CUDA C Basics

Supercomputing 2010 Tutorial

Cyril Zeller, NVIDIA Corporation

Outline

- GPU Memory Management
- Code Execution on the GPU
- Coordinating CPU and GPU Execution
- Development Resources

  - See the Programming Guide for the full API
  - See the Getting Started Guide for installation and compilation instructions
  - Both guides are available at http://developer.nvidia.com/object/cuda_downloads.html

GPU Memory Management

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 in code that executes on the GPU will likely crash
    - Dereferencing GPU pointer in code that executes on the CPU will likely crash

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 memory copies are provided

Code Walkthrough 1

- Allocate CPU memory for *n* integers
- Allocate GPU memory for *n* integers
- Initialize GPU memory to 0s
- Copy from GPU to CPU
- Print the values

Code Walkthrough 1

```
#include <stdio.h>
int main() {
    int dimx = 16;
    int num_bytes = dimx * sizeof(int);
    int *d_a = 0, *h_a = 0; // device and host pointers
    …
```

Code Walkthrough 1

```
#include <stdio.h>
int main() {
    int dimx = 16;
    int num_bytes = dimx * sizeof(int);
    int *d_a = 0, *h_a = 0; // device and host pointers
    h_a = (int*)malloc(num_bytes);
    cudaMalloc((void**)&d_a, num_bytes);
    if (0 == h_a || 0 == d_a) {
        printf("couldn't allocate memory\n");
        return 1;
    }
    …
```

Code Walkthrough 1

```
#include <stdio.h>
int main() {
    int dimx = 16;
    int num_bytes = dimx * sizeof(int);
    int *d_a = 0, *h_a = 0; // device and host pointers
    h_a = (int*)malloc(num_bytes);
    cudaMalloc((void**)&d_a, num_bytes);
    if (0 == h_a || 0 == d_a) {
        printf("couldn't allocate memory\n");
        return 1;
    }
    cudaMemset(d_a, 0, num_bytes);
    cudaMemcpy(h_a, d_a, num_bytes, cudaMemcpyDeviceToHost);
    …
```

Code Walkthrough 1

```
#include <stdio.h>
int main() {
    int dimx = 16;
    int num_bytes = dimx * sizeof(int);
    int *d_a = 0, *h_a = 0; // device and host pointers
    h_a = (int*)malloc(num_bytes);
    cudaMalloc((void**)&d_a, num_bytes);
    if (0 == h_a || 0 == d_a) {
        printf("couldn't allocate memory\n");
        return 1;
    }
    cudaMemset(d_a, 0, num_bytes );
    cudaMemcpy(h_a, d_a, num_bytes, cudaMemcpyDeviceToHost);
    for (int i= 0; i < dimx; i++)
        printf("%d\n", h_a[i]);
    free(h_a);
    cudaFree(d_a);
    return 0;
}
```

Code Execution on the GPU

CUDA Programming Model

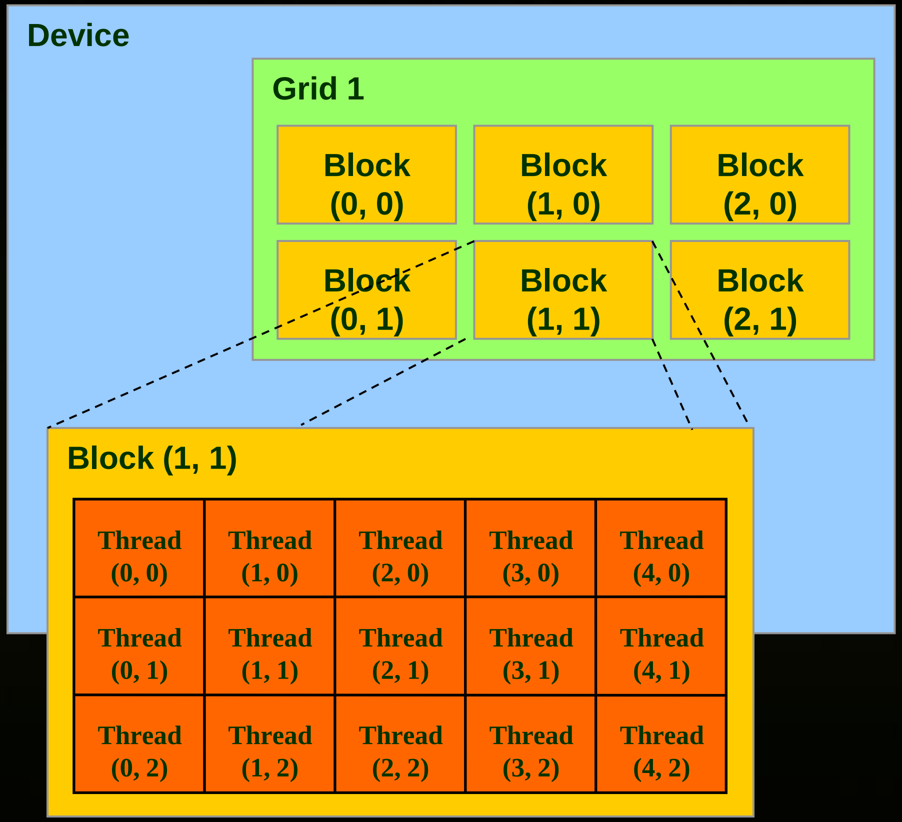
- Parallel code (kernel) is launched and executed on a device by many threads
- Threads are grouped into thread blocks
- Parallel code is written for a thread
  - Each thread is free to execute a unique code path
  - Built-in variables for thread ID and block ID

Thread Hierarchy

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

IDs and Dimensions

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



Code Executed on the GPU

- C function with some restrictions:
  - Can only dereference pointers to GPU memory
  - No static variables
  - No variable number of arguments
  - Some additional restrictions for older GPUs
- The function must be declared with a qualifier:
  - `__global__` : Called from CPU (kernel launch)
    - Cannot be called from GPU
    - Must return `void`
  - `__device__` : Called from other `__global__` and `__device__` functions
    - Cannot be called from CPU
  - `__host__` : can be executed by CPU
    - `__host__` and `__device__` qualifiers can be combined

Code Walkthrough 2

- Build on code walkthrough 1
- Write a kernel to initialize an array of integers
- Copy the result back to CPU
- Print the values

Kernel Code (Executed on the GPU)

```
__global__ void kernel(int* a)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    a[idx] = 7;
}
```

Launching Kernels

- **Launch parameters:**
  - Grid dimensions (up to 2D), `dim3` type
  - Thread block dimensions (up to 3D), `dim3` type
  - Other optional parameters (0 by default):
    - Shared memory allocation (number of bytes per block) for `__shared__` array declared without size
    - Stream ID

```
dim3 grid(16, 16);
dim3 block(16,16);
kernel<<<grid, block, 0, 0>>>(...);
kernel<<<32, 512>>>(...);
```

Code Walkthrough 2

```
#include <stdio.h>
__global__ void kernel(int* a) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    a[idx] = 7;
}
int main() {
    int dimx = 16, num_bytes = dimx*sizeof(int);
    int *d_a = 0, *h_a = 0; // device and host pointers
    h_a = (int*)malloc(num_bytes);
    cudaMalloc((void**)&d_a, num_bytes);
    if (0 == h_a || 0 == d_a) {
        printf("couldn't allocate memory\n");
        return 1;
    }
    cudaMemset(d_a, 0, num_bytes);
    dim3 grid, block;
    block.x = 4;
    grid.x  = dimx / block.x;
    kernel<<<grid, block>>>(d_a);
    cudaMemcpy(h_a, d_a, num_bytes, cudaMemcpyDeviceToHost);
    for(int i = 0; i < dimx; i++)
        printf("%d\n", h_a[i]);
    free(h_a);
    cudaFree(d_a);
    return 0;
}
```

Kernel Variations and Output

```
__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 Walkthrough 3

- Build on code walkthrough 2
- Write a kernel to increment a two-dimensional array of integers
- Copy the result back to CPU
- Print the values

Kernel with 2D Indexing

```
__global__ void kernel(int* a, int dimx, int dimy)
{
    int ix = blockIdx.x * blockDim.x + threadIdx.x;
    int iy = blockIdx.y * blockDim.y + threadIdx.y;
    int idx = iy * dimx + ix;
    a[idx] = a[idx] + 1;
}
```

Code Walkthrough 3

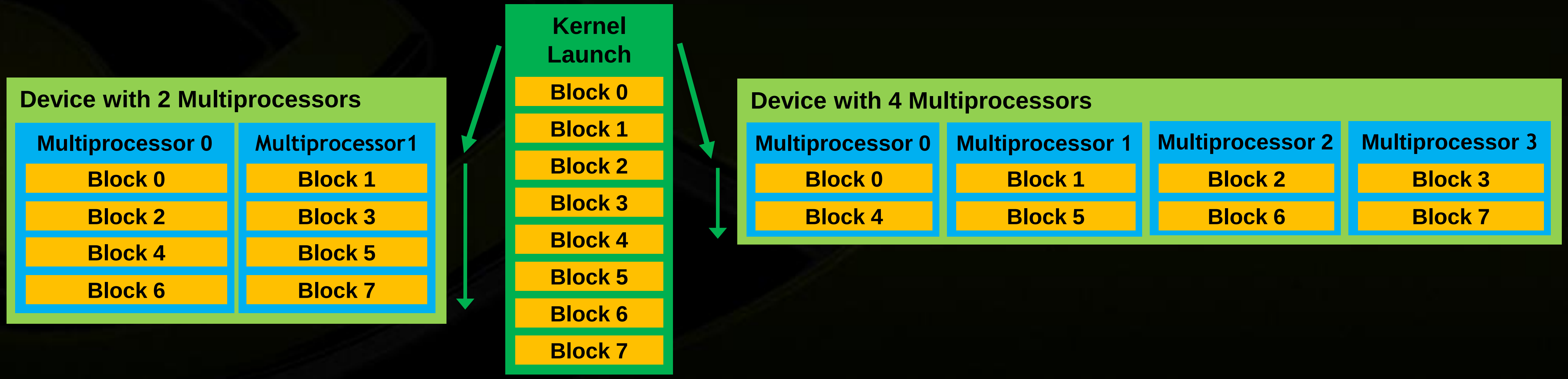
```
int main() {
    int dimx = 16, dimy = 16;
    int num_bytes = dimx * dimy * sizeof(int);
    int *d_a = 0, *h_a = 0; // device and host pointers
    h_a = (int*)malloc(num_bytes);
    cudaMalloc((void**)&d_a, num_bytes);
    if (0 == h_a || 0 == d_a) {
        printf("couldn't allocate memory\n");
        return 1;
    }
    cudaMemset(d_a, 0, num_bytes);
    dim3 grid, block;
    block.x = 4;
    block.y = 4;
    grid.x  = dimx / block.x;
    grid.y  = dimy / block.y;
    kernel<<<grid, block>>>(d_a, dimx, dimy);
    cudaMemcpy(h_a, d_a, num_bytes, cudaMemcpyDeviceToHost);
    for (int row = 0; row < dimy; row++)
        for (int col = 0; col < dimx; col++)
            printf("%d\n", h_a[row * dimx + col]);
    free(h_a);
    cudaFree(d_a);
    return 0;
}
```

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
  - Shared lock: BAD … can easily deadlock

Automatic Scalability

- Block independence requirement gives scalability across any number of multiprocessors



Coordinating CPU and GPU Execution

Synchronizing GPU and CPU

- All kernel launches are asynchronous
  - Control returns to CPU immediately
  - Kernel starts executing after all preceding CUDA calls complete
- `cudaMemcpy()` is synchronous
  - Control returns to CPU once the copy is complete
  - Copy starts once all previous CUDA calls have completed
  - `cudaMemcpyAsync()` is asynchronous
- `cudaThreadSynchronize()`
  - Blocks until all previous CUDA calls complete
- Asynchronous CUDA calls provide ability to:
  - Overlap memory copies and kernel execution
  - Concurrently execute several kernels

CUDA Error Reporting to CPU

- All CUDA calls return error code
  - except kernel launches
  - `cudaError_t` type
- `cudaError_t cudaGetLastError(void)`
  - returns the code for the last error (“no error” has a code)
- `char* cudaGetErrorString(cudaError_t code)`
  - returns a null-terminated character string describing the error

```
printf("%s\n", cudaGetErrorString(cudaGetLastError()));
```

CUDA Event API

- **Events** are inserted (recorded) into CUDA call streams
- Usage scenarios:
  - Measure elapsed time for CUDA calls
  - Query the status of an asynchronous CUDA call
  - Block CPU until CUDA calls prior to the event are completed

```
cudaEvent_t start, stop;
cudaEventCreate(&start), cudaEventCreate(&stop);
cudaEventRecord(start, 0);
kernel<<<grid, block>>>(...);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float time; cudaEventElapsedTime(&time, start, stop);
cudaEventDestroy(start), cudaEventDestroy(stop);
```

Device Management

- CPU can query and select GPU devices
  - `cudaGetDeviceCount(int* count)`
  - `cudaSetDevice(int device)`
  - `cudaGetDevice(int* current_device)`
  - `cudaGetDeviceProperties(cudaDeviceProp* prop, int device)`
  - `cudaChooseDevice(int *device, cudaDeviceProp* prop)`
- Multi-GPU setup:
  - Device 0 is used by default
  - One CPU thread can control one GPU
  - Multiple CPU threads can control the same GPU
    - Calls are serialized by the driver

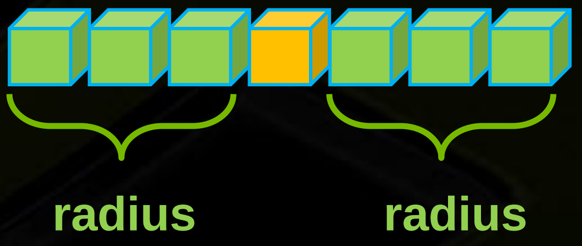
Shared Memory

Shared Memory

- On-chip memory
  - 2 orders of magnitude lower latency than global memory
  - Order of magnitude higher bandwidth than global memory
  - 16 KB or 48 KB per multiprocessor for Fermi architecture (up to 15 multiprocessors)
- Allocated per thread block
- Accessible to any thread in the thread block
  - Not accessible to other thread blocks
- Several uses:
  - Sharing data among threads in a thread block
  - User-managed cache (reducing global memory accesses)

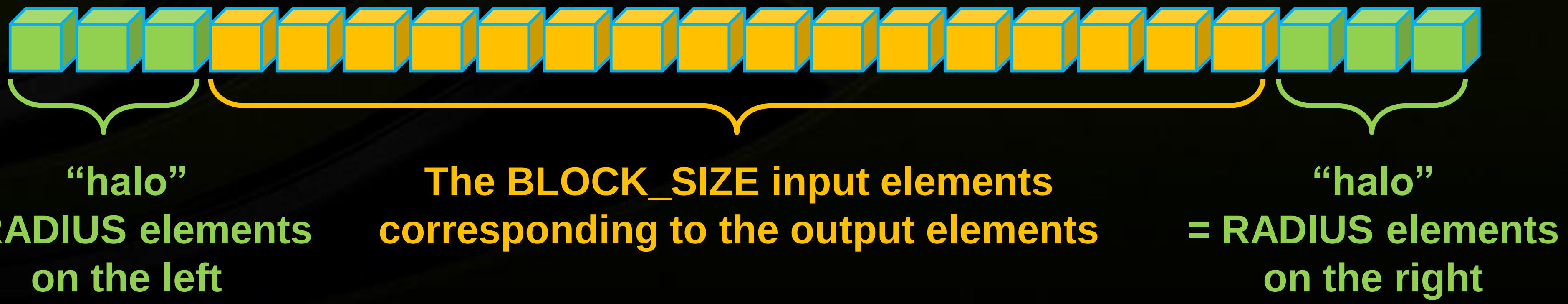
Example of Using Shared Memory

- Applying a 1D stencil to a 1D array of elements:
  - Each output element is the sum of all elements within a radius
- For example, for radius = 3, each output element is the sum of 7 input elements:



Implementation with Shared Memory

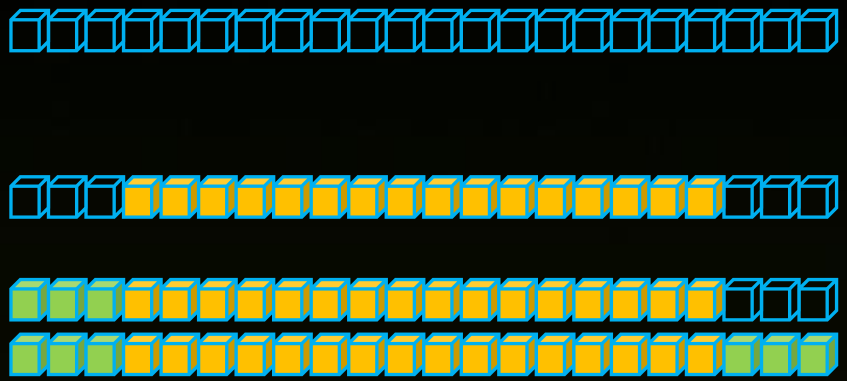
- Each block outputs one element per thread, so a total of BLOCK_SIZE output elements:
  - BLOCK_SIZE = number of threads per block
  - Read (BLOCK_SIZE + 2 * RADIUS) elements from global memory to shared memory
  - Compute BLOCK_SIZE output elements in shared memory
  - Write BLOCK_SIZE output elements to global memory



Kernel Code

```
RADIUS = 3
BLOCK_SIZE = 16
```

```
__global__ void stencil(int* in, int* out) {
  __shared__ int shared[BLOCK_SIZE + 2 * RADIUS];
  int globIdx = blockIdx.x * blockDim.x + threadIdx.x;
  int locIdx  = threadIdx.x + RADIUS;
  shared[locIdx] = in[globIdx];
  if (threadIdx.x < RADIUS) {
    shared[locIdx – RADIUS] = in[globIdx – RADIUS];
    shared[locIdx + BLOCK_DIMX] = in[globIdx + BLOCK_SIZE];
  }
  __syncthreads();
  int value = 0;
  for (offset = - RADIUS; offset <= RADIUS; offset++)
    value += shared[locIdx + offset];
  out[globIdx] = value;
}
```



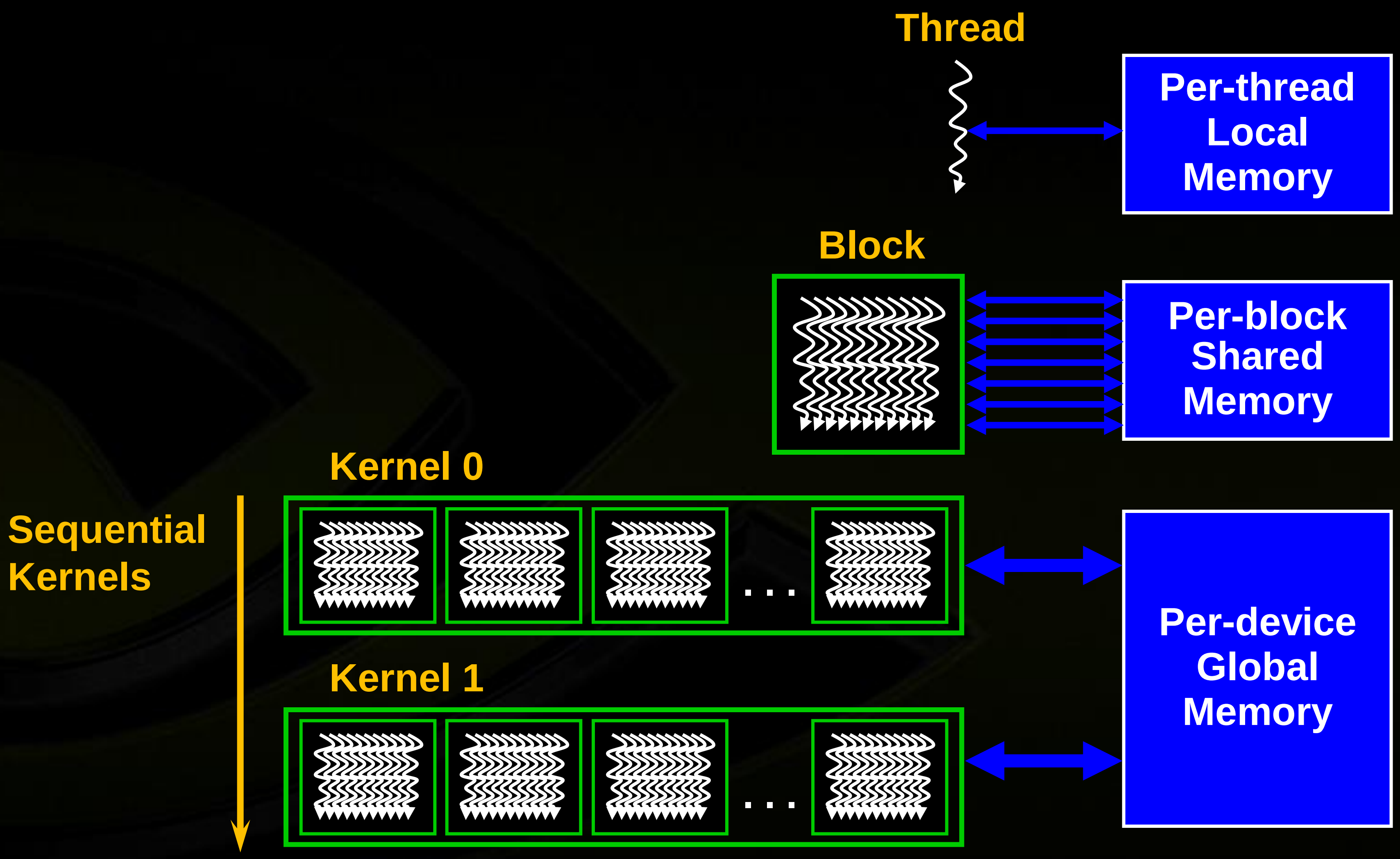
Thread Synchronization Function

- `void __syncthreads();`
- Synchronizes all threads in a thread block
  - Since threads are scheduled at run-time
  - Once all threads have reached this point, execution resumes normally
  - Used to avoid RAW / WAR / WAW hazards when accessing shared memory
- Should be used in conditional code only if the conditional is uniform across the entire thread block

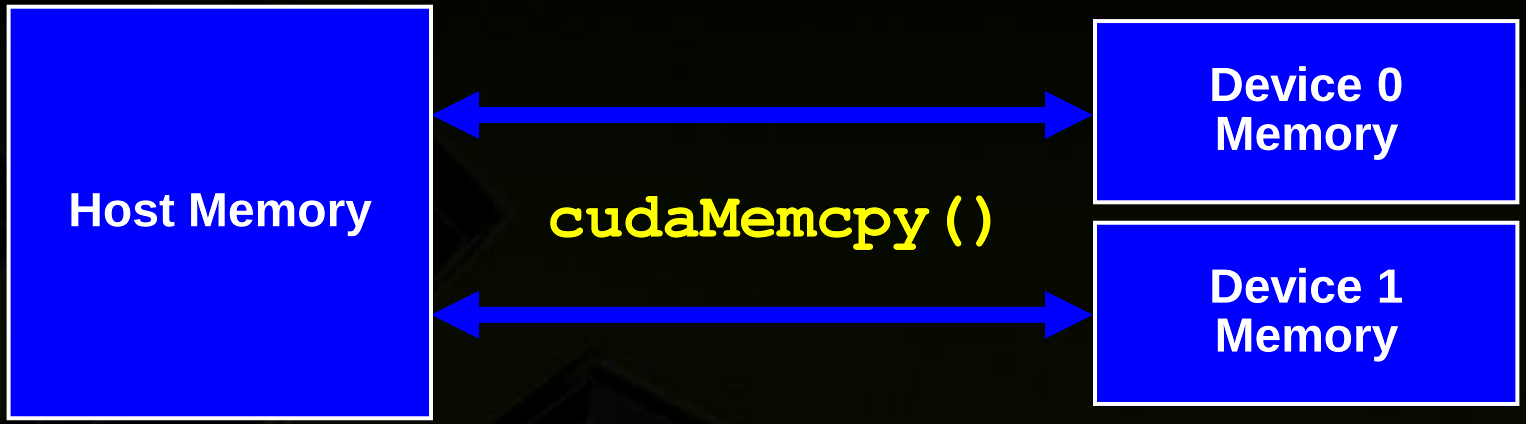
GPU Memory Model Review

- Local storage
  - Each thread has its own local storage
  - Mostly registers (managed by the compiler)
  - Data lifetime = thread lifetime
- Shared memory
  - Each thread block has its own shared memory
    - Accessible only by threads within that block
  - Data lifetime = block lifetime
- Global (device) memory
  - Accessible by all threads as well as host (CPU)
  - Data lifetime = from allocation to deallocation

GPU Memory Model Review



Multi-GPU Memory Model



Questions?