B_d, C_d, n);
}
```

More on CUDA Function Declarations

	Executed on the	Only callable from the
__device__ float DeviceFunction()	device	device
__global__ void KernelFunction()	device	host
__host__ float HostFunction()	host	host

- __global__ defines a kernel function
  - Each __ consists of two underscore characters
  - A kernel function must return void

- `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

```
$ nvcc hello.cu
$ a.out
```

Compiling a CUDA Program