CUDA Programming

15-418/618: Parallel Computer Architecture and Programming

Siddharth Jayashankar

Goals for Today

- Learn to Write Programs in CUDA

1. CUDA Basics : How to write CUDA Programs

2. Advanced Topics : How to write Fast CUDA Programs
    1. Shared Memory
    2. Atomics

What is a GPU?

- GPU = Graphics Processing Unit
- Originally used to display graphics on computer scences
- Now has a new use – AI
- Massively Parallel Processor
- Most famous name in GPUs - Nvidia

What is a CUDA?

- CUDA stood for Compute Unified Device Architecture
- Nvidia’s Programming Language for Programming Nvidia GPUs
- Works with C, C++, Fortran
- Some support for python

CUDA Programming Model

- Three kinds of functions:

Decorator	Run On	Called From
__host__	CPU	CPU
__device__	GPU	GPU
__global__	GPU	CPU

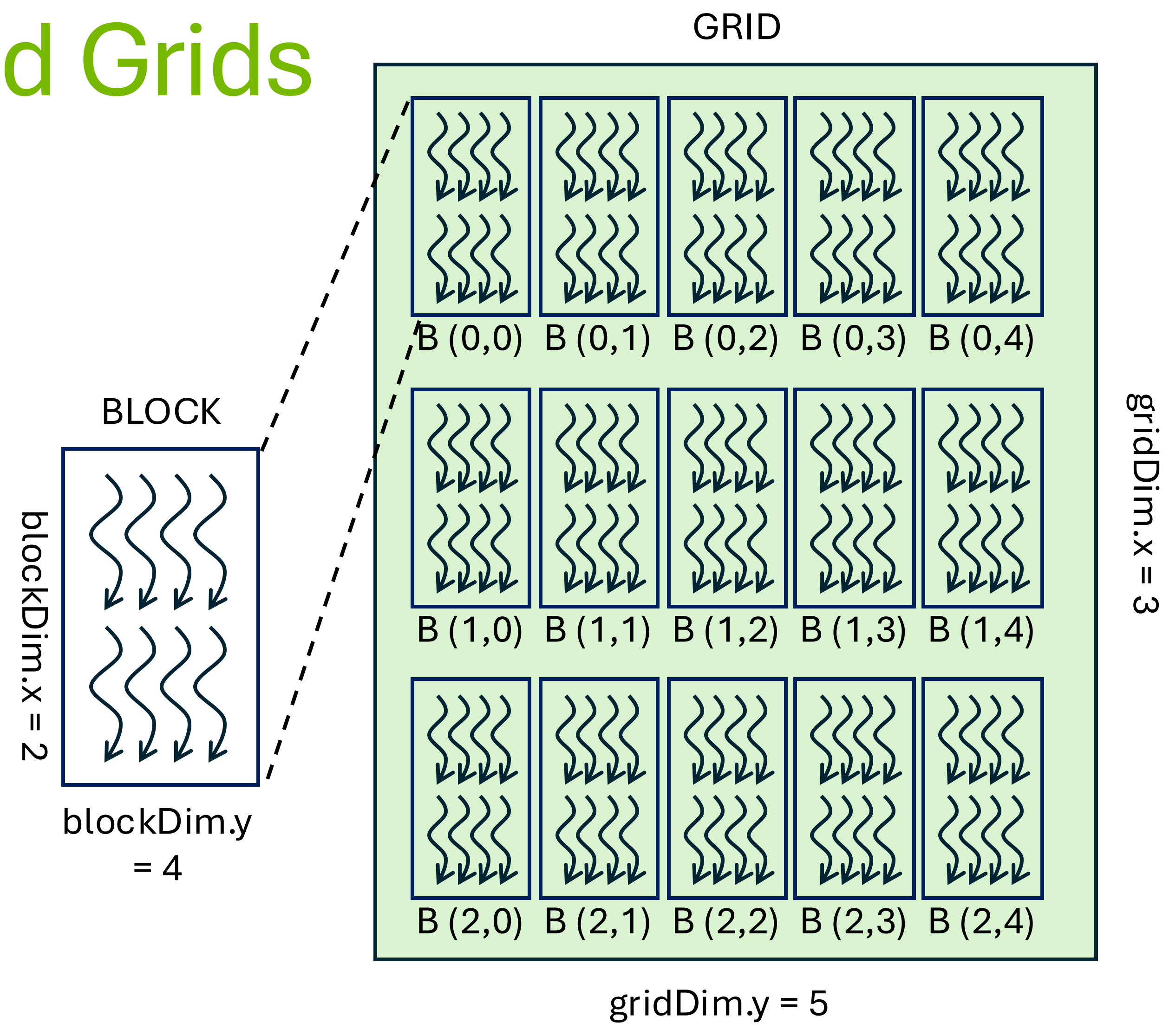
- __global__ functions are called Kernels
- Kernels must have a void return type

CUDA Programming Model – Threads, Blocks and Grids

- GPU massively parallel processor
- Can run Thousands of ”Threads” in Parallel
- CUDA uses a SIMT Model
- SIMT = Single Instruction Multiple Thread
- Thousands of threads. Each thread running the same instruction but on different data

CUDA Programming Model – Threads, Blocks and Grids

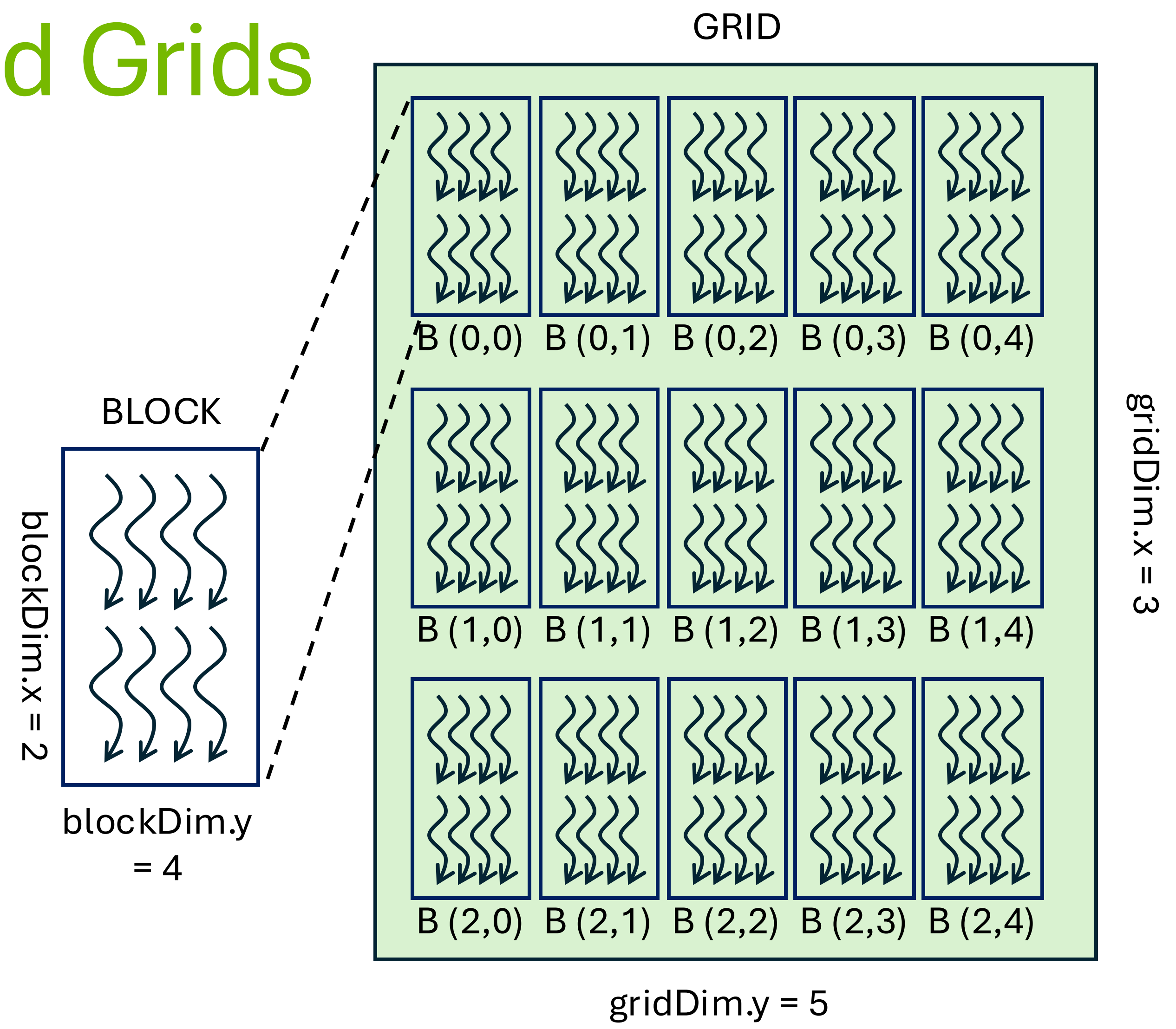
- Every Kernel can map to 1000s of threads
- Threads are Logically Organized into Blocks
- Block are Logically Organized into a Grid
- Grids and Blocks are 3D structures
- Shown here 2D Grids and Blocks



CUDA Programming Model – Threads, Blocks and Grids

- CUDA struct dim3:

  dim3 { uint x,y,z };
- Used to specify grid and block dimensions for kernels
- Example:
  - dim3 grid_dimension(3,5,1);
  - dim3 block_dimension(2,4,1);
- Total Threads:
  - # threads per block = 4x2x1 = 8
  - # blocks in grid = 3x5x1 = 15
  - Total = 8 x 15 = 120



CUDA Example 1

- Write a kernel to add two matrices:
- Open example1/matadd.cu

```
for (int x = 0; x < X_len; x++) {
    for (int y = 0; y < Y_len; y++){
        C[x][y] = A[x][y] + B[x][y];
    }
}
```

- Declare the kernel function

```
__device__ void
matrix_add_kernel(float * C, float * A, float *
B){
     int x = ?
     int y = ?
     C[x][y] = A[x][y] + B[x][y];
}
```

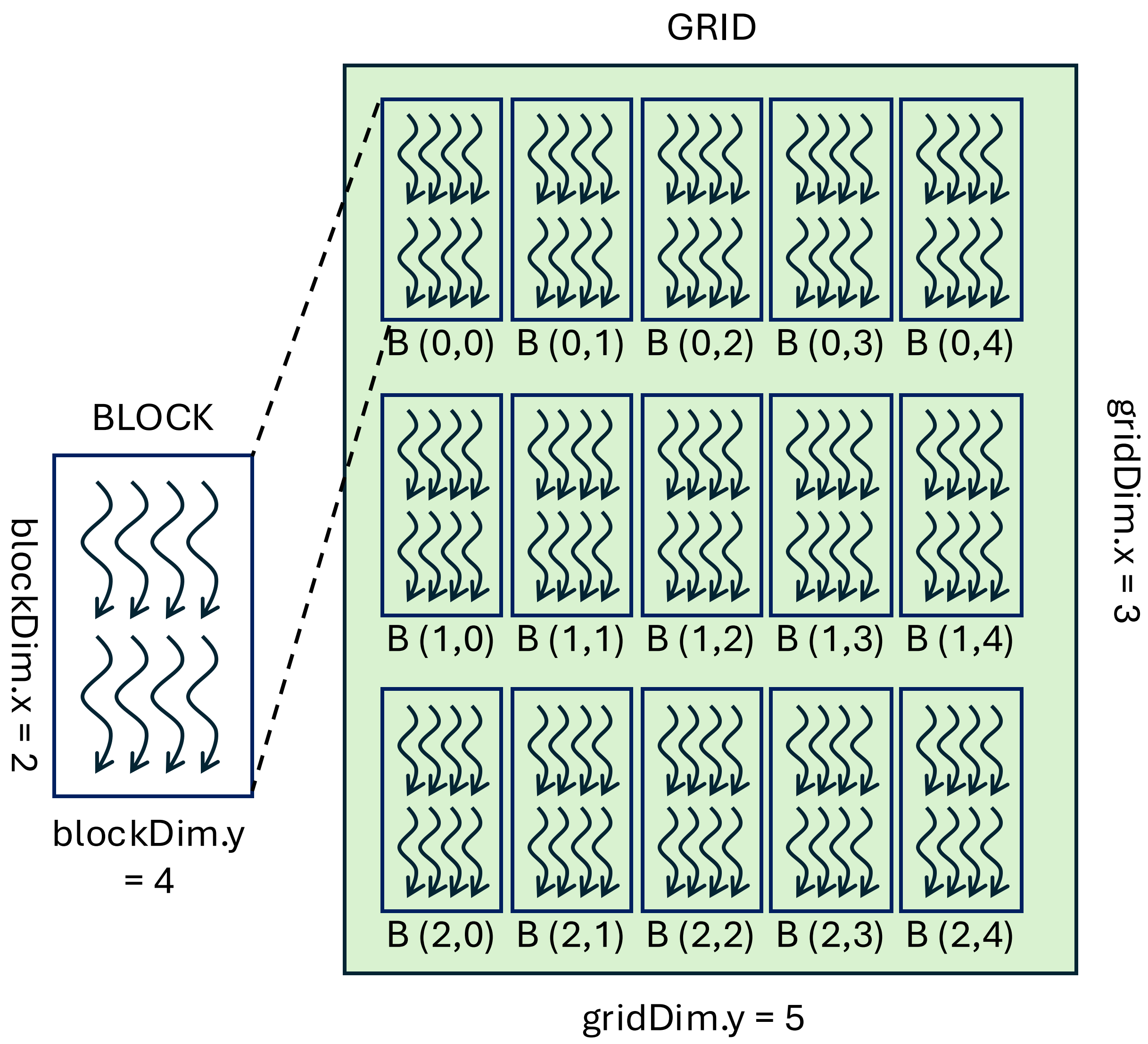
- Need a way to map threads to x and y

CUDA Example 1

- CUDA variables
  - threadIdx: index of thread in Block
  - blockIdx: index of block in Grid
  - blockDim: dimensions of Block
  - gridDim: dimension of Grid
- To get the "overall" index of a thread:

  tid_x = blockIdx.x * blockDim.x + threadIdx.x

  tid_y = blockIdx.y * blockDim.y + threadIdx.y



CUDA Example 1

- Compute the thread Indices

```
__global__ void
matadd_kernel(float ** C, const float ** A, const float ** B) {
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    C[x][y] = A[x][y] + B[x][y];
}
```

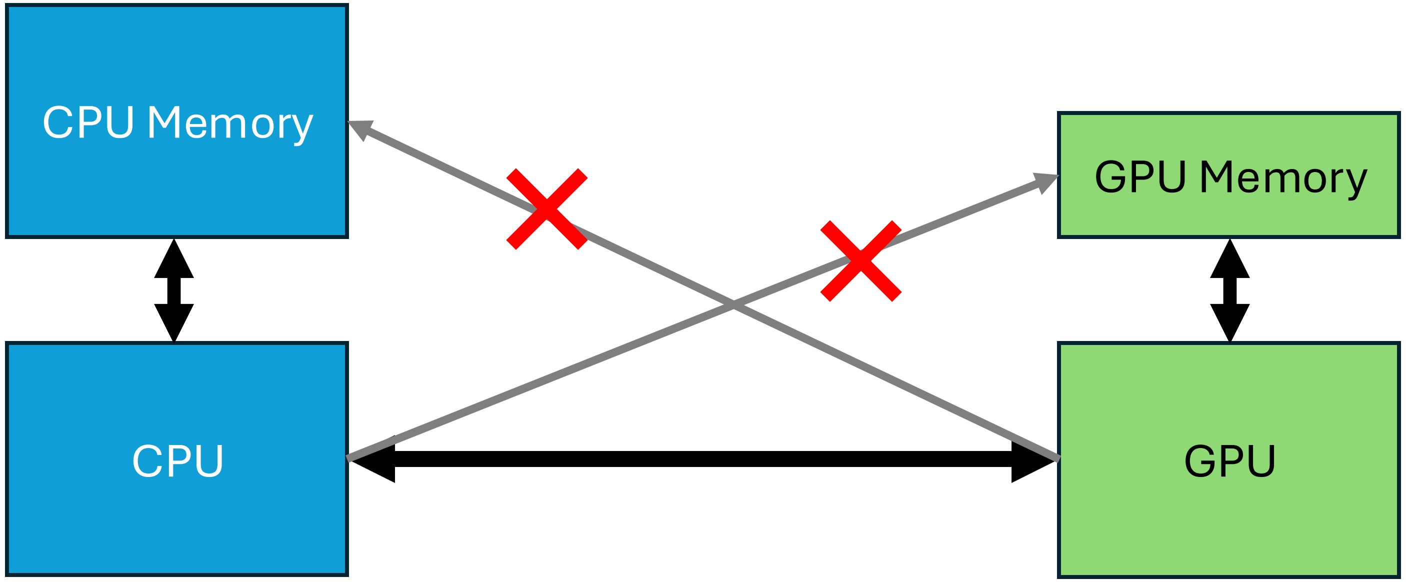
- Determine Block and Grid Size to Launch Kernel

```
__host__ void
matadd(float ** C, const float ** A, const float ** B, const int X_len,
       const int Y_len) {
    dim3 blockSize(16,32);
    dim3 gridSize(X_len/blockSize.x, Y_len/blockSize.y);

    matadd_kernel <<<gridSize,blockSize>>> (C,A,B);
}
```

CUDA Memory Management

- GPU and CPU Memory are not in the same address space
- Programmers need to explicitly move memory between the two
- GPU and CPU cannot access each others' memory



CUDA example 2: Memory Management

- Simple Program to Add two Vectors
- Open example2/main.cu

1. Allocate Memory on CPU

```
float* vec1_host = (float*) malloc(sizeof(float) * length);
```

2. Initialize Inputs
3. Allocate Memory on GPU

```
float * vec1_device = nullptr;
cudaMalloc((void **)&vec1_device, sizeof(float)*length);
gpuErrChk();
```

4. Copy Inputs from CPU to GPU

```
cudaMemcpy(vec1_device, vec1_host, sizeof(float)*length, cudaMemcpyHostToDevice)
gpuErrChk();
```

CUDA example 2: Memory Management

4. Launch Kernel
5. Copy Result from GPU to CPU

```
cudaMemcpy(res_host, res_device, sizeof(float)*length, cudaMemcpyDeviceToHost);
gpuCheckErr();
```

6. Free Memory on GPU

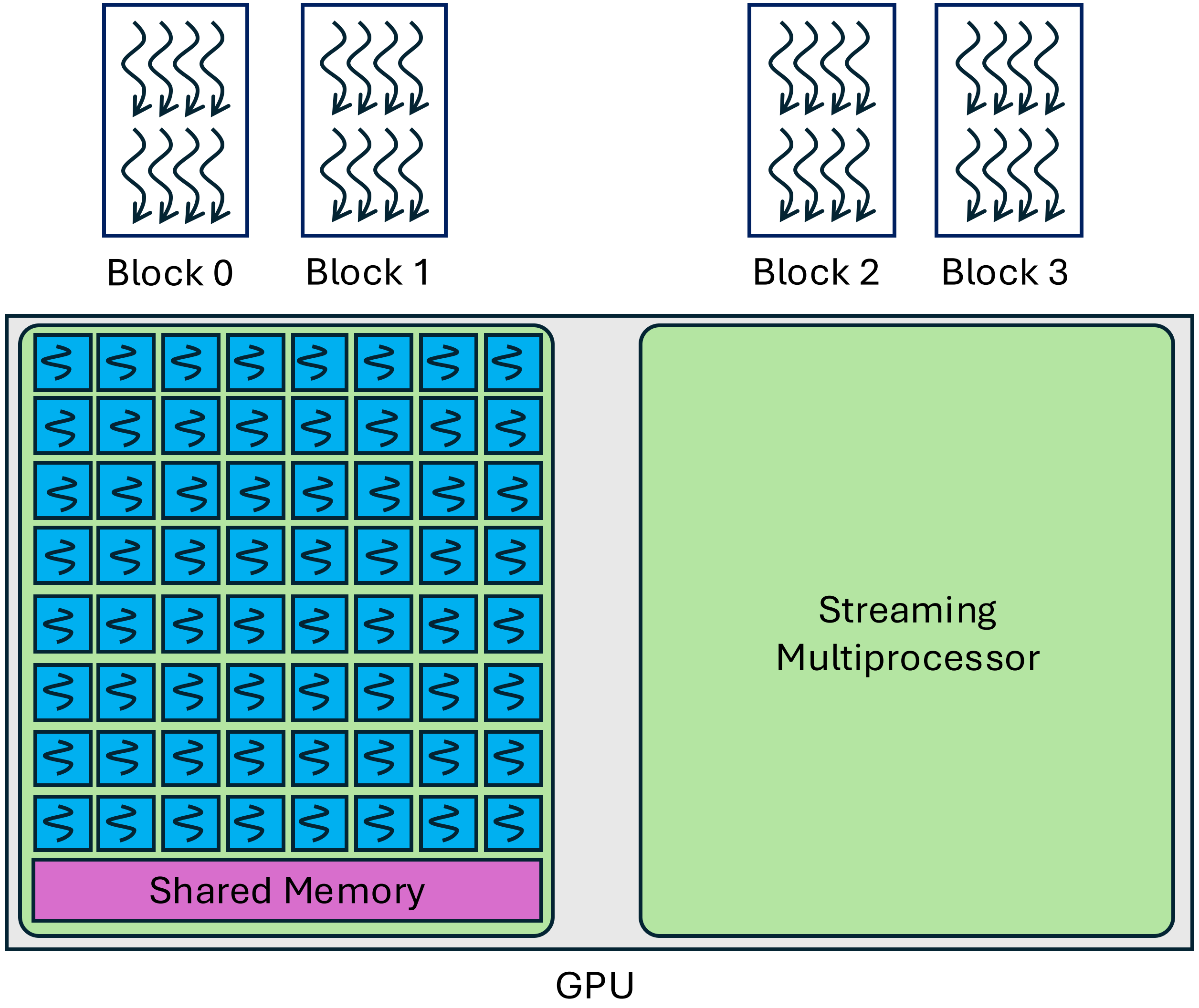
```
cudaFree(vec1_device);
gpuCheckErr();
```

7. Free Memory on CPU

```
free(vec1_host);
```

CUDA and GPU Architecture

- A GPU has 1000s of cores
- A core runs a thread.
- Cores are organized into Streaming Multiprocessors (SM)
- A block of threads is guaranteed to run on a single SM
- Additional Features only in a Block
  - __syncthreads()
  - Shared Memory



CUDA Example 3: Reduction

- Compute the sum of an array
- Challenge: Dependencies across iterations
- Want to reduce dependencies between iterations
- Use a reduction tree

```
Float sum = 0.0f
for (int i=0; i < length; x++) {
    sum += A[i];
}
```

```
for (int d=1; d < length; d <<= 1) {
    for (int i=0; i < length; x+=(d << 1)) {
        A[i] += A[i+d];
    }
}
float sum = A[0];
```

CUDA Example 3: Reduction

- Compute the sum of an array
- Challenge: Dependencies across iterations
- Want to reduce dependencies between iterations
- Use a reduction tree
- Implement as a kernel

```
Float sum = 0.0f
for (int i=0; i < length; x++) {
    sum += A[i];
}
```

```
for (int d=1; d < length; d <<= 1) {
    for (int i=0; i < length; x+=(d << 1)) {
        A[i] += A[i+d];
    }
}
float sum = A[0];
```

```
int i = threadIdx.x + blockIdx.x * blockDim.x;
for (int d=1; d < length; d <<= 1) {
    if(i < length && i % (d << 1) == 0) {
        A[i] += A[i+d];
    }
}
```

CUDA Example 3: Reduction

- This kernel is not correct
- Previous iteration needs to be completed before next iteration begins
- CUDA doesn’t guarantee lockstep execution for all threads
- Need to implement a barrier

```
__global__ void reduce(float * A, int length) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    for (int d=1; d < length; d <<= 1) {
        if(i < length && i % (d << 1) == 0) {
             A[i] += A[i+d];
        }
    }
}
```

CUDA Example 3: Reduction

- Idea1: Use Kernel end as a barrier
- Implement every iteration as a kernel
- Move for loop outside kernel into host code
- Solution is correct

```
__global__ void reduce_1(float * A, int length,
                                   int d) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if(i < length && i % (d << 1) == 0) {
        A[i] += A[i+d];
    }
}
```

```
for (int d=1; d < length; d <<= 1) {
    reduce_1 <<<gridDim,blockDim>>>(A,length,d);
}
```

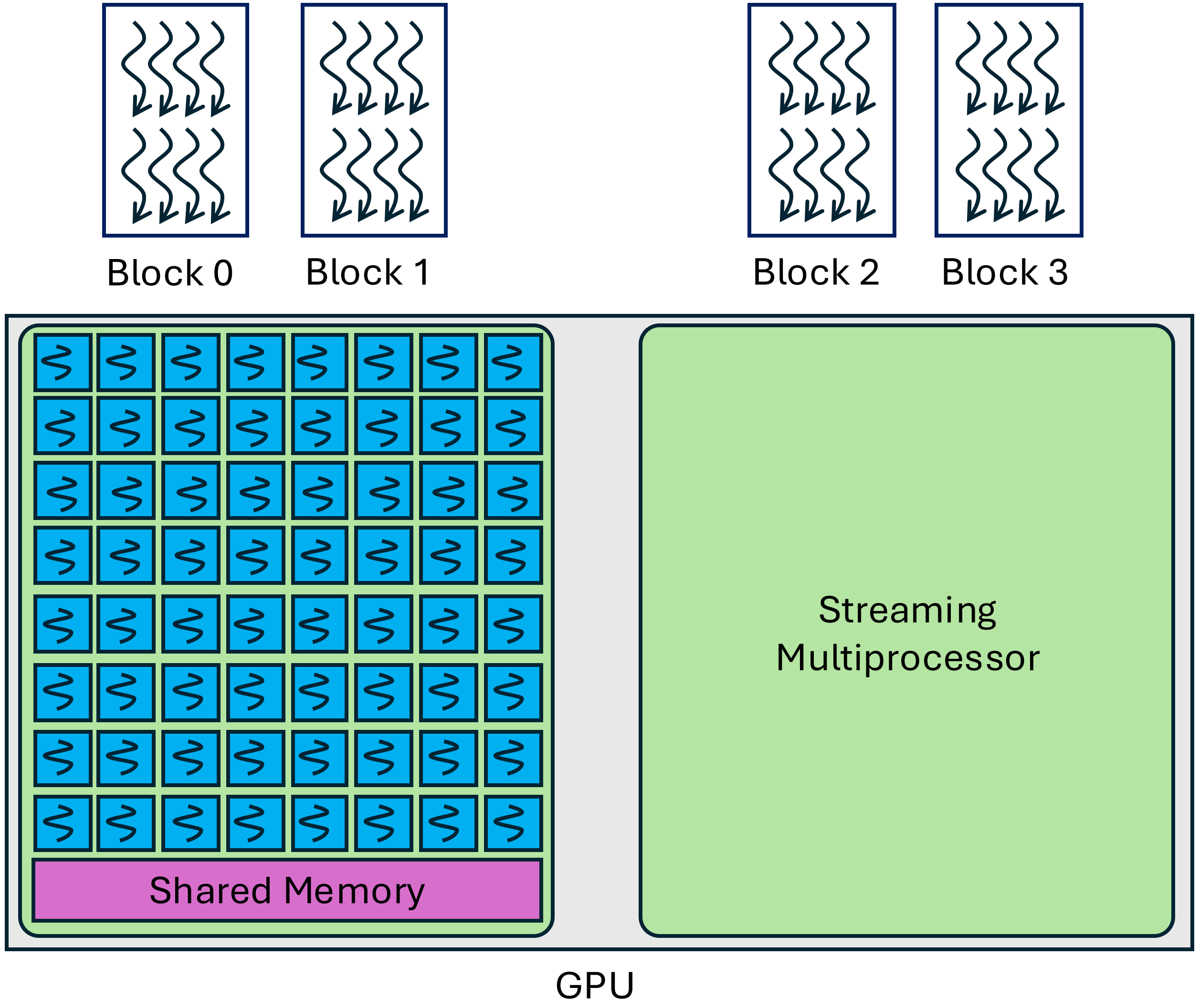
CUDA Example 3: Reduction

- Idea1 is correct but slow
- Launching kernels is an expensive process
- Idea2: Implement barriers in kernel
  - __syncthreads()
- This implementation is not correct

```
__global__ void reduce_2(float * A, int length) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    for (int d=1; d < length; d <<= 1) {
        if(i < length && i % (d << 1) == 0) {
            A[i] += A[i+d];
        }
        __syncthreads();
    }
}
```

CUDA and GPU Architecture

- A block of threads is guaranteed to run on a single SM
- __syncthreads() only works within a block
- Cannot synchronize within a kernel across blocks



CUDA Example 3: Reduction

- Idea2.1 : Implement barriers in kernel
  - __syncthreads()
- __syncthreads() works within a block
- So only launch 1 block
- Good when length is small

```
__global__ void reduce_2_1(float * A, int length) {
    for (int d=1; d < length; d <<= 1) {
        int i = threadIdx.x;
        for(; i < length; i+= blockDim.x) {
            if(i % (d << 1) = 0) {
                A[i] += A[i+d];
            }
        }
        __syncthreads();
    }
}
```

CUDA Example 3: Reduction

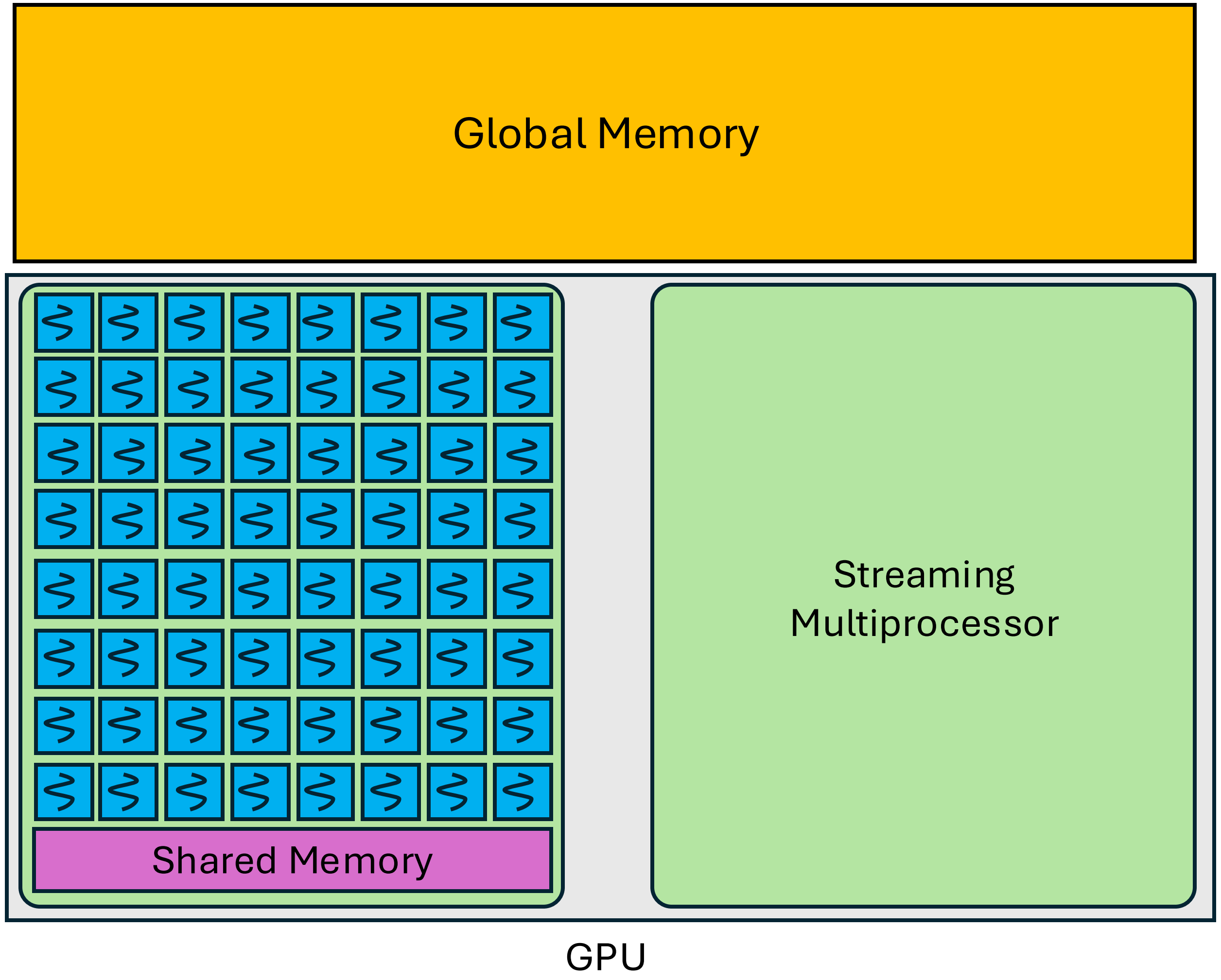
- Idea2.2 :
  - Synchronize within a block using __syncthreads()
  - across blocks using kernels
- Good when length is large

```
__global__ void reduce_2_2(float * A, int length,
                           int depth) {
    int iters = min(blockDim.x,length/depth);
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    for (int d = 1; d < iters; d <<= 1) {
        if(i < length/depth && i % (2*d) == 0) {
            int idx = i*depth;
            A[i*depth] += A[(i+d)*depth];
        }
        __syncthreads();
    }
}
```

```
for (int d=1; d < length; d *= blockDim.x) {
    reduce_2_2 <<<gridDim,blockDim>>>(A,length,d);
}
```

CUDA and GPU Architecture

- Memory Architecture
  - Global Memory - Slow
  - Per Block Shared Memory - Fast



CUDA Example 3: Using Shared Memory

```
__global__ void reduce_2_3(float * A, int length, int depth) {
    int iters = min(blockDim.x,length/depth);
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    external __shared__ float sh_mem[];
    int tid = threadIdx.x;
    if(i < length/depth) { sh_mem[tid] = A[i*depth]; }
    __syncthreads();
    for (int d = 1; d < iters; d <<= 1) {
        if(i < length/(depth) && i % (2*d) == 0) {
            sh_mem[tid] += sh_mem[tid+d];
        }
        __syncthreads();
    }
    if(i < length/depth) { A[i*depth] = sh_mem[tid]; }
}
```

```
int shm_size = sizeof(float) * blockDim;
for (int d=1; d < length; d *= blockDim) {
    reduce_2_2 <<<gridDim,blockDim,shm_size>>>(A,length,d);
}
```

- Idea 2.3
- Use Shared Memory To Speedup Computation
- Remember: Shared Memory is a per Block Construct

Questions?

Thank You