Lecture 02 CUDA Programming Model Basics

Outline of CUDA Basics

- Basic Kernels and Execution on GPU
- Basic Memory Management
- Coordinating CPU and GPU Execution

- See the Programming Guide for the full API

CUDA Programming Model

- Parallel code (kernel) is launched and executed on a device by many threads
  - Kernel is equivalent to the thread function in pthreads.
  - Kernel is executed by thousands of threads on GPU at a time in parallel.
  - Kernel is the entry point of the parallel code.
    - Kernel is equivalent to the main() function in sequential program.

CUDA Programming Model

- A kernel launch generates thousands of threads on GPU hardware.
- Launches are hierarchical,
  - Threads are grouped into blocks
  - Blocks are grouped into grids
- Familiar serial code is written for a thread
  - Each thread is free to execute a unique code path (SIMT)
    - This is programming model, not hardware model.
  - Built-in thread and block ID variables

Execution of a CUDA program

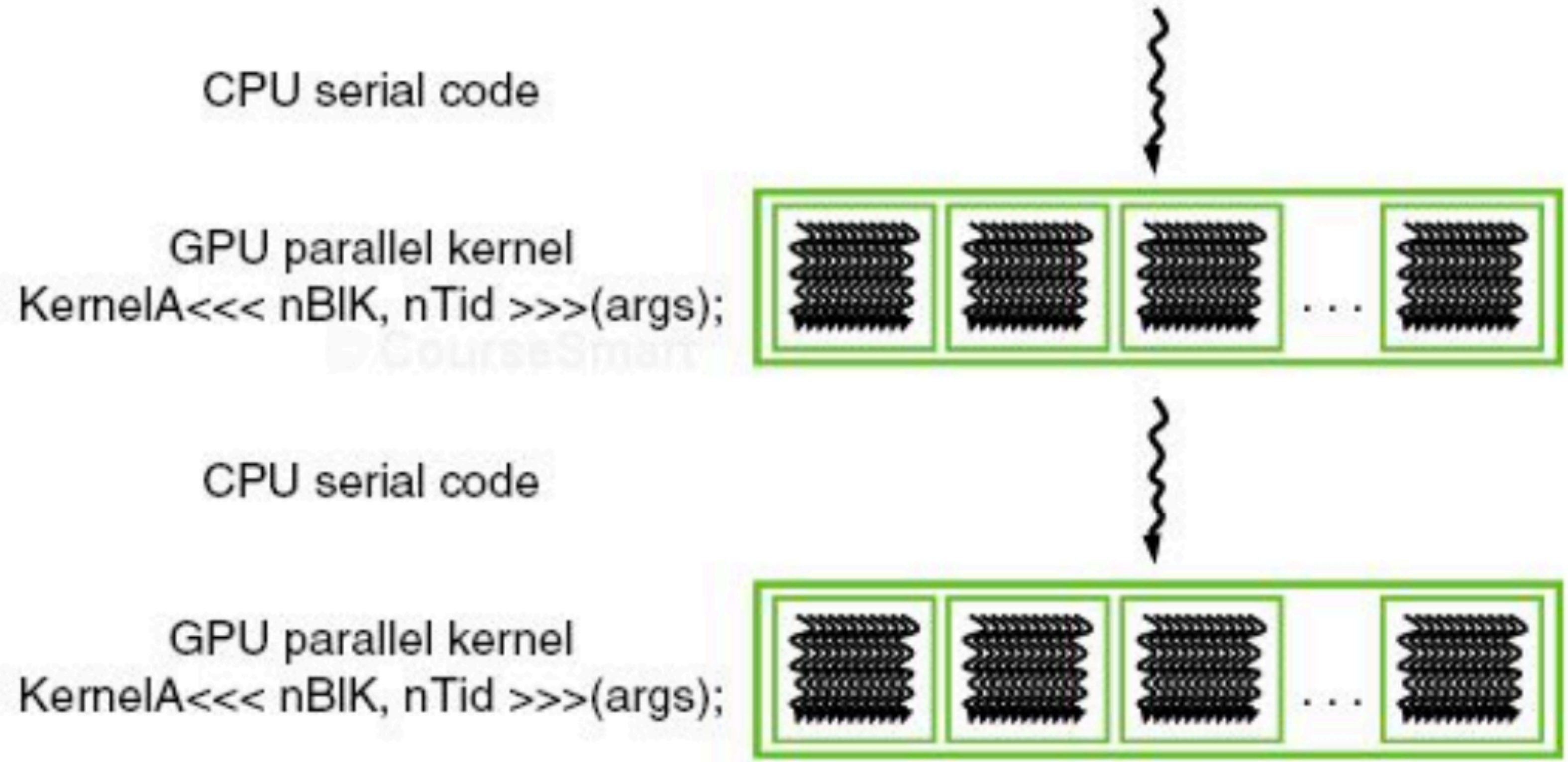
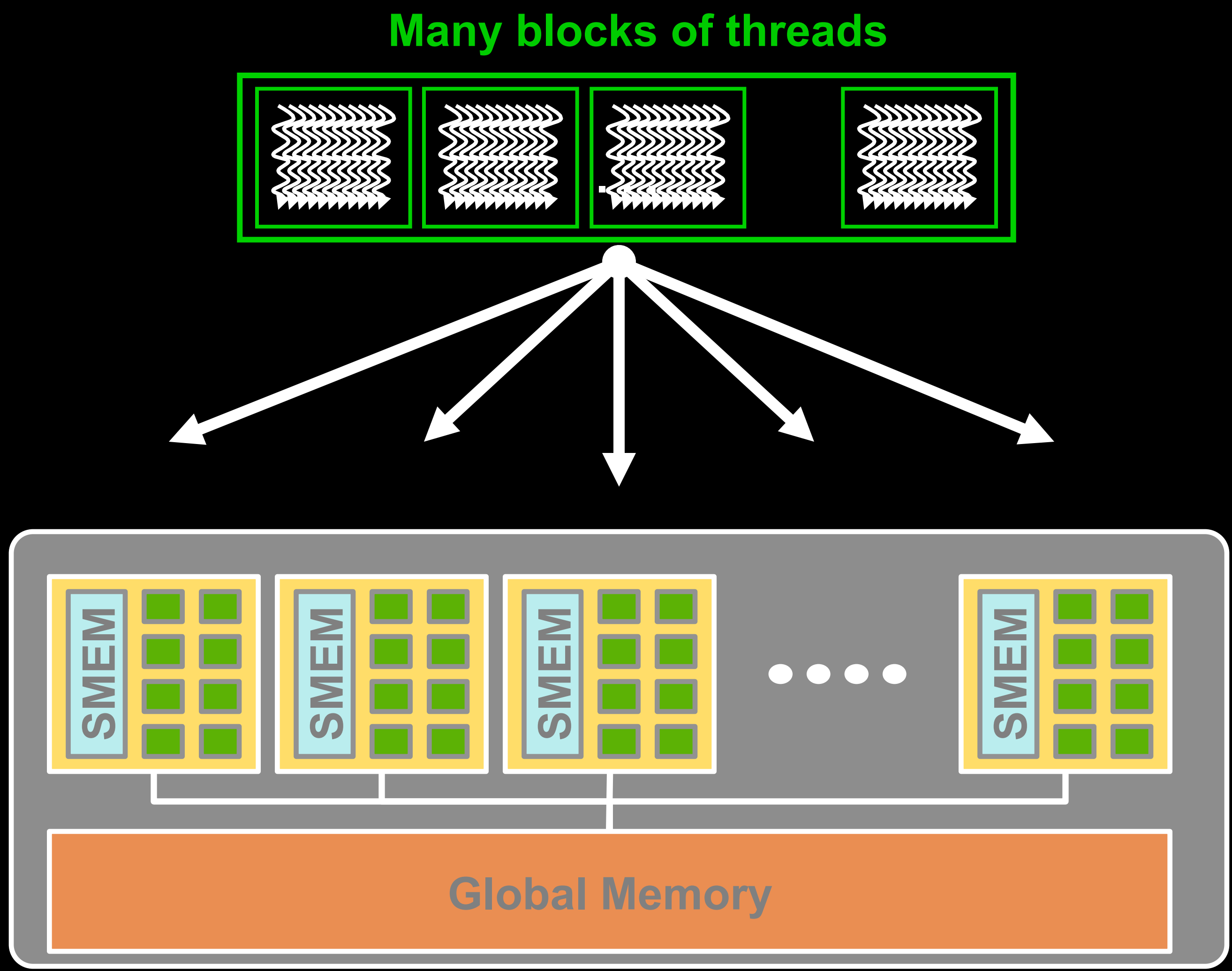


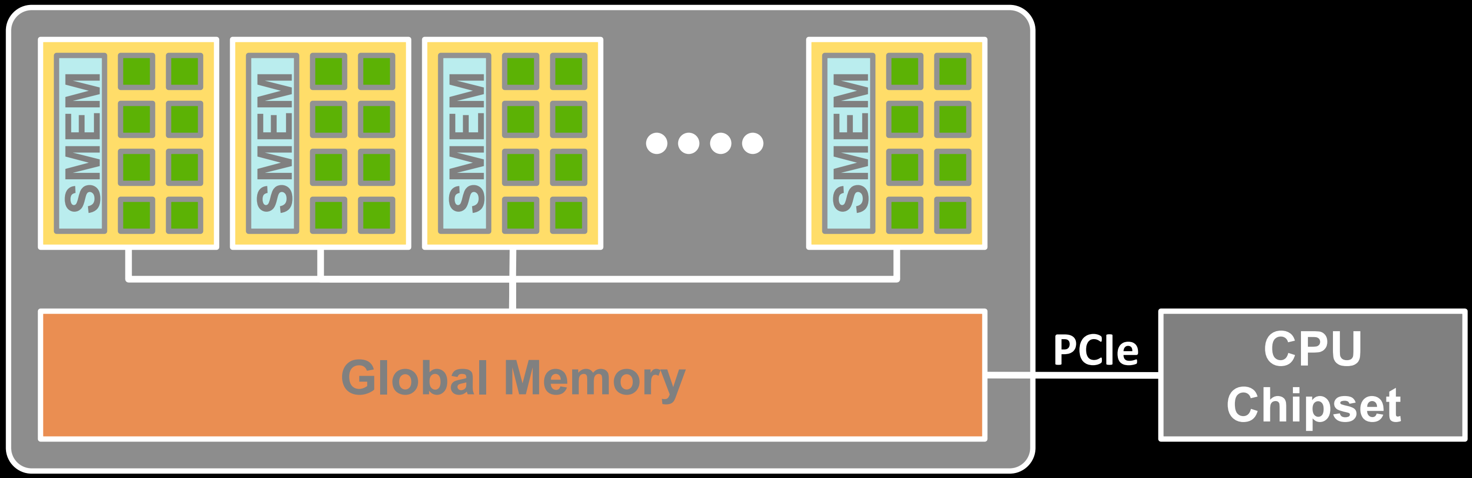
FIGURE 3.3

Execution of a CUDA program.

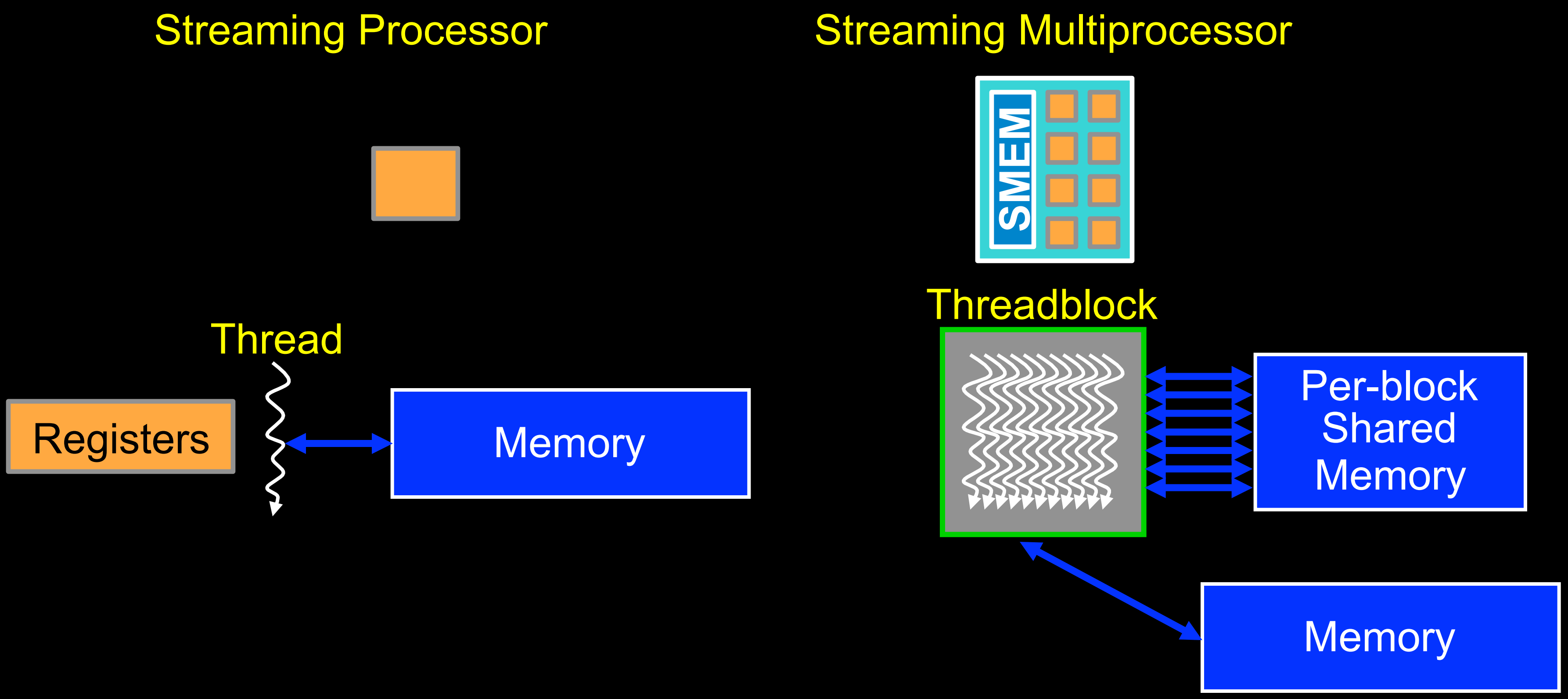
Execution of a CUDA program



High Level View

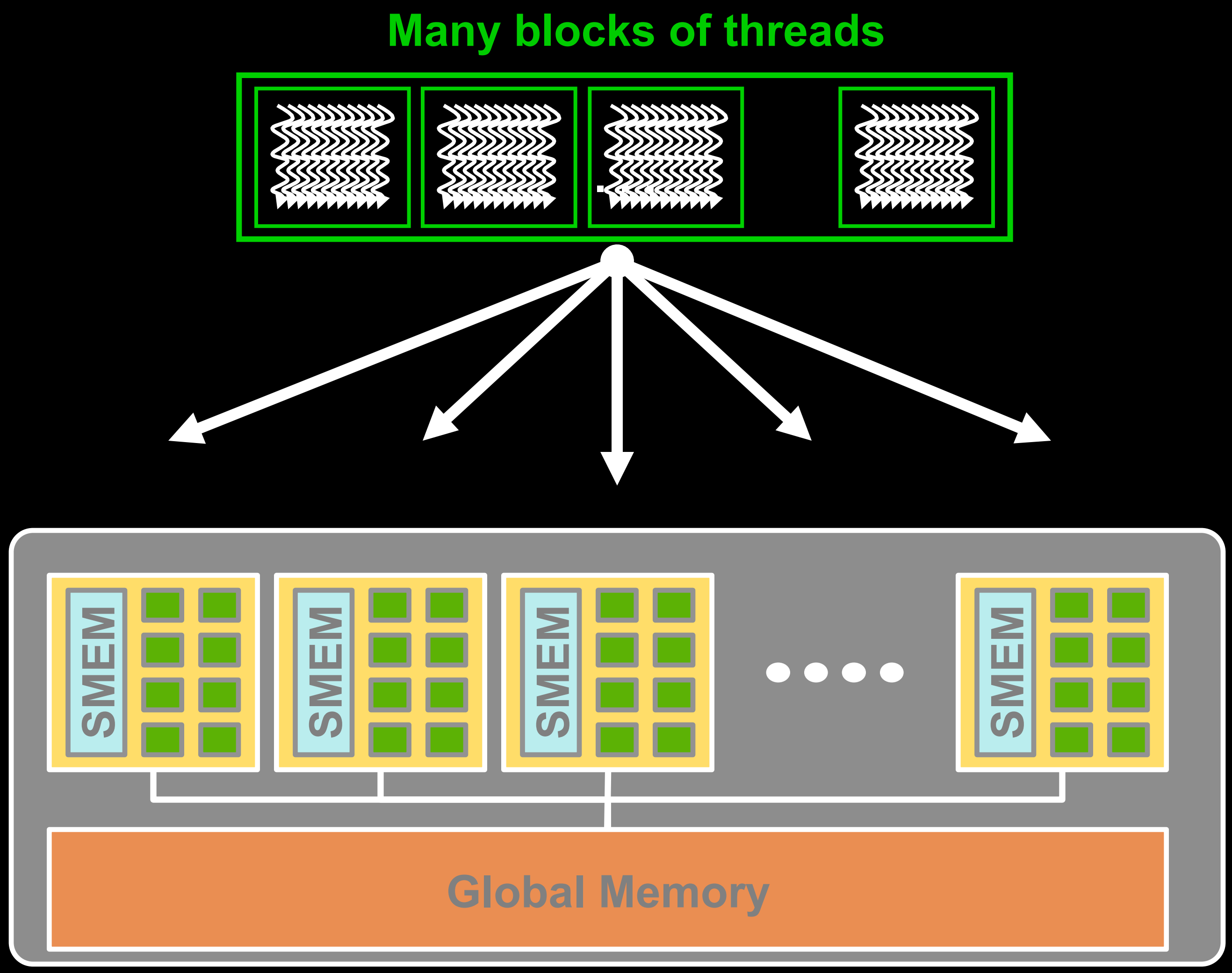


Blocks of threads run on an SM



- Latency Costs are significant,
  - when transferring data between host and device memory
  - accessing global memory from a cuda kernel program might be 100 time slower than accessing shared memory

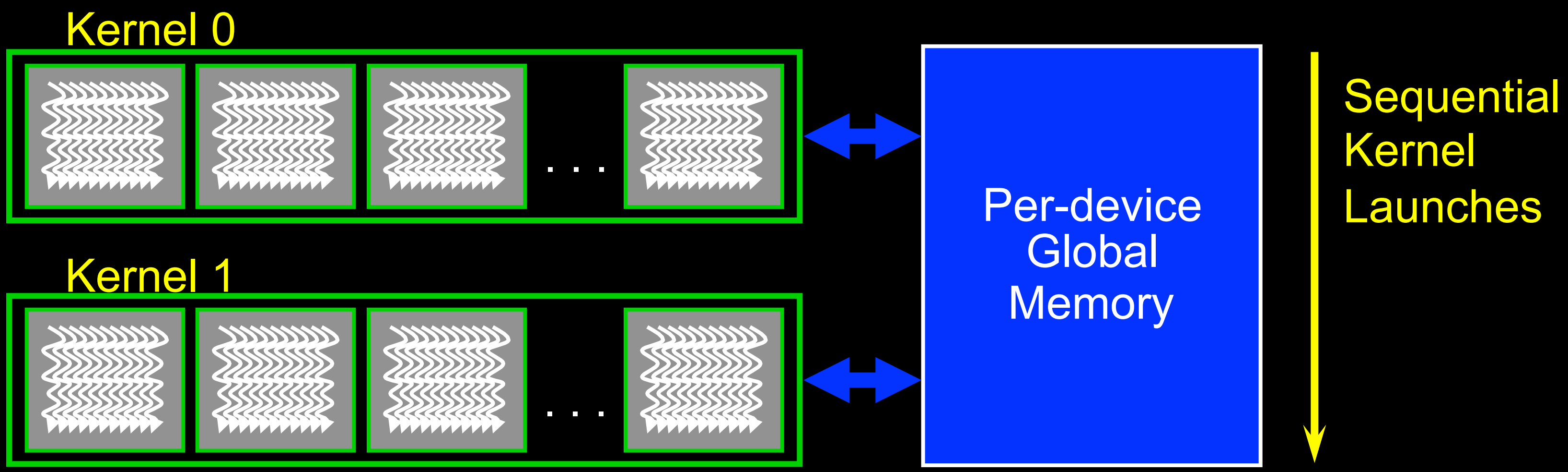
Whole grid runs on GPU



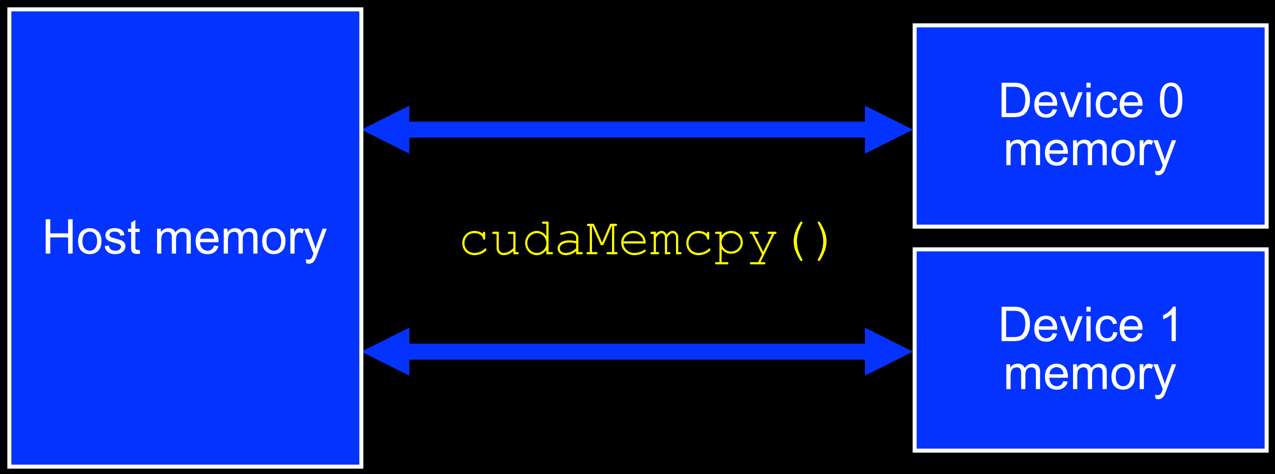
Thread Hierarchy

- Threads launched for a parallel section are partitioned into thread blocks
  - Grid = all blocks for a given launch
- Thread block is a group of threads that can:
  - Synchronize their execution
  - Communicate via shared memory

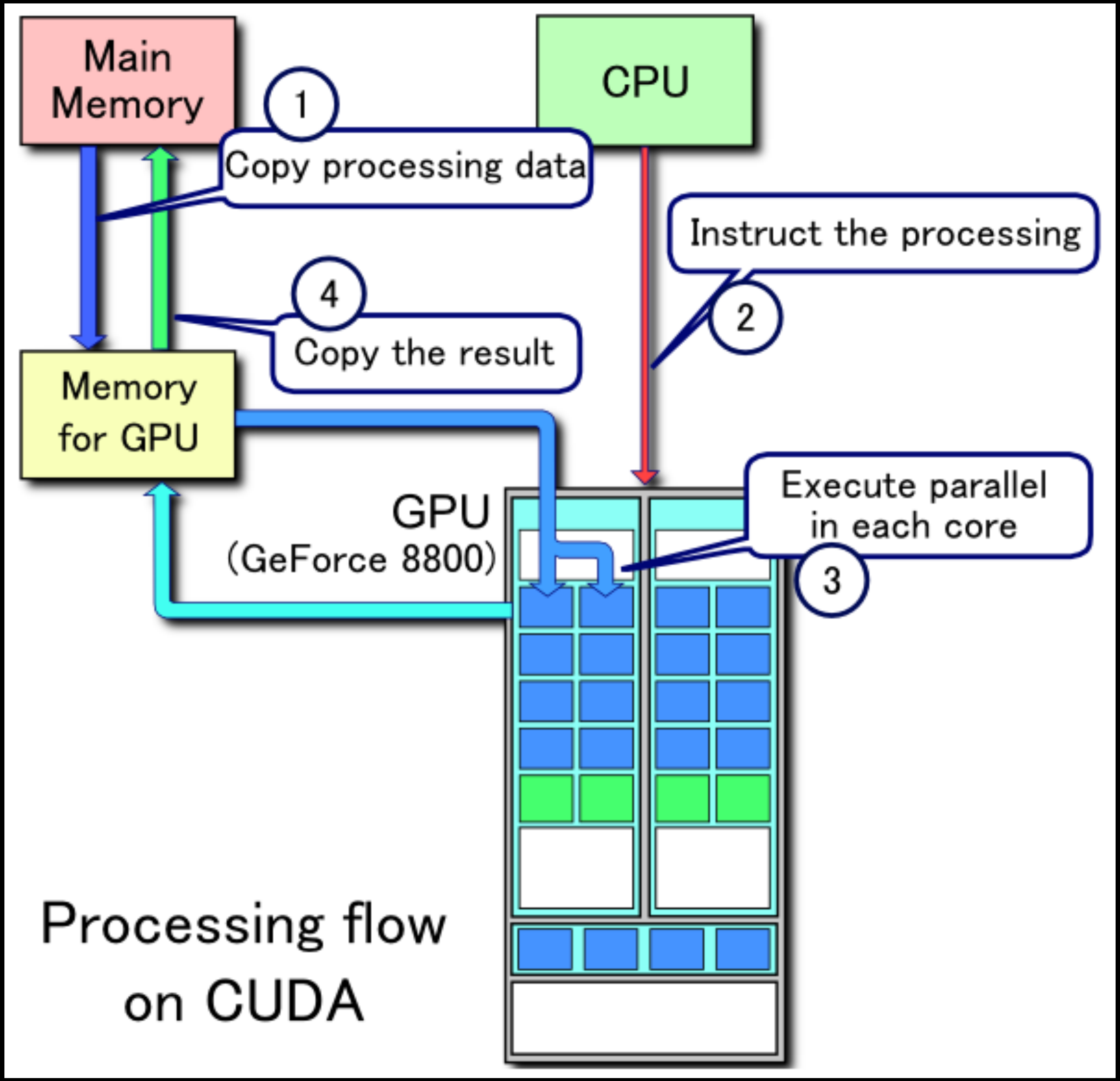
Memory Model



Memory Model



General Processing Flow on GPU



Example: Vector Addition

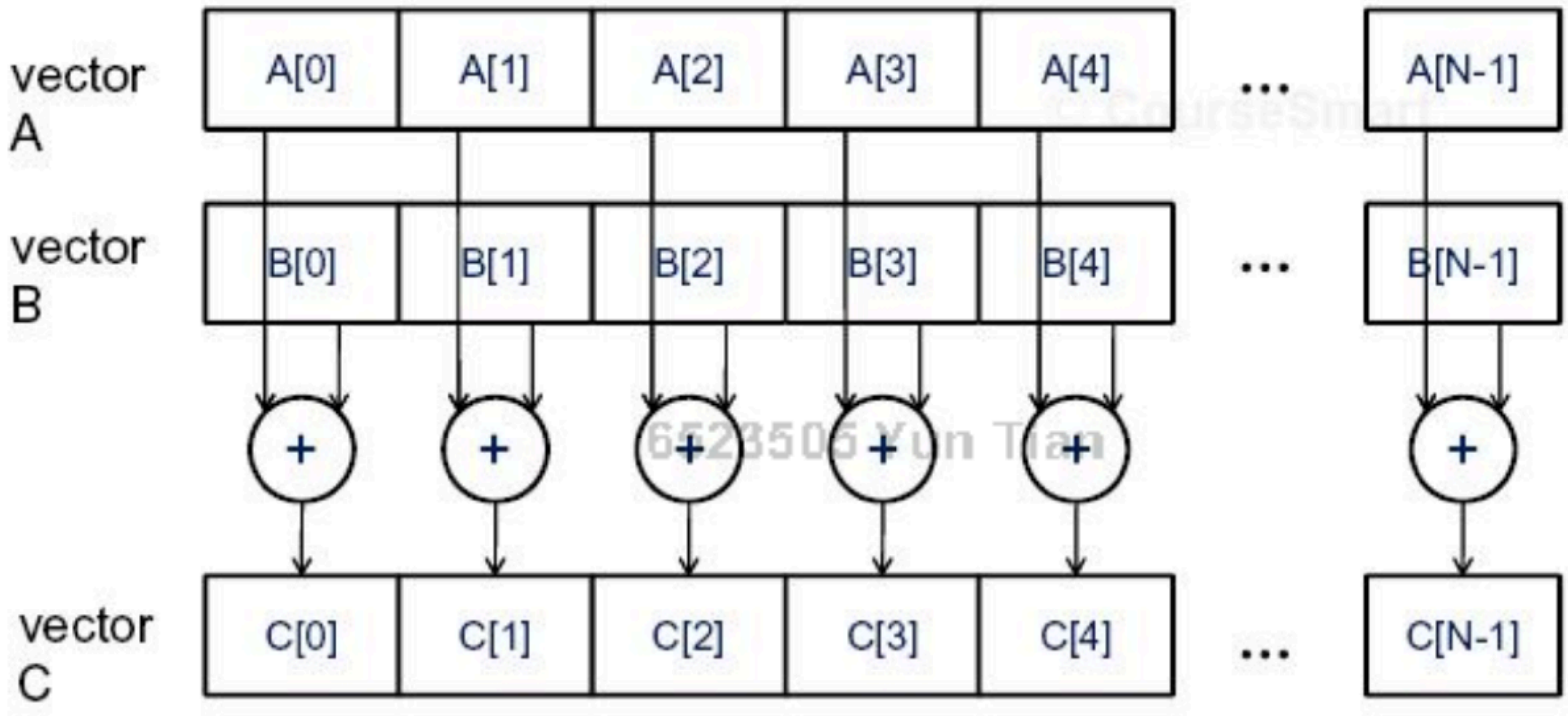


FIGURE 3.1

Data parallelism in vector addition.

Example: Vector Addition Kernel

Device Code

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

int main()
{
    // Run grid of N/256 blocks of 256 threads each
    vecAdd<<< n/256, 256>>>(d_A, d_B, d_C, n);
}
```

Example: Vector Addition Kernel

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

Host Code

```
int main()
{
    // Run grid of N/256 blocks of 256 threads each
    vecAdd<<< n/256, 256>>>(d_A, d_B, d_C, n);
}
```

Example: Vector Addition Kernel

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

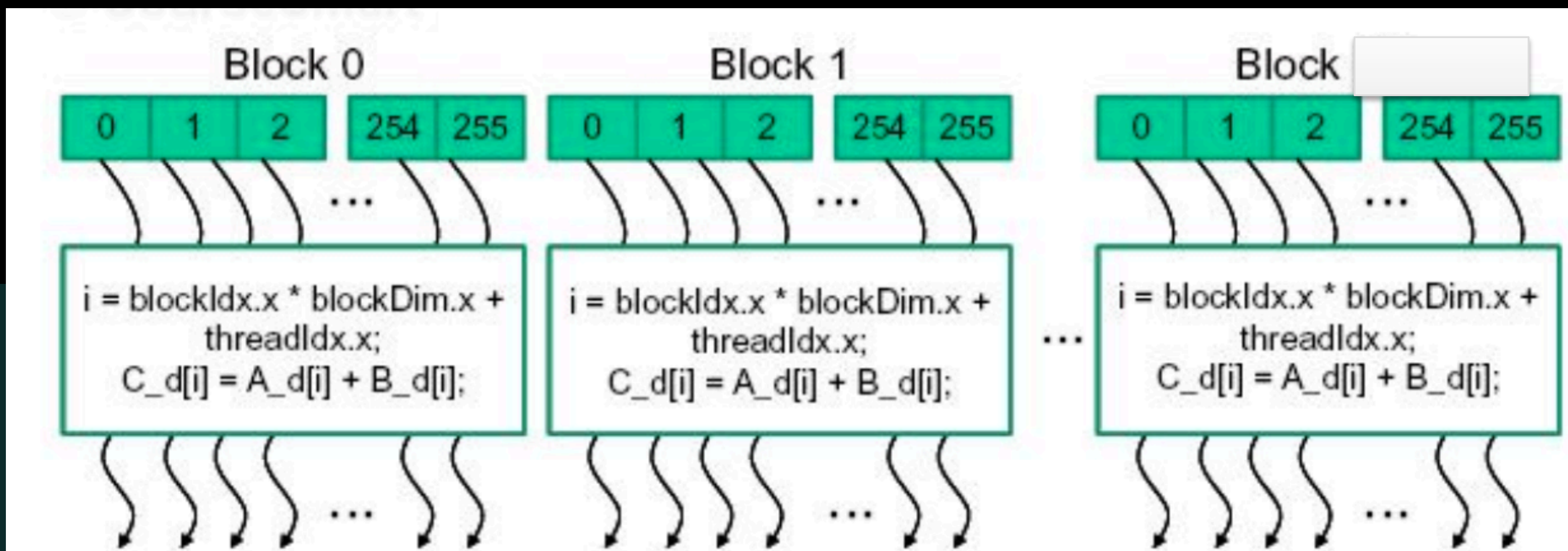


FIGURE 3.10

All threads in a grid execute the same kernel code.

Example: Vector Addition Kernel

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C, int
    n)
{
     int i = threadIdx.x + blockDim.x * blockIdx.x;
     if( i < n )
        C[i] = A[i] + B[i];
}
//Each thread gets a unique i value stored locally in
    hardware register. We use i as a global index to a spot
    in vector A, B, and C.
//For block 0 (where blockIdx.x = 0), thread 0(threadIdx.x
    = 0) in block 0 get i = 0, thread 1 in block 0 get i =
    1,….So the first block of threads will process elem in
    range of 0 to 255, because blockDim.x is 256, set by
    kernel launch.
```

Example: Vector Addition Kernel

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C,
   int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if( i < n )
      C[i] = A[i] + B[i];
}
.
//For block 1, it process elements in range of 256
   to 511 in the array of A, B and C.
//threadIdx.x, blockDim.x, blockIdx.x are built-in
   constant, set by GPU after kernel is launched.
   Keep it unchanged during that kernel execution.
// Why use if ( i < n )?
```

Example: Host code for `vecAdd`

```
#define N 1029
// allocate and initialize host (CPU) memory
float *h_A = …,   *h_B = …; *h_C = …(empty)

// allocate device (GPU) memory
float *d_A, *d_B, *d_C;
cudaMalloc( (void**) &d_A, N * sizeof(float));
cudaMalloc( (void**) &d_B, N * sizeof(float));
cudaMalloc( (void**) &d_C, N * sizeof(float));

// copy host memory to device
cudaMemcpy( d_A, h_A, N * sizeof(float),
    cudaMemcpyHostToDevice) );
cudaMemcpy( d_B, h_B, N * sizeof(float),
    cudaMemcpyHostToDevice) );

// execute grid of N/256 blocks of 256 threads each
vecAdd<<<(ceil)(N/(float)256), 256>>>(d_A, d_B, d_C, N);
```

Example: Host code for `vecAdd (2)`

```
// copy result back to host memory
cudaMemcpy( h_C, d_C, N * sizeof(float),
    cudaMemcpyDeviceToHost) );

// do something with the result…

// free device (GPU) memory
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
```

Kernel Variations and Output

```
// Assume a[ ] has 16 elements, the kernel generate a 1D grid of 4 blocks and each
block has 4 threads.
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = 7;
}
```

Output: 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = blockIdx.x;
}
```

Output: 0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = threadIdx.x;
}
```

Output: 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3

Code executed on GPU

- C/C++ with some restrictions:
  - Can only access GPU memory
  - No variable number of arguments
  - No static variables
  - No recursion
  - No dynamic polymorphism

- Must be declared with a qualifier:
  - __global__ : launched by CPU,
    cannot be called from GPU must return void
  - __device__ : called from other GPU functions,
    cannot be called by the CPU
  - __host__ : can be called by CPU
  - __host__ and __device__ qualifiers can be combined
    - sample use: overloading operators

Memory Spaces

- CPU and GPU have separate memory spaces
  - Data is moved across PCIe bus
  - Use functions to allocate/set/copy memory on GPU
    - Very similar to corresponding C functions
- Pointers are just addresses
  - Can't tell from the pointer value whether the address is on CPU or GPU
  - Must exercise care when dereferencing:
    - Dereferencing CPU pointer on GPU will likely crash
    - Same for vice versa

GPU Memory Allocation / Release

- Host (CPU) manages device (GPU) memory:
  - cudaMalloc (void ** pointer, size_t nbytes)
  - cudaMemset (void * pointer, int value, size_t count)
  - cudaFree (void* pointer)

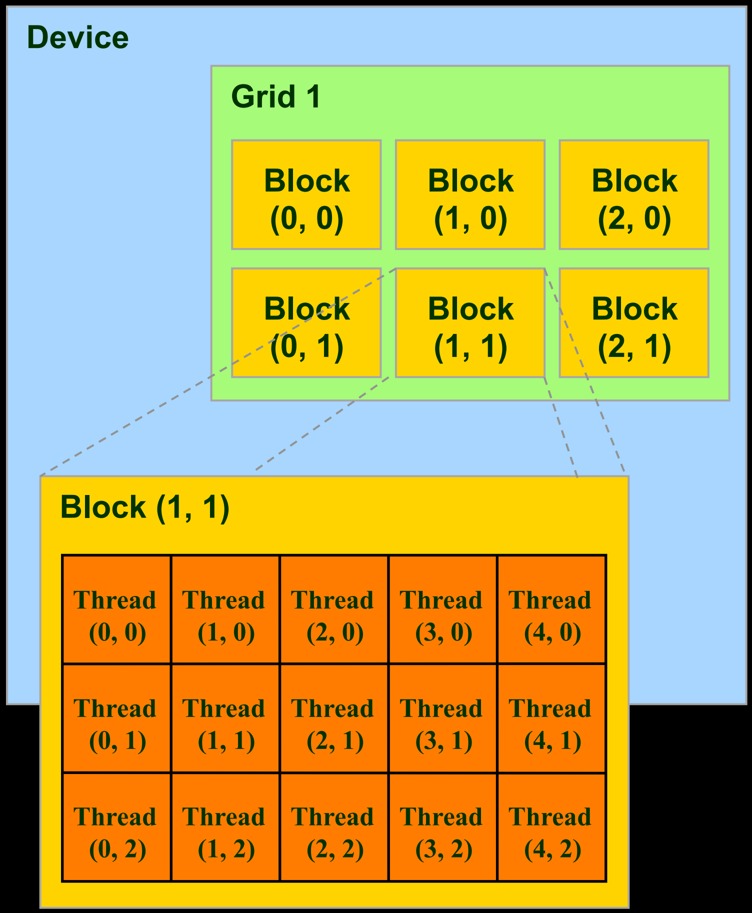
```
int n = 1024;
int nbytes = 1024*sizeof(int);
int * d_a = 0;
cudaMalloc( (void**)&d_a,  nbytes );
cudaMemset( d_a, 0, nbytes);
cudaFree(d_a);
```

Data Copies

- cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);
  - returns after the copy is complete
  - blocks CPU thread until all bytes have been copied
  - doesn't start copying until previous CUDA calls complete
- enum cudaMemcpyKind
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice
- Non-blocking copies are also available

IDs and Dimensions

- **Threads:**
  - 3D IDs, unique within a block
- **Blocks:**
  - 2D IDs, unique within a grid
- **Dimensions set at launch**
  - Can be unique for each grid
- **Built-in variables:**
  - threadIdx, blockIdx
  - blockDim, gridDim



Blocks must be independent

- Any possible interleaving of blocks should be valid
  - presumed to run to completion without pre-emption
  - can run in any order
  - can run concurrently OR sequentially

- Blocks may coordinate but not synchronize
  - shared queue pointer: OK
- Independence requirement gives scalability

Questions?

- Demo is provided on canvas!