CUDA and GPGPU Computing

Wei Wang

GPGPU Programming

- As GPU is a drastically different from CPU, programming on GPU requires extra compiler and run-time system support.
- Common programming GPGPU programming frameworks
  - CUDA by Nvidia
  - OpenCL
    - Aimed at providing support for heterogeneous computing on CPU, GPU, FPGA and DSPs
  - OpenACC
    - Aimed at providing support for heterogeneous computing with code annotation similar to OpenMP
  - OpenHMPP
    - An academia attempt for heterogeneous programming standard

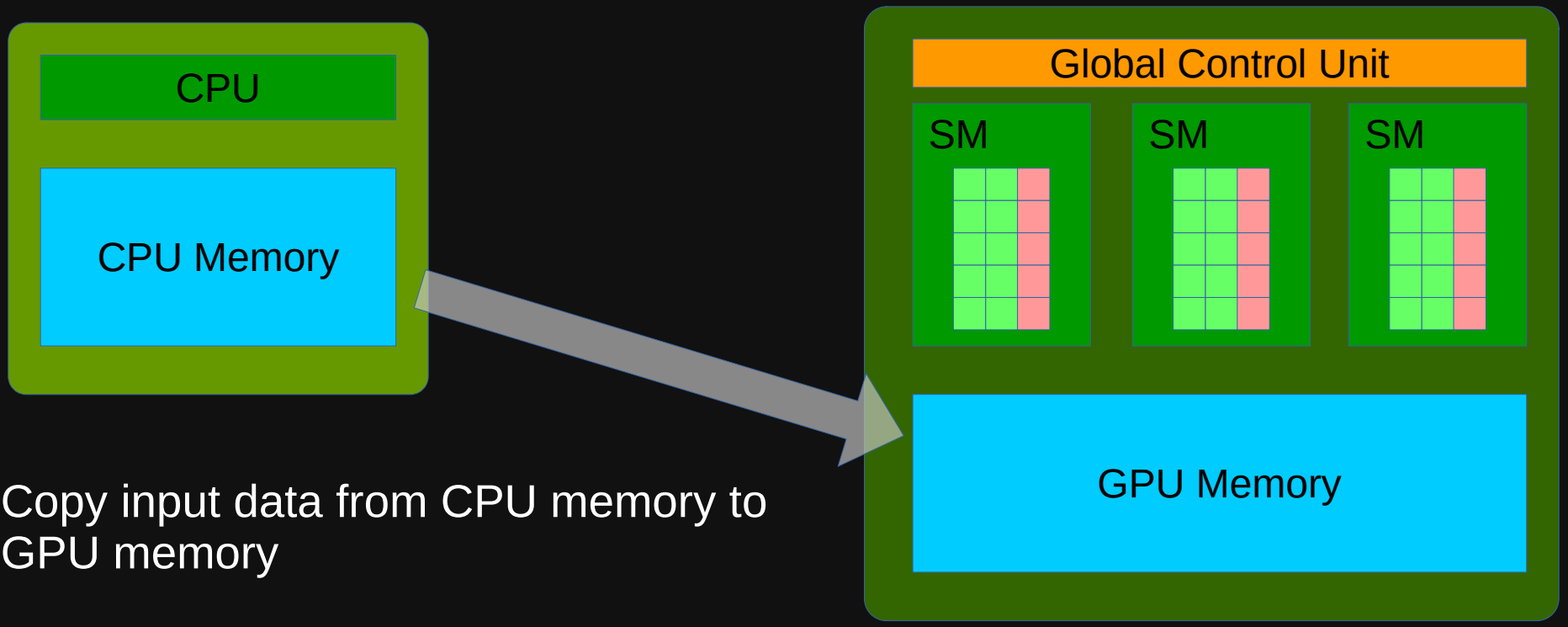
CUDA

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

GPGPU is Heterogeneous Computing

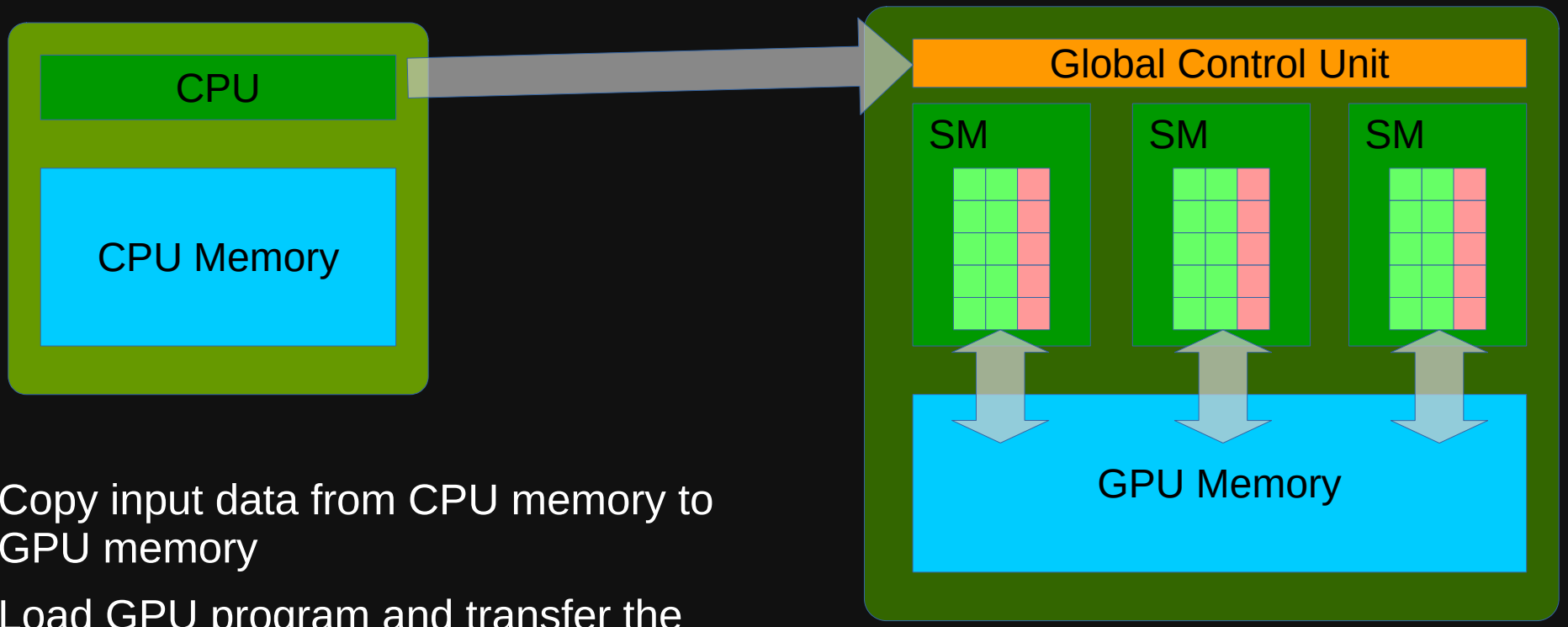
- Users interacts with CPU
  - To execute code on GPU, control has to be transferred from CPU to GPU
- CPU and GPU are two separate devices with their own memory
  - To solve a problem on GPU, data has to be transferred from CPU to GPU

GPGPU Execution Flow



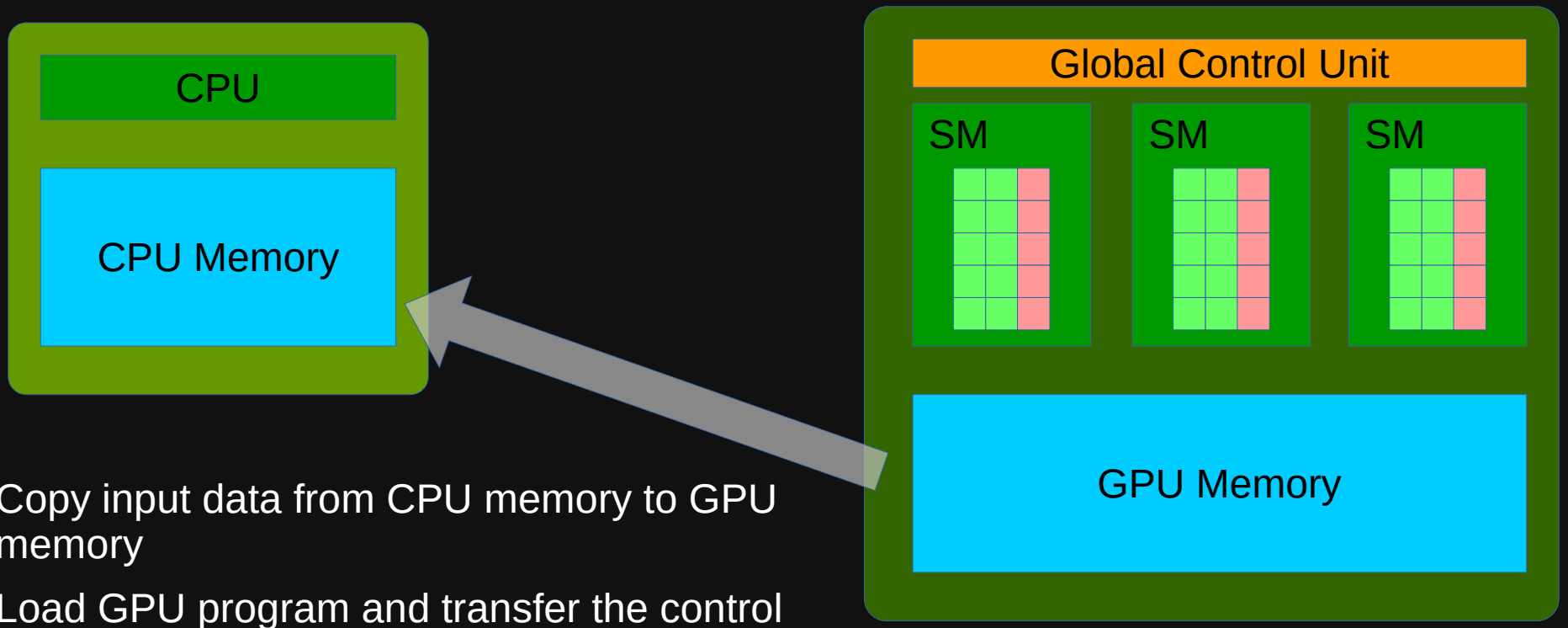
1. Copy input data from CPU memory to GPU memory

GPGPU Execution Flow cont'd



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and transfer the control to GPU to execute

GPGPU Execution Flow cont'd



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and transfer the control to GPU to execute
3. Copy results from GPU memory to CPU memory

GPGPU Execution Flow from Code Perspective

```
#define N (1024*1024)
  #define M (1000000)

  __global__ void cudakernel(float *buf)
  {
     int i = threadIdx.x + blockIdx.x * blockDim.x;
     buf[i] = 1.0f * i / N;
     for(int j = 0; j < M; j++)
        buf[i] = buf[i] * buf[i] - 0.25f;
  }

  int main()
  {
     float *data; int count = 0;
     float *d_data;
     Data = malloc(sizeof(float)*N);

     cudaMalloc(&d_data, N * sizeof(float));
     cudakernel<<<N/256, 256>>>(d_data);
     cudaMemcpy(data, d_data, N * sizeof(float),
cudaMemcpyDeviceToHost);
     cudaFree(d_data);

     int sel;
     printf("Enter an index: ");
     scanf("%d", &sel);
     printf("data[%d] = %f\n", sel, data[sel]);
  }
```

GPU Parallel Function

Run on CPU

Run on GPU

Run on CPU

Some CUDA Terms

- Host: the CPU and its memory (host memory)
- Device: the GPU and its memory (device memory)
- Device code: code that executes on GPU

“Hello World” with Device Code

```
hello_world.cu:
__global__ void mykernel(void) {
}

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

- Compile the code:
  - nvcc hello_world.cu -o hello_world.bin
  - nvcc is Nvida’s CUDA compiler
  - CUDA source code usually have extension name of “cu”, but they are just C code with CUDA extensions.
- Execute the code:
  - Just as execute any executable file:
    - $ ./hello_world.bin

The __global__ key world

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

- CUDA C/C++ keyword __global__ indicates a function that:
  - Runs on the device
  - Is called from host code
  - A device code function is usually called as a compute kernel
  - This kernel does nothing, it is only for illustration
- nvcc separates source code into host and device components
  - Device functions (e.g. mykernel()) processed by NVIDIA compiler
  - Host functions (e.g. main()) preprocessed or compiled by standard host compiler
    - gcc, cl.exe
  - Error messages may be all from nvcc.

Invoking Device Code

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

- Invoke the device code function by its name
- Triple angle brackets mark a call from host code to device code
  - Also called a “kernel launch”
- That’s all that is required to execute a function on the GPU!

Simple Sum With CUDA: The “Add” Function

- The C code to add two integers:

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

- The CUDA C code to add two integers:
  - Add the __global__ key world
  - The “add” function now can be executed on GPU

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

Simple Sum With CUDA: Passing Data/Parameters to GPU

- 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()

Simple Sum With CUDA: Passing Data/Parameters cont'd

```
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;

    // 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);

    // rest of the code on next page
```

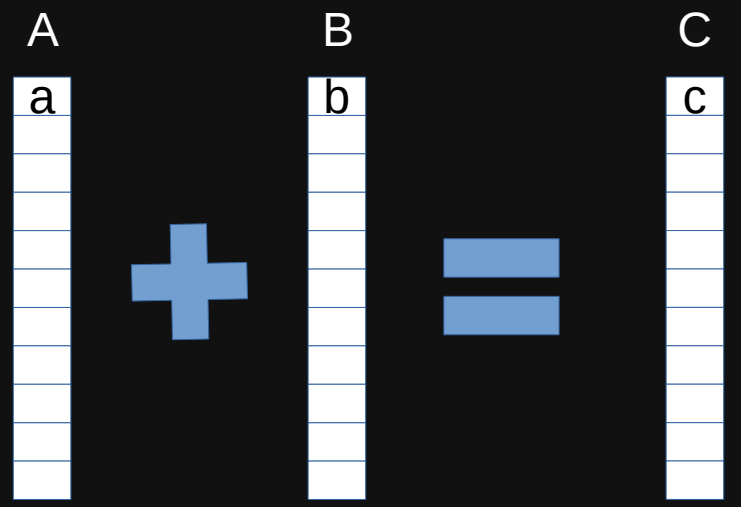
Simple Sum With CUDA: Passing Data/Parameters cont'd

```
    // continue from previous slide, after invoking add
    // on device

    // Copy result back to host
    cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);
    // Cleanup
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);

    return 0;
}
```

- In summary, to process data on GPU:
  - Allocate memory on GPU with cudaMalloc
  - Copy data to GPU with cudaMemcpy
  - Invoke the kernel with data copied to GPU
  - After kernel finishes, copy data back to CPU
  - Deallocate memory on GPU

Parallel Vector Addition with CUDA

- Vector addition:



- Parallel vector addition:
  - Reuse the “add” function we already have
  - Use the “add” function to sum one element from A and one from B
  - Execute these “add” functions in parallel on CUDA using the massive number of functional units

Parallel Vector Addition with CUDA: Running “add” in Parallel

- How do we run code in parallel on the device?

```
// Launch add() kernel on GPU
add<<<1,1>>>(d_a, d_b, d_c);
```

```
// Launch add() kernel on GPU
add<<<N,1>>>(d_a, d_b, d_c);
```

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

Parallel Vector Addition with CUDA: Make “add” Handle Arrays of Data

- Two vectors will be passed to “add”, how does “add” know which two integers it should add?
- Each parallel invocation of add() is referred to as a block
  - Each invocation can refer to its block index using blockIdx.x

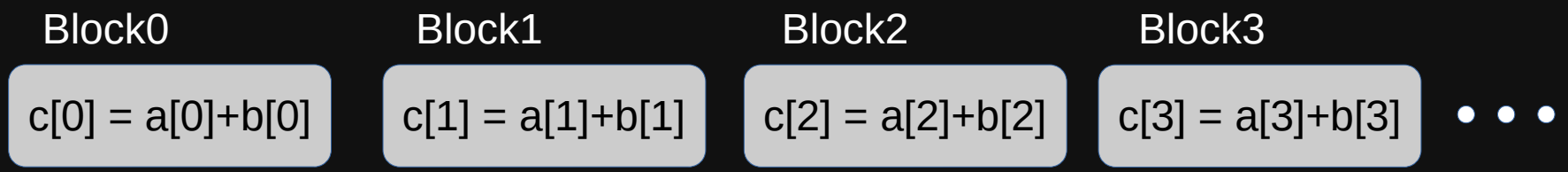
```
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- By using blockIdx.x to index into the array, each block handles a different index
  - similar with thread_id we used in Pthreads

Parallel Vector Addition with CUDA: Vector Addition on GPU

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

- With the above code, each block on the device executes in parallel:



Parallel Vector Addition with CUDA: The “main” function

- As we are adding vectors, we need to allocate space for all three vectors in “main”

```
#define N (2048*2048)
int main(void) {
    int a[N], b[N], c[N]; // 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 input values
    a = random_N_ints(); b = random_N_ints();

    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
    // rest of the code on next page
```

Parallel Vector Addition with CUDA: The “main” function cont’d

```
    // continue from previous page

    // 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
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);

    return 0;
}
```

CUDA Threads

- In CUDA, a block can be split into parallel threads
- We can also use threads, and thread indices to run “add” in parallel to add two vectors:
  - Use threadIdx instead of blockIdx

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

CUDA Threads cont'd

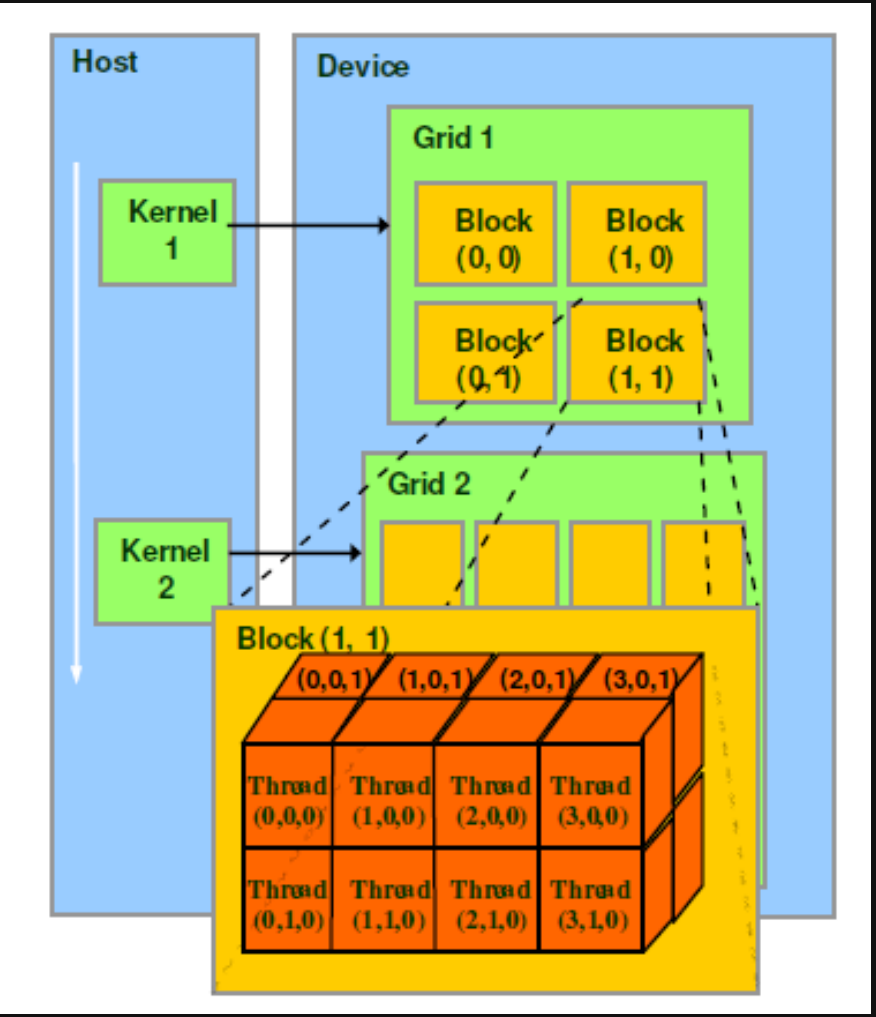
- One small change to invoke "add" in parallel:

```
// Launch add() kernel on GPU
add<<<N,1>>>(d_a, d_b, d_c);
```

```
// Launch add() kernel on GPU
add<<<1,N>>>(d_a, d_b, d_c);
```

Grids, Blocks and Threads

- CUDA partition the parallel execution into grids, thread blocks and threads
  - A grid is usually associated with the launch of a kernel
  - A grid can have at most 65535 blocks
  - A block can have at most 1024 threads
- GPU scheduler internally handles the mapping of blocks/threads to SMs
- Block and threads somewhat reflect the original use of GPU – graphic processing
  - Images are processed in blocks
  - Each block is parallel processed by threads
  - GPU scheduler is designed to understand this image processing requirement
  - Many scientific problems also share this structure



Grids, Blocks and Threads cont’d

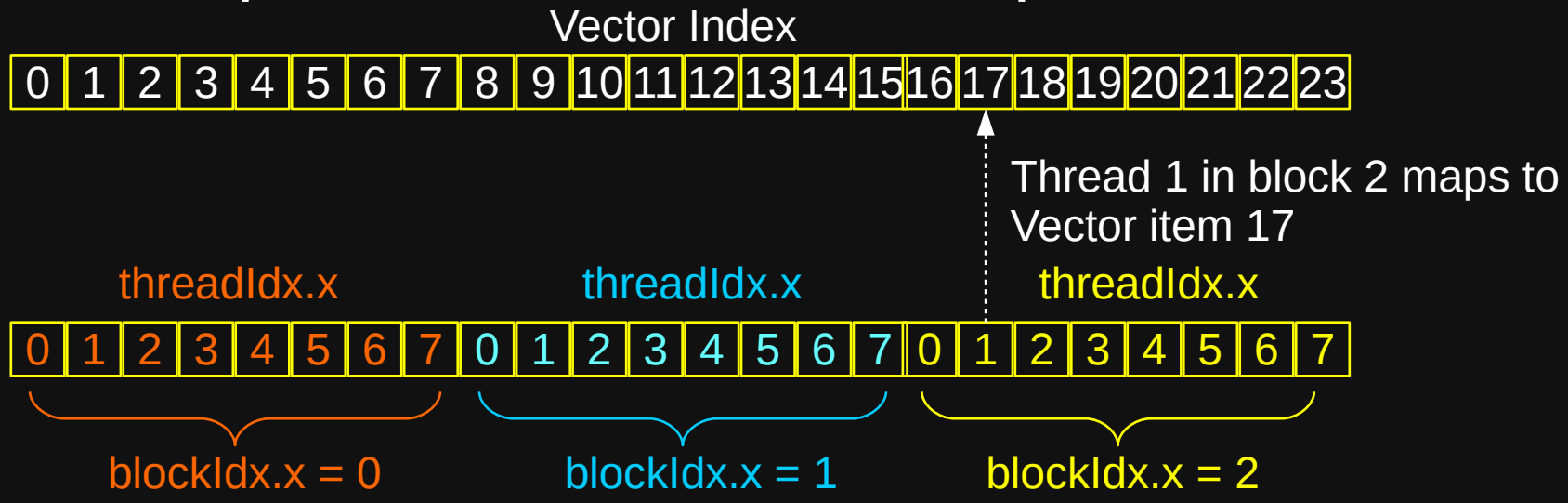
- Thread: Sequential execution unit
  - All threads execute same sequential program
  - Threads execute in parallel
- Threads Block: a group of threads
  - Executes on a single Streaming Multiprocessor (SM)
  - Threads within a block can cooperate
    - Light-weight synchronization
    - Data exchange
- Grid: a collection of thread blocks
  - Thread blocks of a grid execute across multiple SMs
  - Thread blocks do not synchronize with each other
  - Communication between blocks is expensive

Indexing Arrays with Blocks and Threads

- So far we have been using only one thread per block or using only one block
- For the vector addition problem, how does a thread determine its correct vector index if multiple blocks each multiple threads are used?

Indexing Arrays with Blocks and Threads: An Example

- For example, assume 8 threads per block:



- For the second thread (thread 1) in the third block (block 2), it maps to index:
  - vector_index = blockIdx.x * block_size + threadIdx.x
    = 2 * 8 + 1 = 17

Indexing Arrays with Blocks and Threads: The New Kernel

- Use the built-in variable blockDim.x for threads per block:

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

- Note that, the index computed in this way can also be viewed as the global index of a thread when all threads from all blocks are considered.
- There is also a gridDim object that gives the number of blocks within a grid.

Indexing Arrays with Blocks and Threads: The New “main”

- We can now use multiple blocks and threads

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main(void) {
    int a[N], b[N], c[N]; // 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 input values
    a = random_N_ints(); b = random_N_ints();

    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
    // rest of the code on next page
```

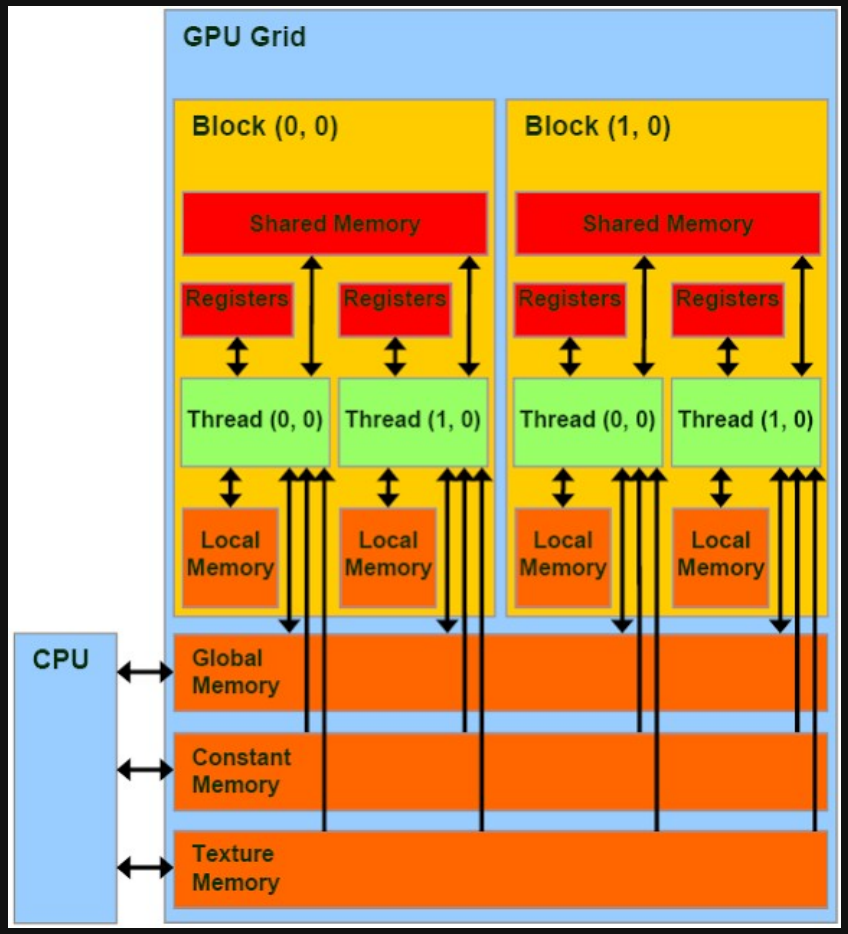
Indexing Arrays with Blocks and Threads: The New “main” cont’d

```
    // continue from previous page

    // Launch add() kernel on GPU with N blocks
    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
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);

    return 0;
}
```

The Rational Of Having Threads

- Threads seem unnecessary
  - They add a level of complexity
  - What do we gain?
- Unlike parallel blocks, threads have mechanisms to:
  - Communicate
  - Synchronize
- For graphic processing, it usually required to communicate within a block
  - Global communication is less common

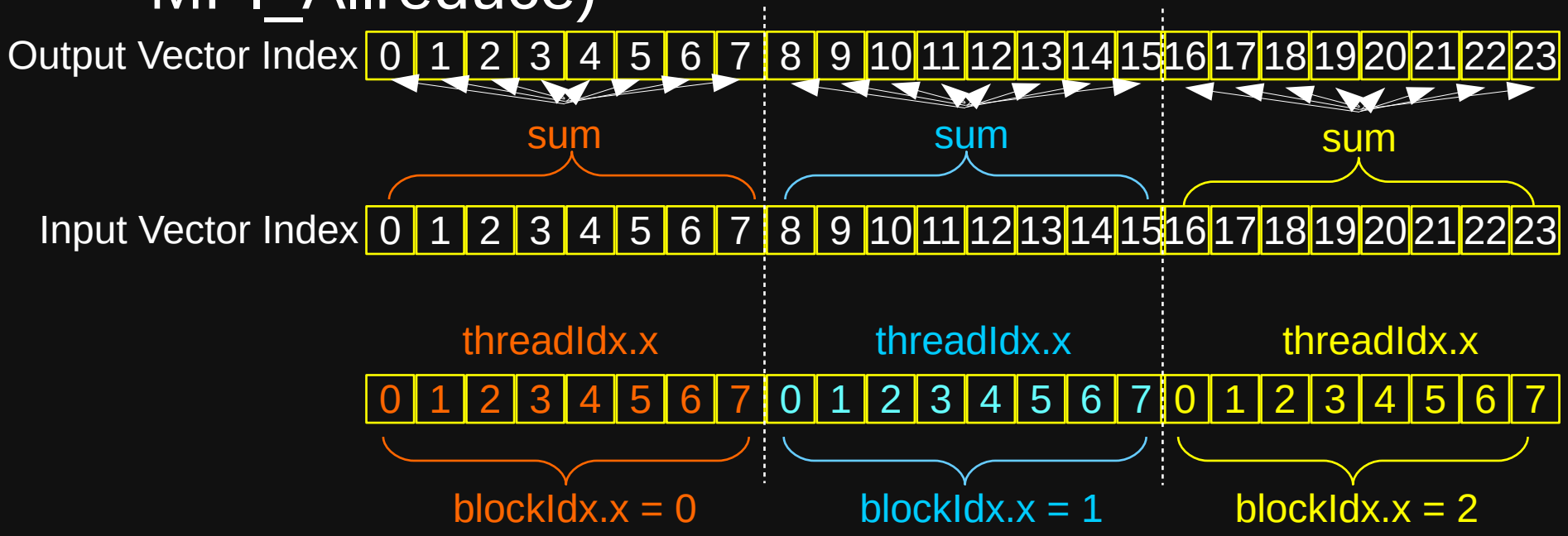
Thread Communications



- Threads within a block is scheduled by the GPU to run on SMs that are directly connected to the same “shared memory”
- Other synchronization mechanisms are also possible among these SMs

An Example of Using Shared Memory: All-Reduction

- All reduction with sum: sum the values (of a vector) processed by the threads of a block, and save the sum into an output vector (similar to MPI_Allreduce)



An Example of Using Shared Memory: All-Reduction Algorithm

- Idea of the algorithm:
  - Copy the part of array that has to be summed into shared memory
  - Each thread will then read the data from shared memory, sum them and store the result back to the out vector
- Why shared memory? Why not directly sum the values from vector?
  - Each thread has to read all values to sum them
  - It takes more time to read the values from main memory than from shared cache

An Example of Using Shared Memory: All-Reduction Kernel

- The code for the kernel:

```
__global__ void all_reduce(int *in, int *out) {

    int index = blockIdx.x * blockDim.x + threadIdx.x;
    // allocate an array in shared memory
    __shared__ int temp[BLOCK_SIZE];

    // copy one value into shared memory
    temp[threadIdx.x] = in[index];

    // sum all values up
    int sum = 0;
    for(int i = 0; i < blockDim.x; i++)
        sum += temp[i];

    // output the sum to out array
    out[index] = sum;
}
```

__shared__ keyword declares a variable in shared memory

An Example of Using Shared Memory: All-Reduction Kernel cont'd

- The code for the kernel:

```
__global__ void all_reduce(int *in, int *out) {

    int index = blockIdx.x * blockDim.x + threadIdx.x;
    // allocate an array in shared memory
    __shared__ int temp[BLOCK_SIZE];

    // copy on value into shared memory
    temp[threadIdx.x] = in[index];

    // sum all values up
    int sum = 0;
    for(int i = 0; i < blockDim.x; i++)
        sum += temp[i];

    // output the sum to out array
    out[index] = sum;
}
```

Race condition: if some threads haven't copied their values then we cannot do sum. We need a barrier here.

An Example of Using Shared Memory: All-Reduction Kernel cont'd

- CUDA's barrier is: void __syncthreads();

```
__global__ void all_reduce(int *in, int *out) {

    int index = blockIdx.x * blockDim.x + threadIdx.x;
    // allocate an array in shared memory
    __shared__ int temp[BLOCK_SIZE];

    // copy on value into shared memory
    temp[threadIdx.x] = in[index];
    __syncthreads(); // barrier
    // sum all values up
    int sum = 0;
    for(int i = 0; i < blockDim.x; i++)
        sum += temp[i];

    // output the sum to out array
    out[index] = sum;
}
```

Starting Kernels with Multi-Dimensional Blocks/Threads

- If the block count and threads per block are integers when launching a kernel, the blocks and threads are organized as a 1D vector space.
- To launch 2D or 3D blocks and threads:

```
dim3  dimGrid(8, 32);  // a grid has 8*32 = 256 blocks
dim3  dimBlock(4, 5, 6); // a block has 4*5*6 = 120 threads
myKernel <<<dimGrid, dimBlock>>>(...);
```

Dynamic Shared Memory

- In the previous example, we statically declared an array in shared memory
  - i.e., the shared memory is allocated with a compile-time determined size
  - Static shared memory allocation is usually limited to the case where block count and thread count are known before hand
- There are cases where block/thread counts are unknown at compile time, and the size of required shared memory is only known at run-time

Allocating and Using Shared Memory Dynamically

- The size of dynamically allocated shared memory are passed to GPU as kernel launch parameter
- Dynamically allocated memory is declared in a kernel with extern key word

```
__global__ myKernel(...){
    …
    extern __shared__ int a[];
    …
}

Main(){
    …
    Mykernel<<<gridDim, blockDim, sharedMemSize >>>(...)
    …
}
```

A Note on the Shared Memory Size

- Shared memory has limited size.
- If a shared object requires more memory than the physical size of the shared memory, the shared object has to be put in GPU global memory.

Device and Host Key words

- __device__ keyword specifies a function to be invoked from and executed on GPU
- __host__ keyword specifies a function to be invoked from and executed on CPU
- A function can have both __device__ and __host__ keywords.
- __global__ keyword specifies a function to be invoked from CPU and executed on GPU

Debugging GPU Programs

- “printf” works in CUDA sm20 and higher architecture models.
- After executing a kernel, call CUDA APIs to check for errors
  - cudaGetLastError
  - cudaPeekAtLastError
  - CudaGetErrorString
- CUDA-GDB provides debugging support with GUI on Linux and Mac
- NVIDIA has Nsight debugger for Visual Studio and Eclipse

Acknowledgement

- Slides are based on “CUDA C/C++ BASICS”, Cyril Zeller, NVIDIA Corporation, Supercomputing 2011