Parallel Programming Language CUDA (C extension)
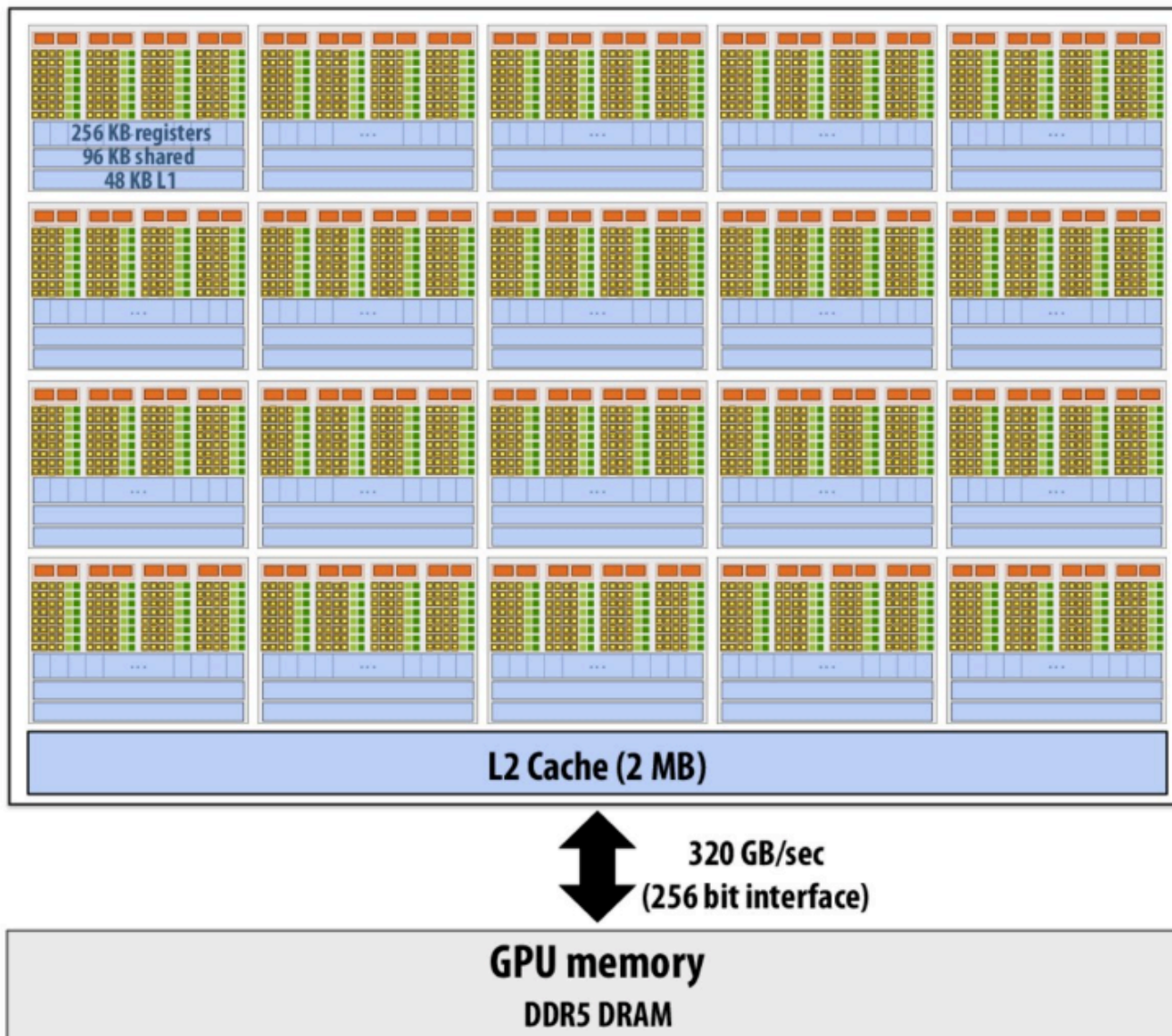
Jiwon Seo
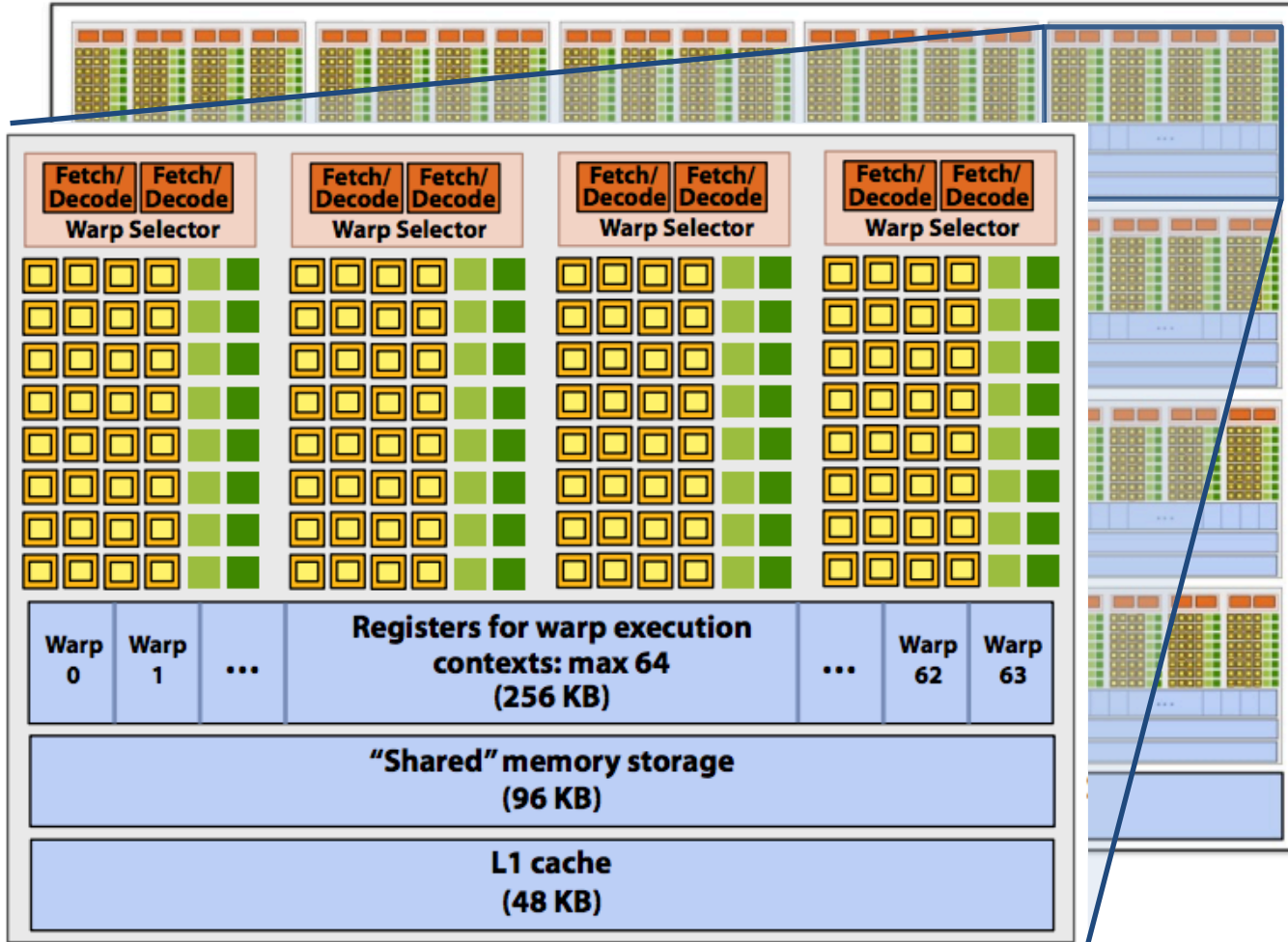
GPU Architecture   GTX 1080

256 KB registers
96 KB shared
48 KB L1

L2 Cache (2 MB)

320 GB/sec
(256 bit interface)

GPU memory
DDR5 DRAM

256 KB registers
96 KB shared
48 KB L1

Stream
Multiprocessor
(SM)

L2 Cache (2 MB)

320 GB/sec
(256 bit interface)

GPU memory
DDR5 DRAM

GPU Architecture GTX 1080

= SIMD functional unit,
control shared across 32 units
(1 MUL-ADD per clock)

= load/store

= SIMD special function unit
(sin, cos, etc.)
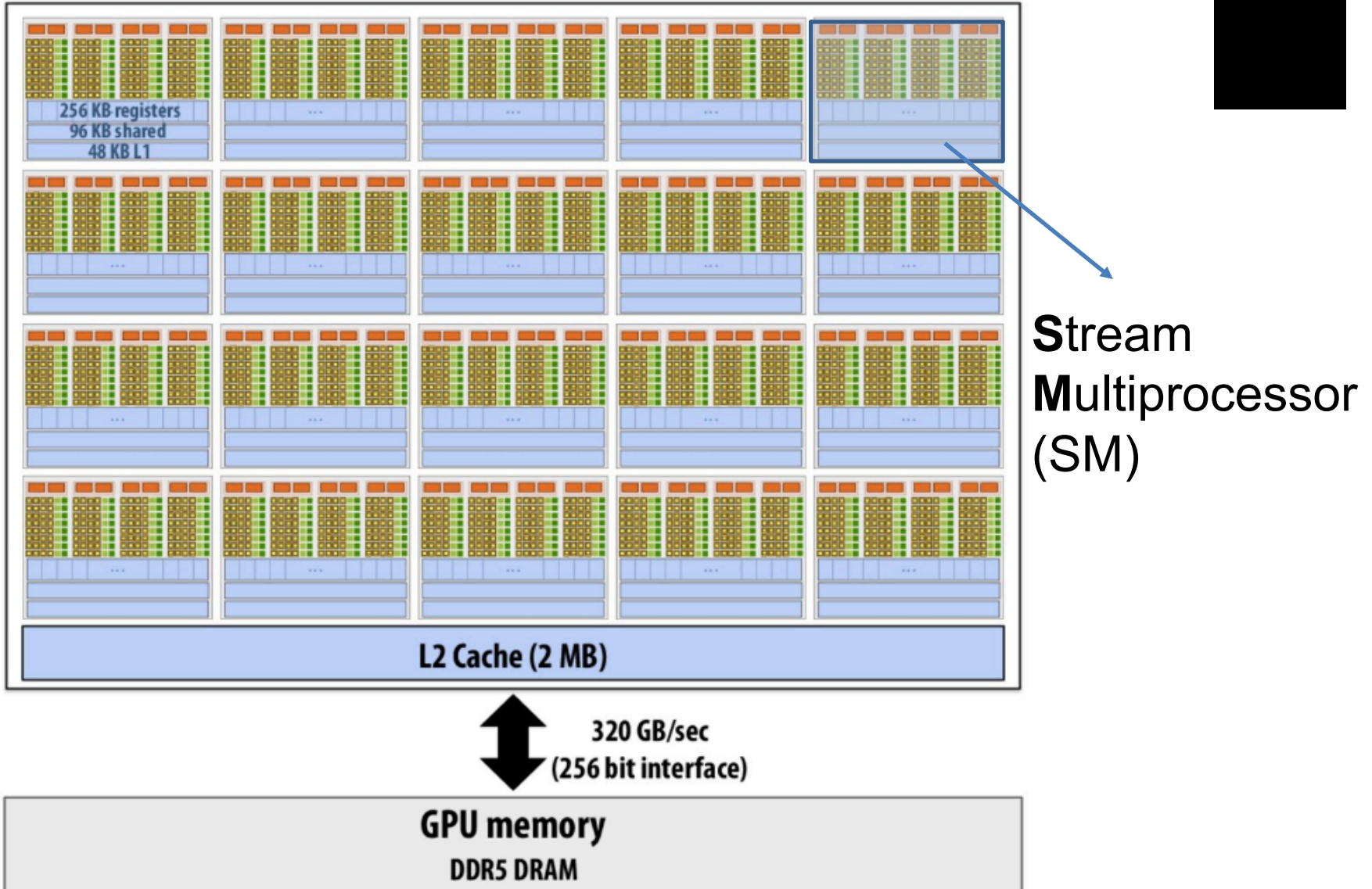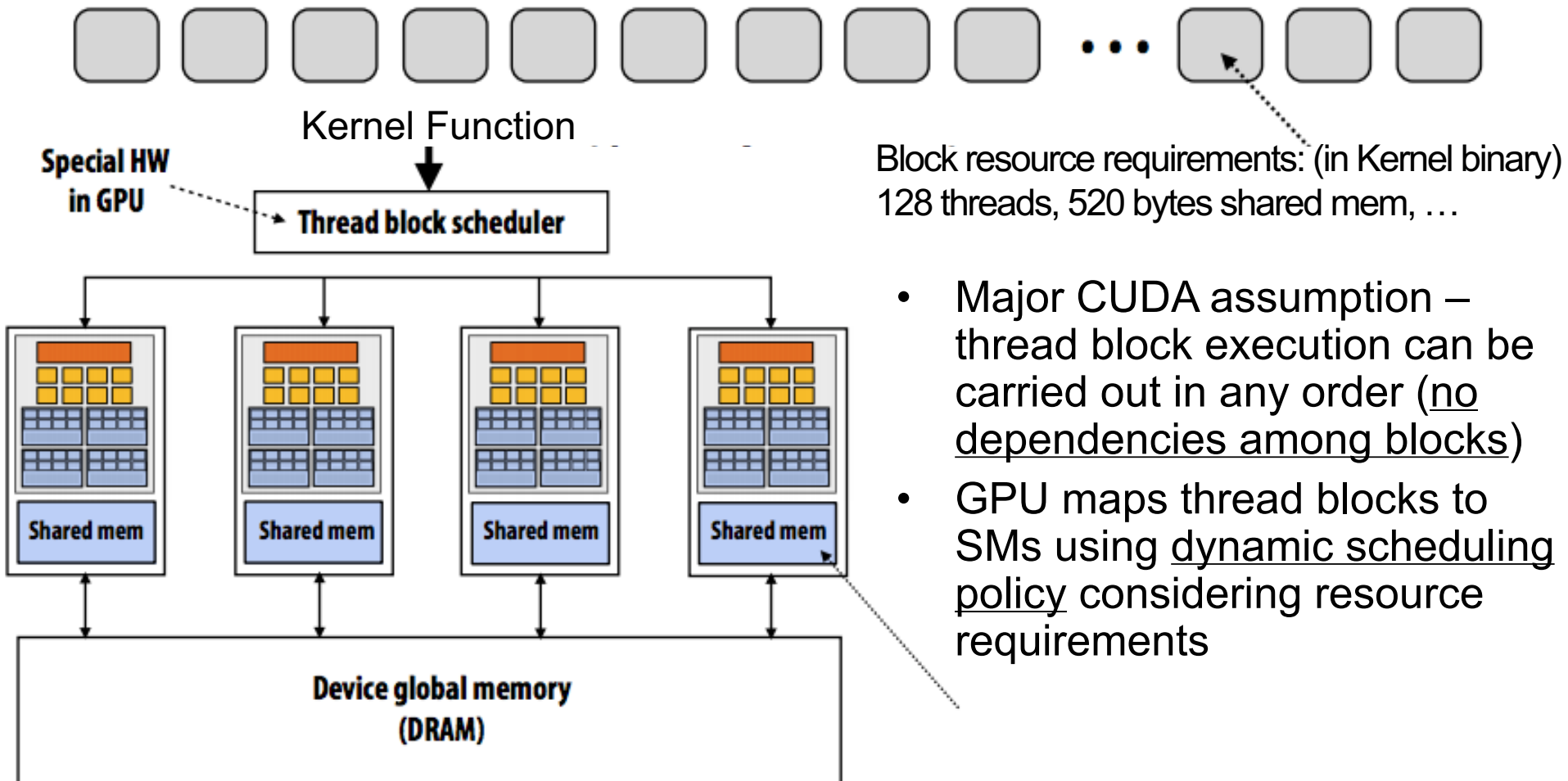
Running a Thread Block on SM
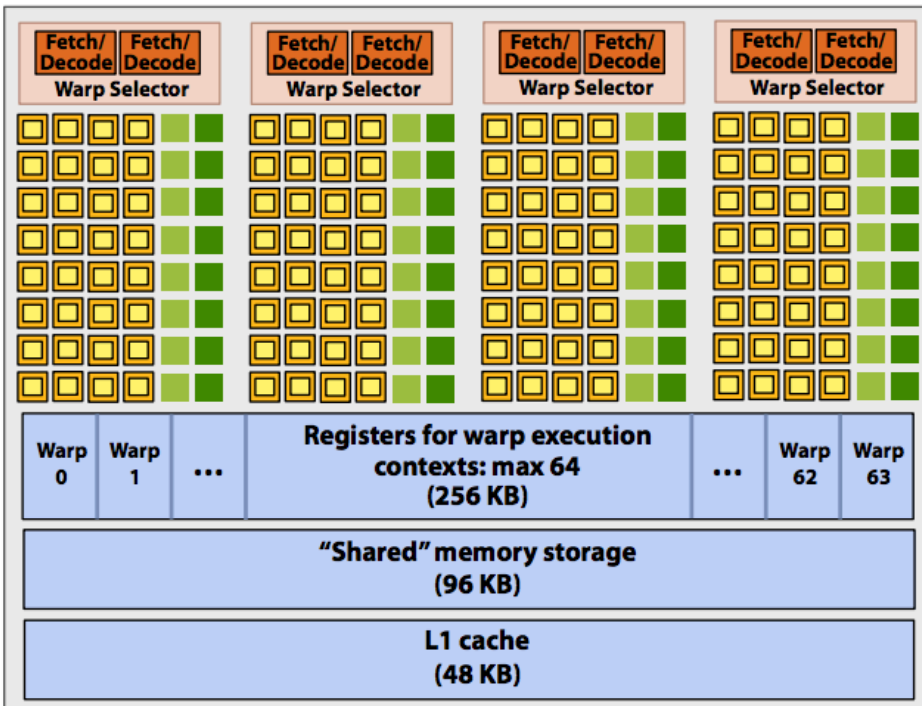
- Groups of 32 threads share instruction stream → Warp

- SM can have up to 64 runnable warps at a time

- SM at each clock:

  - Select up to 4 runnable warps from 64 runnable ones

  - Select up to 2 (runnable) instructions per warp

Scheduling Thread Blocks

Kernel Function

Special HW in GPU

→ **Thread block scheduler**

Block resource requirements: (in Kernel binary) 128 threads, 520 bytes shared mem, …

Shared mem

Shared mem

Shared mem

Shared mem

Device global memory (DRAM)

- Major CUDA assumption – thread block execution can be carried out in any order (<u>no dependencies among blocks</u>)
- GPU maps thread blocks to SMs using <u>dynamic scheduling policy</u> considering resource requirements

Independency of Thread Blocks

- 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

- Independence requirement gives scalability

More on Warp

- A warp is a group of 32 threads
  - Warp is CUDA implementation detail on NVIDIA GPUs
  - Not a PL (programming language) abstraction
- A warp (32 thread group) shares an instruction stream
- In a thread block, threads 0-31 are in a same warp (so do therads 32-63, etc)
- SM can schedule and interleave execution of up to 64 warps (in GTX 1080)
- SM can run multiple thread block concurrently
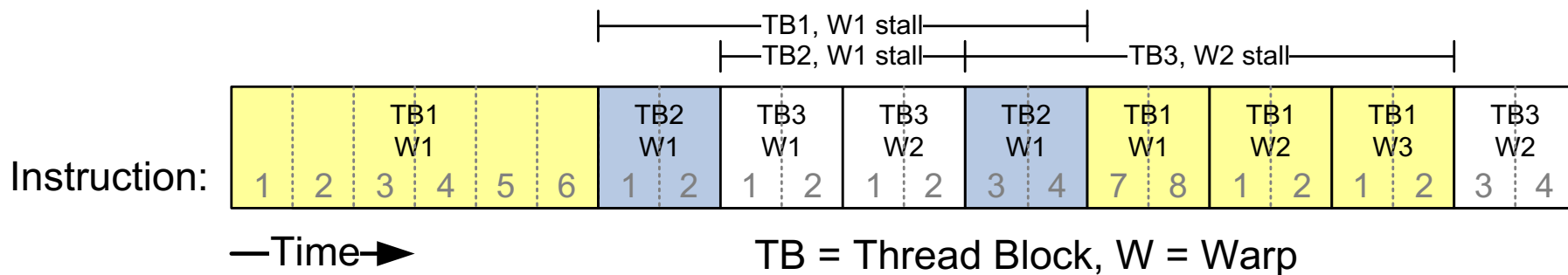
Warps

Multi-core CPU

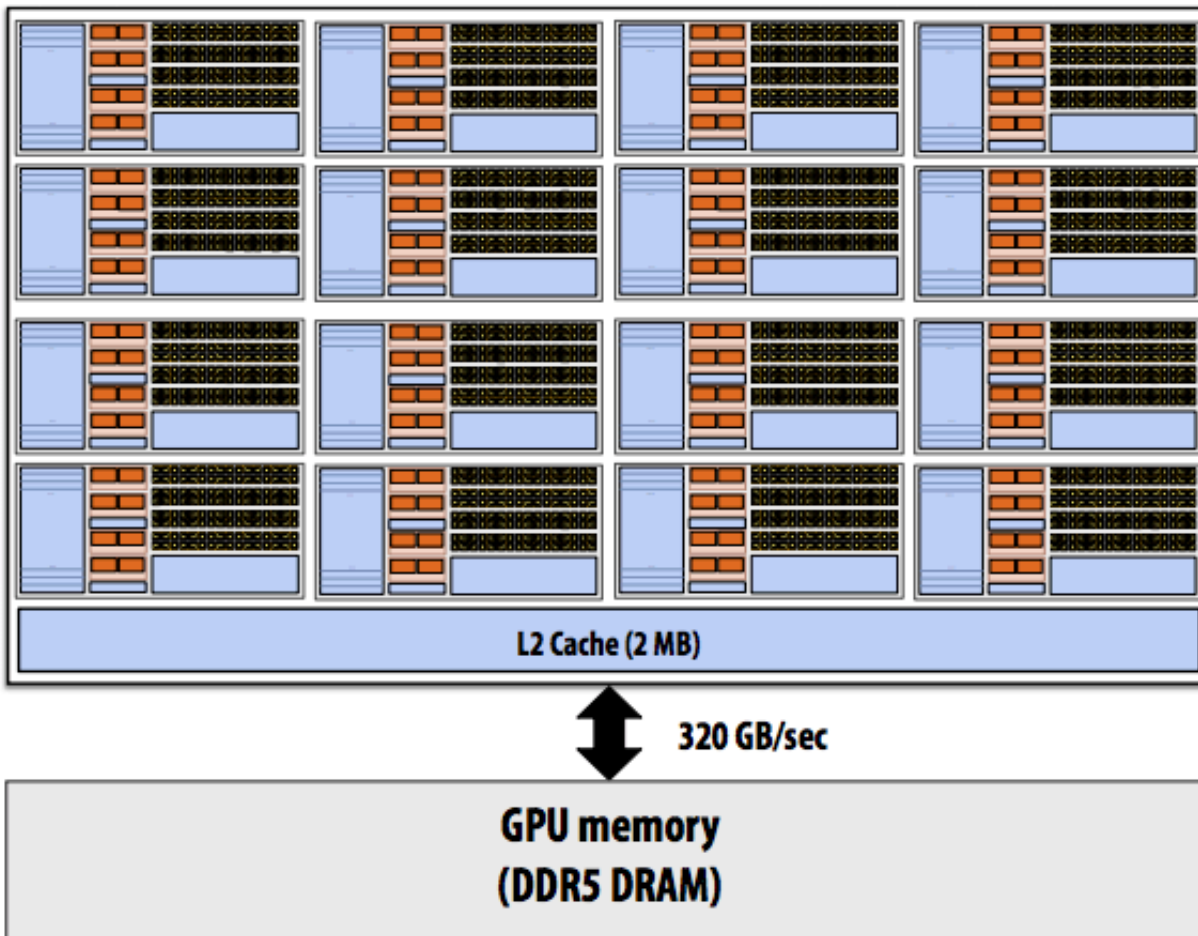| Control	Control	Control	Control	Control	Control
ALU	ALU	ALU	ALU	ALU	ALU
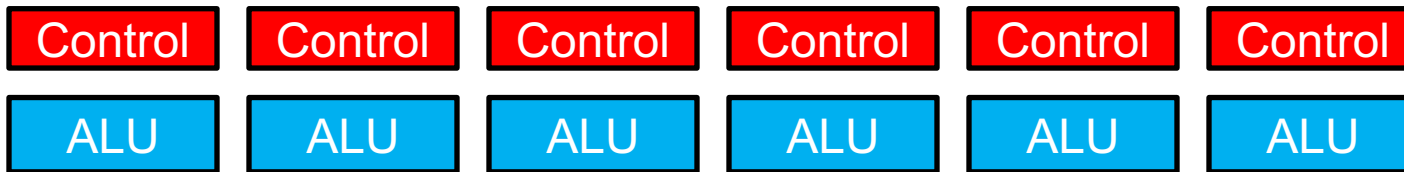
GPU Warp
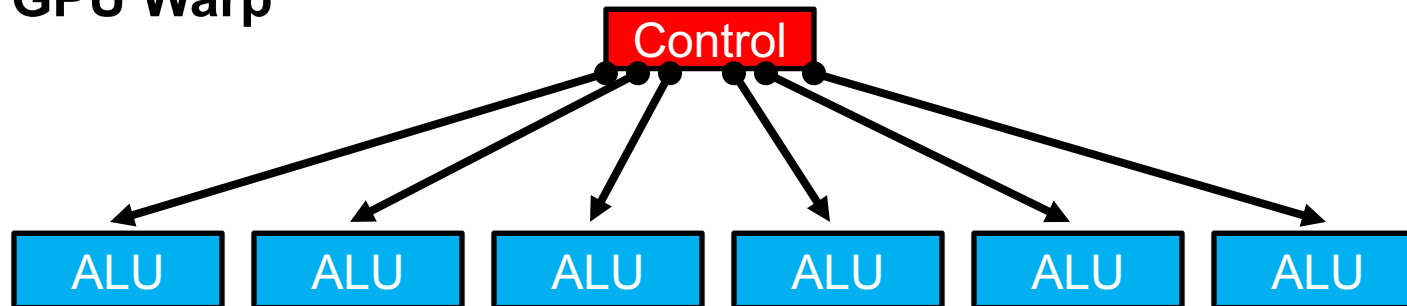
- **A warp = 32 threads launched together**
  - **Usually, execute together as well**

Thread Scheduling Example

- SM implements zero-overhead warp scheduling
  - At any time, a subset of the warps is executed by SM *
  - Warps whose next instruction has its inputs ready are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a warp execute the same instruction

	TB1, W1 stall				
	TB2, W1 stall		TB3, W2 stall		

TB1 W1	TB2 W1	TB3 W1	TB3 W2	TB2 W1	TB1 W1	TB1 W2	TB1 W3	TB3 W2
1 2 3 4 5 6	1 2	1 2	1 2	3 4	7 8	1 2	1 2	3 4

Instruction:

⎯Time➤

TB = Thread Block, W = Warp

More on GTX 1080

1.6 GHz clock

20 SM cores per chip

20 x 128 = 2,560 SIMD mul-add ALUs
= 8.1 TFLOPs

Up to 20 x 64 = 1280 interleaved warps per chip (40,960 CUDA threads/chip)

TDP: 180 watts
(thermal design power)

L2 Cache (2 MB)
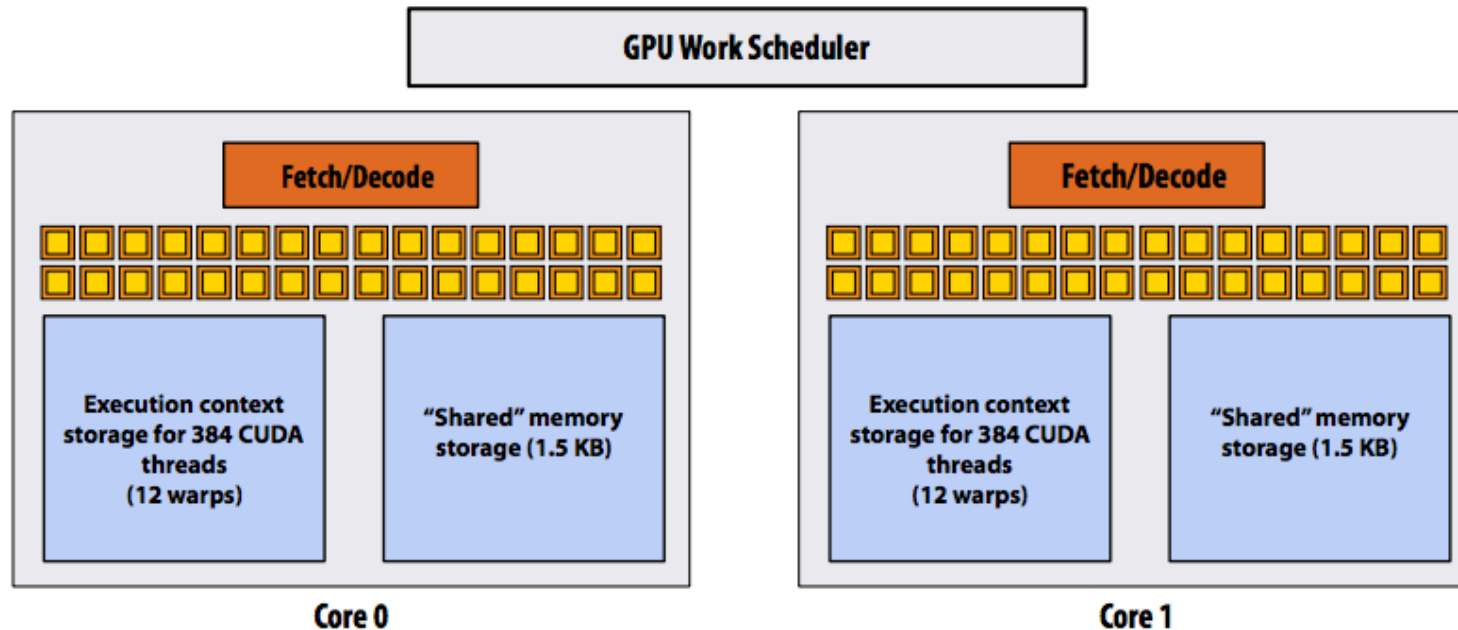
320 GB/sec

GPU memory
(DDR5 DRAM)

Running a CUDA Kernel

Kernel execution requirements:
  Each thread block has 128 threads
  Each thread block requires 520 bytes of shared memory

Assume Kernel runs with large number of thread blocks

Running a CUDA Kernel

Kernel execution requirements:
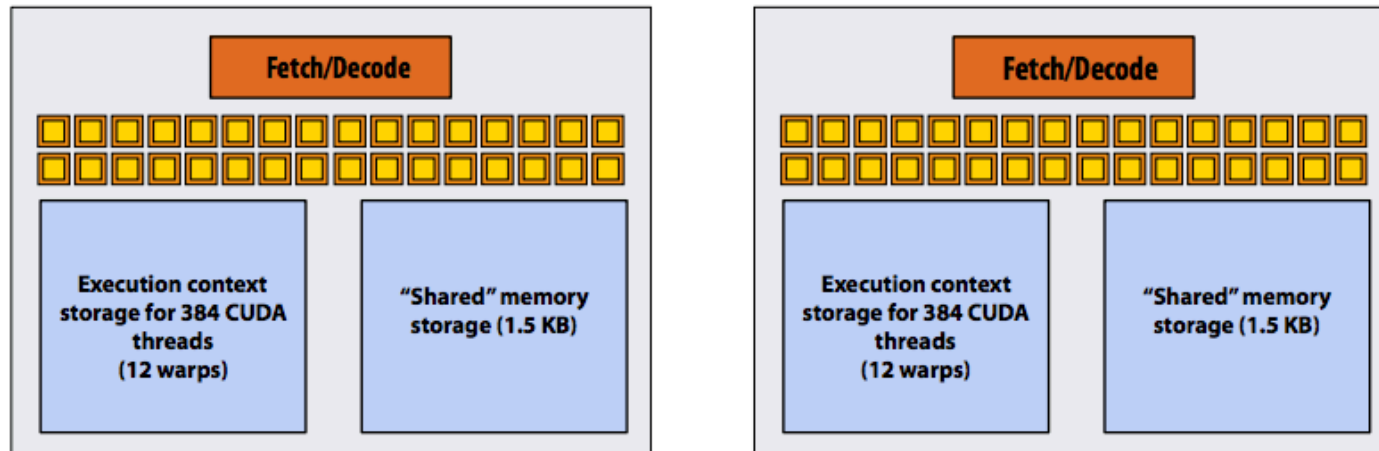>   Each thread block has 128 threads
>   Each thread block requires 520 bytes of shared memory

Step 1: host sends CUDA device (GPU) a command ("execute this kernel")

```
EXECUTE:     kernel
ARGS:        N, input_array, output_array
NUM_BLOCKS:  1000
```

GPU Work Scheduler

Fetch/Decode	Fetch/Decode
Execution context storage for 384 CUDA threads (12 warps)	**"Shared" memory storage (1.5 KB)**
Execution context storage for 384 CUDA threads (12 warps)	**"Shared" memory storage (1.5 KB)**

Running a CUDA Kernel

Kernel execution requirements:
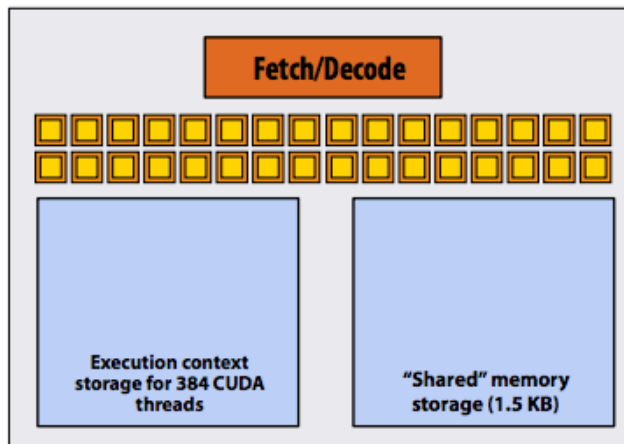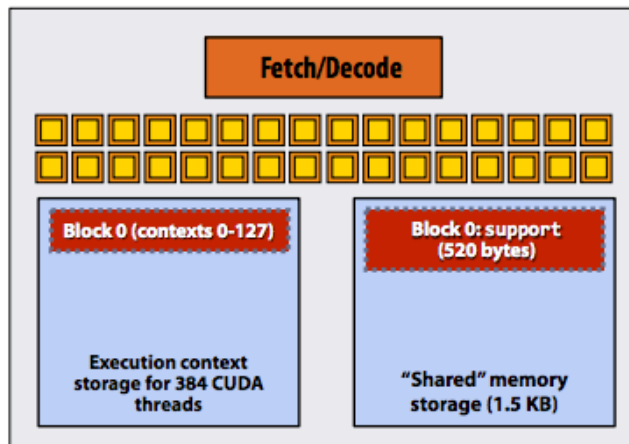   Each thread block has 128 threads
    Each thread block requires 520 bytes of shared memory

Step 2: scheduler maps block 0 to core 0 (reserves execution contexts for 128 threads and 520 bytes of shared storage)

EXECUTE: kernel
ARGS: N, input_array, output_array
NUM_BLOCKS: 1000

GPU Work Scheduler

NEXT = 1
TOTAL = 1000

Fetch/Decode

Block 0 (contexts 0-127)

Block 0: support
(520 bytes)

Execution context
storage for 384 CUDA
threads

"Shared" memory
storage (1.5 KB)

Fetch/Decode

Execution context
storage for 384 CUDA
threads

"Shared" memory
storage (1.5 KB)

Running a CUDA Kernel

Kernel execution requirements:
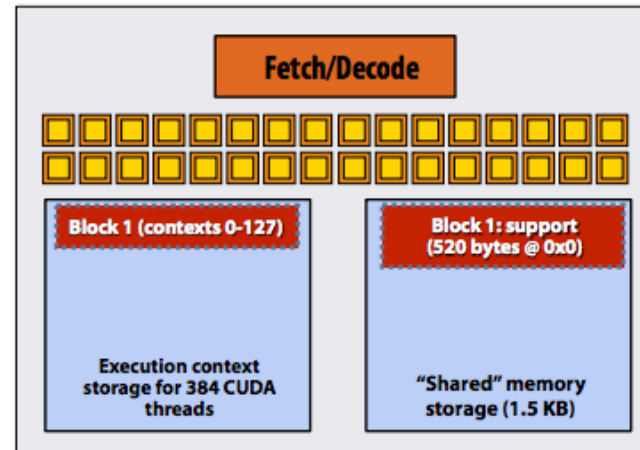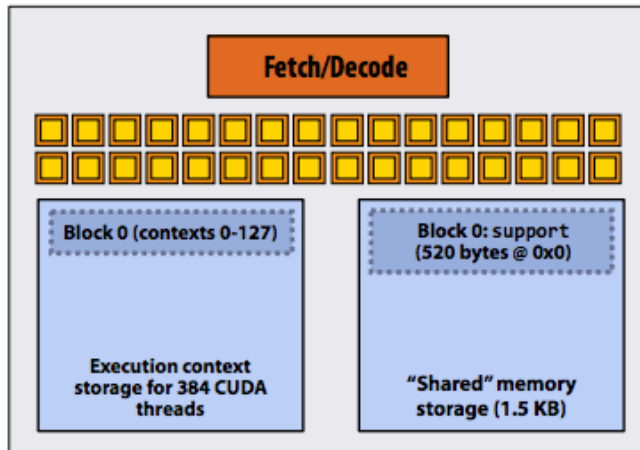 Each thread block has 128 threads
  Each thread block requires 520 bytes of shared memory

Step 3: scheduler continues to map blocks to available execution contexts (interleaved mapping shown)

EXECUTE: kernel
ARGS: N, input_array, output_array
NUM_BLOCKS: 1000

GPU Work Scheduler
NEXT = 2
TOTAL = 1000

Fetch/Decode

Block 0 (contexts 0-127)

Block 0: support
(520 bytes @ 0x0)

Execution context
storage for 384 CUDA
threads

"Shared" memory
storage (1.5 KB)

Fetch/Decode

Block 1 (contexts 0-127)

Block 1: support
(520 bytes @ 0x0)

Execution context
storage for 384 CUDA
threads

"Shared" memory
storage (1.5 KB)

Running a CUDA Kernel

Kernel execution requirements:
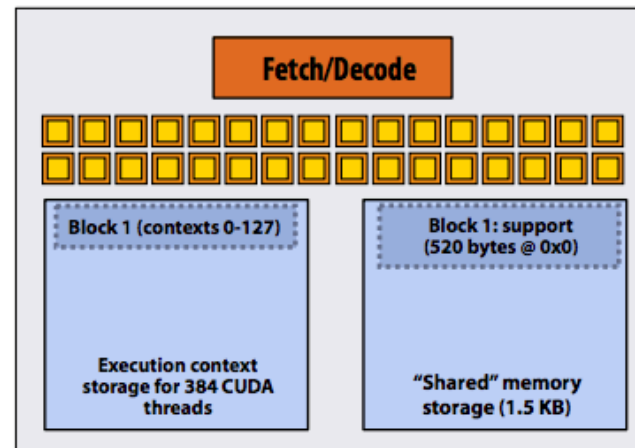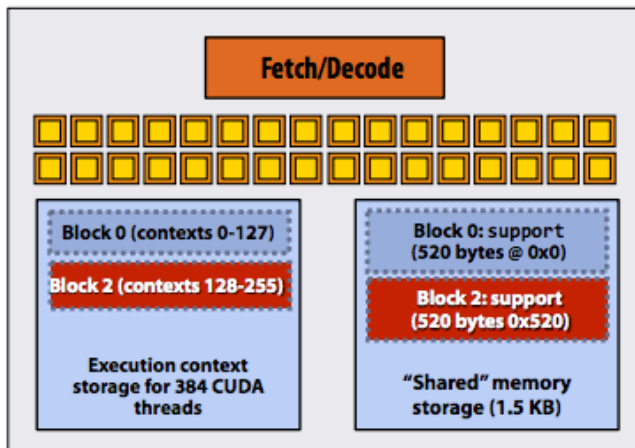   Each thread block has 128 threads
      Each thread block requires 520 bytes of shared memory

Step 3: scheduler continues to map blocks to available execution contexts (interleaved mapping shown)

```
EXECUTE:     kernel
ARGS:        N, input_array, output_array
NUM_BLOCKS: 1000
```

GPU Work Scheduler

```
NEXT = 3
TOTAL = 1000
```

Fetch/Decode

Block 0 (contexts 0-127)

Block 2 (contexts 128-255)

Block 0: support (520 bytes @ 0x0)

Block 2: support (520 bytes 0x520)

Execution context storage for 384 CUDA threads

"Shared" memory storage (1.5 KB)

Fetch/Decode

Block 1 (contexts 0-127)

Block 1: support (520 bytes @ 0x0)

Execution context storage for 384 CUDA threads

"Shared" memory storage (1.5 KB)

Running a CUDA Kernel

Kernel execution requirements:
    Each thread block has 128 threads
    Each thread block requires 520 bytes of shared memory

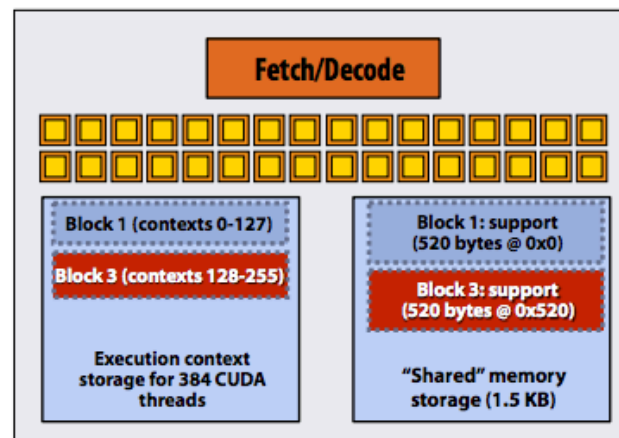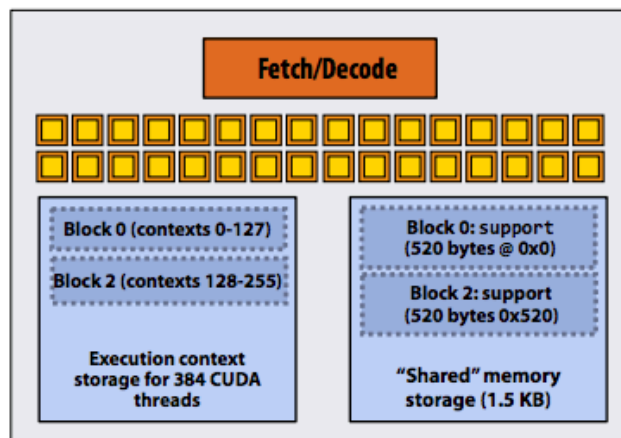**Step 3: scheduler continues to map blocks to available execution contexts (interleaved mapping shown).**
Only two thread blocks fit on a core
(third block won't fit due to insufficient shared storage 3 x 520 bytes > 1.5 KB)

```
EXECUTE:     kernel
ARGS:        N, input_array, output_array
NUM_BLOCKS: 1000
```

| NEXT = 4 | **GPU Work Scheduler** |
| TOTAL = 1000 | |

Fetch/Decode	**Fetch/Decode**
Block 0 (contexts 0-127)	Block 0: support (520 bytes @ 0x0)
Block 2 (contexts 128-255)	Block 2: support (520 bytes 0x520)
Execution context storage for 384 CUDA threads	"Shared" memory storage (1.5 KB)
Block 1 (contexts 0-127)	Block 1: support (520 bytes @ 0x0)
Block 3 (contexts 128-255)	Block 3: support (520 bytes @ 0x520)
Execution context storage for 384 CUDA threads	"Shared" memory storage (1.5 KB)

Running a CUDA Kernel

Kernel execution requirements:
      Each thread block has 128 threads
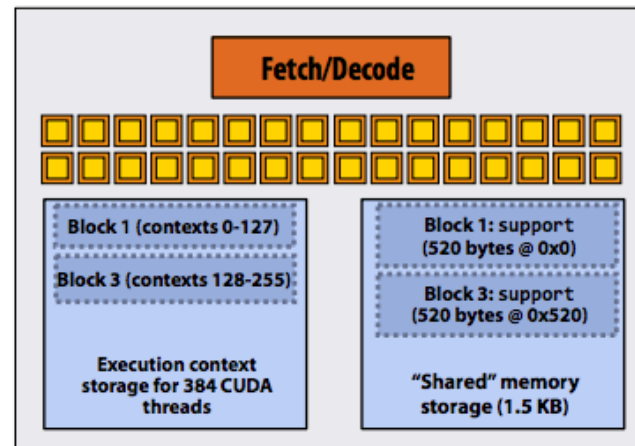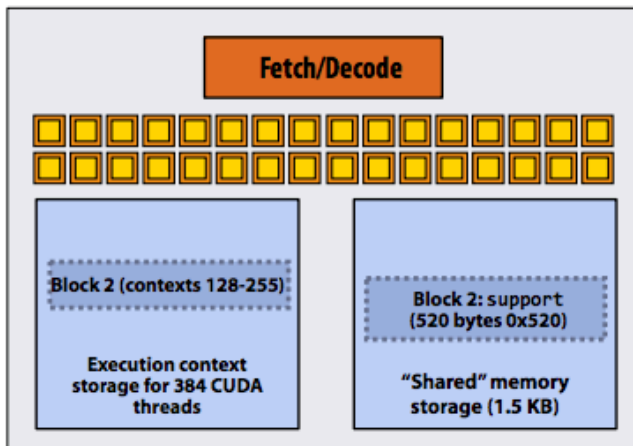   Each thread block requires 520 bytes of shared memory

Step 4: thread block 0 completes on core 0

```
EXECUTE:      kernel
ARGS:         N, input_array, output_array
NUM_BLOCKS: 1000
```

GPU Work Scheduler
NEXT = 4
TOTAL = 1000

Fetch/Decode

Block 2 (contexts 128-255)

Block 2: support
(520 bytes 0x520)

Execution context
storage for 384 CUDA
threads

"Shared" memory
storage (1.5 KB)

Fetch/Decode

Block 1 (contexts 0-127)

Block 3 (contexts 128-255)

Block 1: support
(520 bytes @ 0x0)

Block 3: support
(520 bytes @ 0x520)

Execution context
storage for 384 CUDA
threads

"Shared" memory
storage (1.5 KB)

Running a CUDA Kernel

Kernel execution requirements:
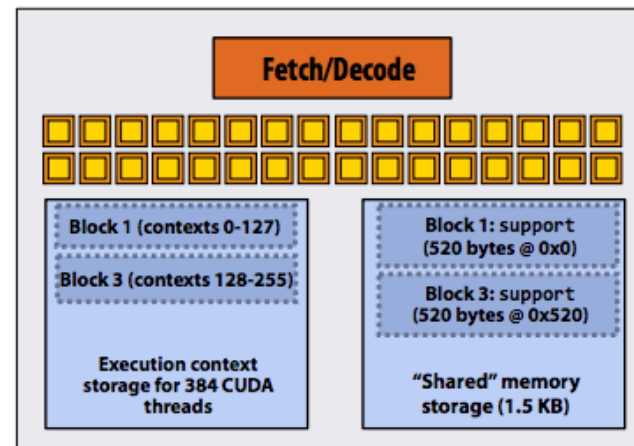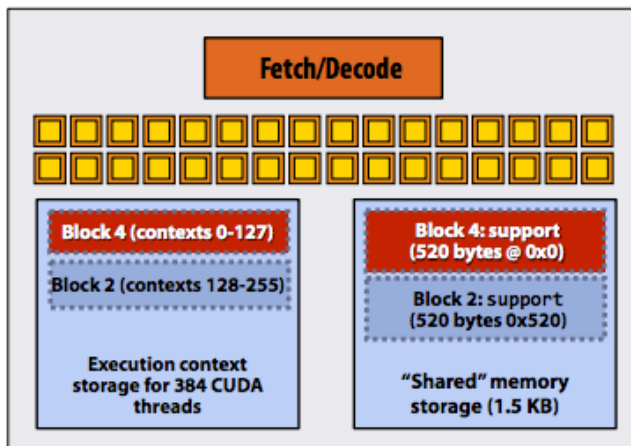   Each thread block has 128 threads
    Each thread block requires 520 bytes of shared memory

Step 5: block 4 is scheduled on core 0 (mapped to execution contexts 0-127)

```
EXECUTE:      kernel
ARGS:         N, input_array, output_array
NUM_BLOCKS: 1000
```

```
NEXT = 5        GPU Work Scheduler
TOTAL = 1000
```

Fetch/Decode

Block 4 (contexts 0-127)

Block 2 (contexts 128-255)

Execution context storage for 384 CUDA threads

Block 4: support (520 bytes @ 0x0)

Block 2: support (520 bytes 0x520)

"Shared" memory storage (1.5 KB)

Fetch/Decode

Block 1 (contexts 0-127)

Block 3 (contexts 128-255)

Execution context storage for 384 CUDA threads

Block 1: support (520 bytes @ 0x0)

Block 3: support (520 bytes @ 0x520)

"Shared" memory storage (1.5 KB)

Running a CUDA Kernel

Kernel execution requirements:
    Each thread block has 128 threads
    Each thread block requires 520 bytes of shared memory

Running a CUDA Kernel

Kernel execution requirements:
      Each thread block has 128 threads
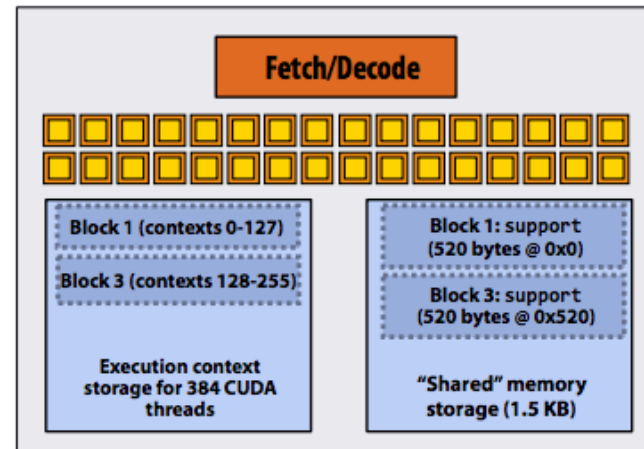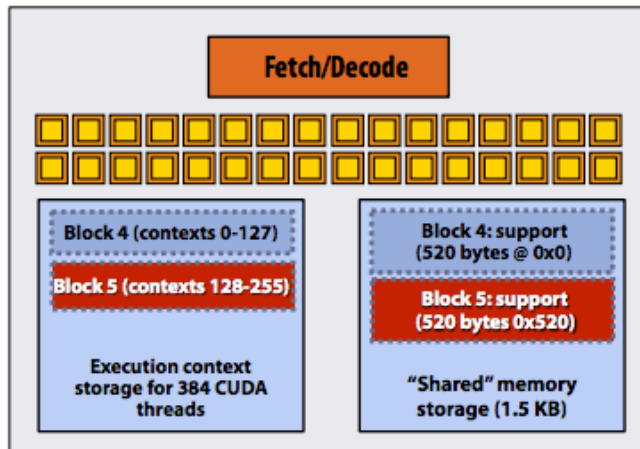   Each thread block requires 520 bytes of shared memory

Step 7: thread block 5 is scheduled on core 0 (mapped to execution contexts 128-255)

```
EXECUTE:      kernel
ARGS:         N, input_array, output_array
NUM_BLOCKS: 1000
```
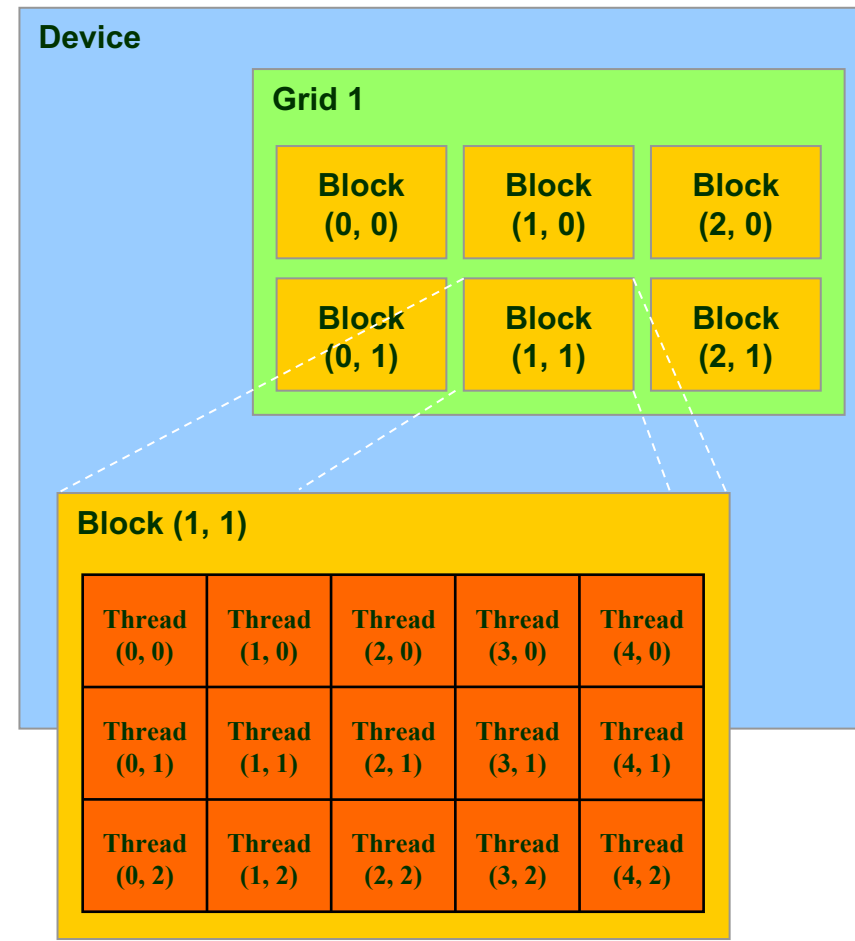
GPU Work Scheduler

```
NEXT = 6
TOTAL = 1000
```

Fetch/Decode

Block 4 (contexts 0-127)

Block 5 (contexts 128-255)

Block 4: support
(520 bytes @ 0x0)

Block 5: support
(520 bytes 0x520)

Execution context
storage for 384 CUDA
threads

"Shared" memory
storage (1.5 KB)

Fetch/Decode

Block 1 (contexts 0-127)

Block 3 (contexts 128-255)

Block 1: support
(520 bytes @ 0x0)

Block 3: support
(520 bytes @ 0x520)

Execution context
storage for 384 CUDA
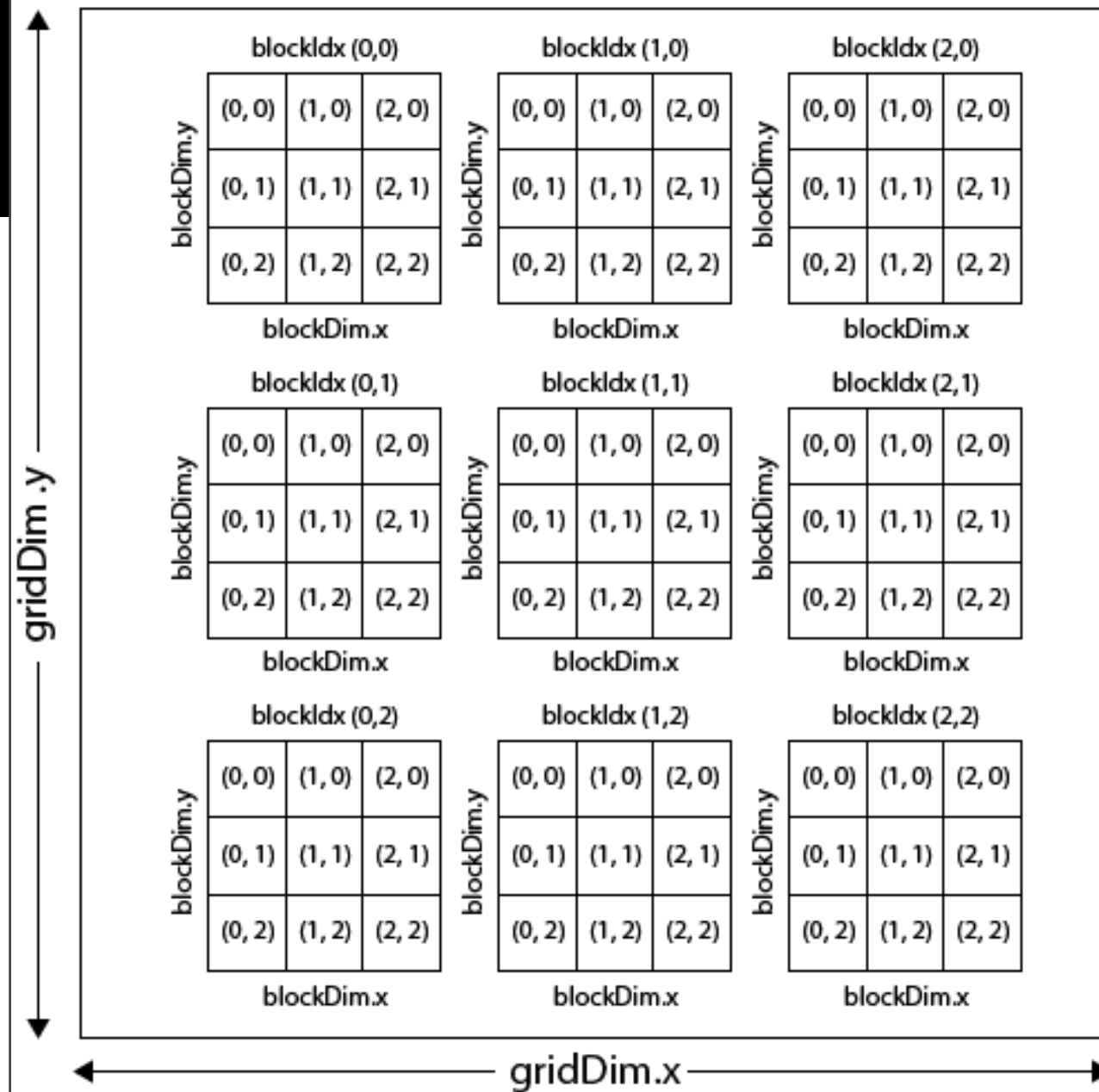threads

"Shared" memory
storage (1.5 KB)

Grid, Blocks, Threads

GPU abstraction: *Grid, Block, Threads*

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

Device

Grid 1

| Block (0, 0) | Block (1, 0) | Block (2, 0) |
| Block (0, 1) | Block (1, 1) | Block (2, 1) |

Block (1, 1)

| Thread (0, 0) | Thread (1, 0) | Thread (2, 0) | Thread (3, 0) | Thread (4, 0) |
| Thread (0, 1) | Thread (1, 1) | Thread (2, 1) | Thread (3, 1) | Thread (4, 1) |
| Thread (0, 2) | Thread (1, 2) | Thread (2, 2) | Thread (3, 2) | Thread (4, 2) |

CUDA Grid

Grid, Blocks, Threads

Our Array Additon Kernel: 1D block, 1D threads

```
__global__
void add(int n, float *x, float *y)
{
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;
  for (int i = index; i < n; i += stride)
    y[i] = x[i] + y[i];
}
```

Grid, Blocks, Threads
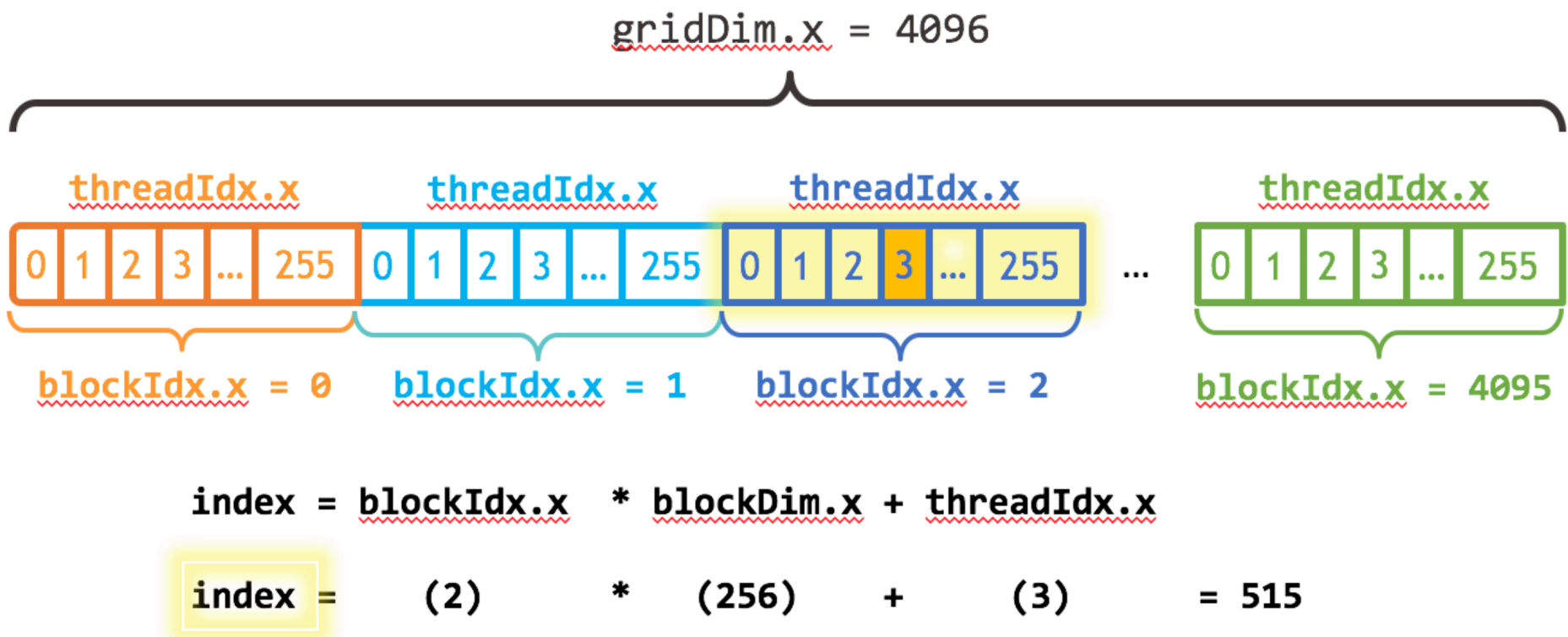
Our Array Additon Kernel: 1D block, 1D threads

Some Example Kernels

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

Assume blockDim.x = 4
Output:

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

Output:

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

Output:

Some Example Kernels

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

Assume blockDim.x = 4

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

More Example – Shuffling Data

```
// Reorder values based on indices
// Each thread moves one element
__global__ void shuffle(int* prev_array, int* new_array, int* indices)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i < N)
        new_array[i] = prev_array[indices[i]];
}
```

Host Code

```
int main()
{
    // Run grid of (N+255)/256 blocks of 256 threads each
    shuffle<<< (N+255)/256, 256>>>(d_old, d_new, d_ind);
}
```

More Example Kernel (2D)

```
__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 =

    a[idx]  = a[idx]+1;
}
```

More Example Kernel (2D)   cont'd

```
__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 =

    a[idx]  = a[idx]+1;
}
```

```
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 );

    free( h_a );
    cudaFree( d_a );
    return 0;
}
```
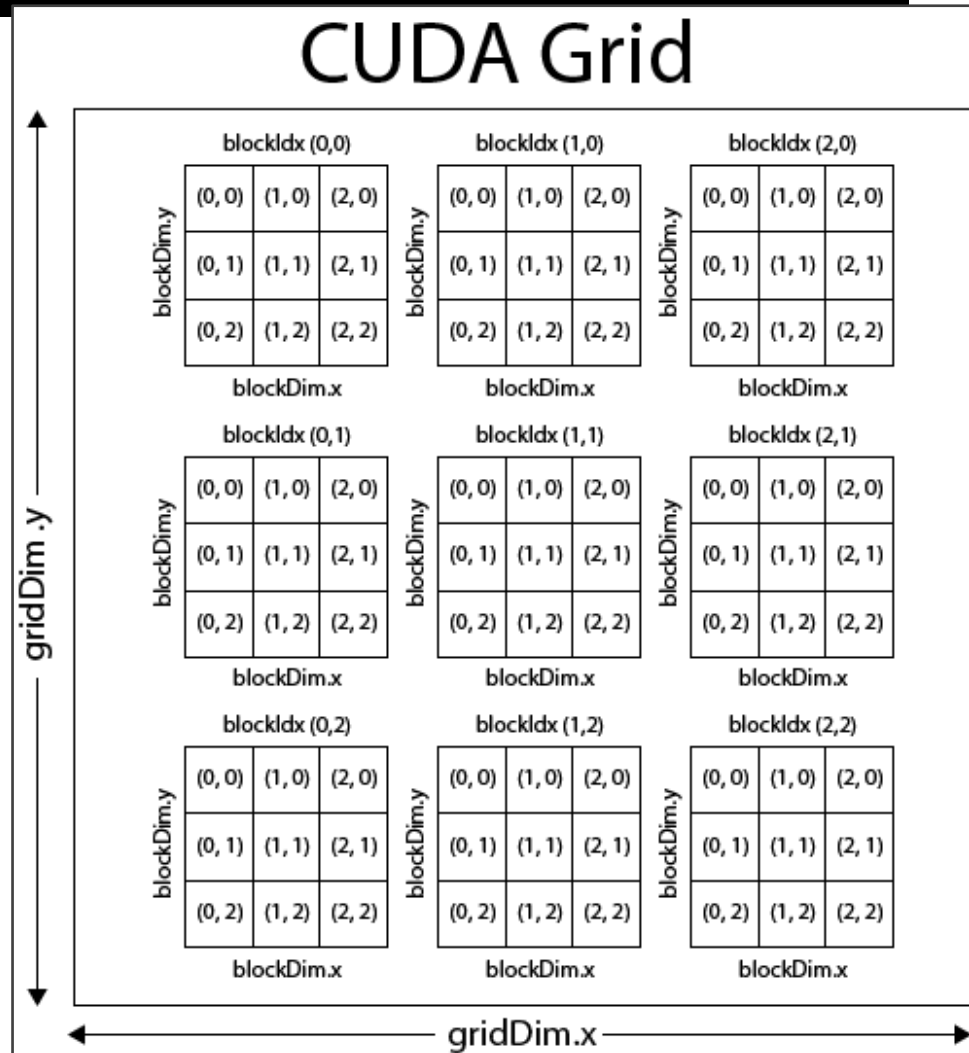
More Example Kernel (2D) cont'd

```
__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 = 

    a[idx]  = a[idx]+1;
}
```

CUDA Grid

blockIdx (0,0) | blockIdx (1,0) | blockIdx (2,0)

blockDim.y

(0, 0) | (1, 0) | (2, 0)
(0, 1) | (1, 1) | (2, 1)
(0, 2) | (1, 2) | (2, 2)

blockDim.x

blockIdx (0,1) | blockIdx (1,1) | blockIdx (2,1)

blockDim.y

(0, 0) | (1, 0) | (2, 0)
(0, 1) | (1, 1) | (2, 1)
(0, 2) | (1, 2) | (2, 2)

blockDim.x

blockIdx (0,2) | blockIdx (1,2) | blockIdx (2,2)

blockDim.y

(0, 0) | (1, 0) | (2, 0)
(0, 1) | (1, 1) | (2, 1)
(0, 2) | (1, 2) | (2, 2)

blockDim.x

gridDim.y
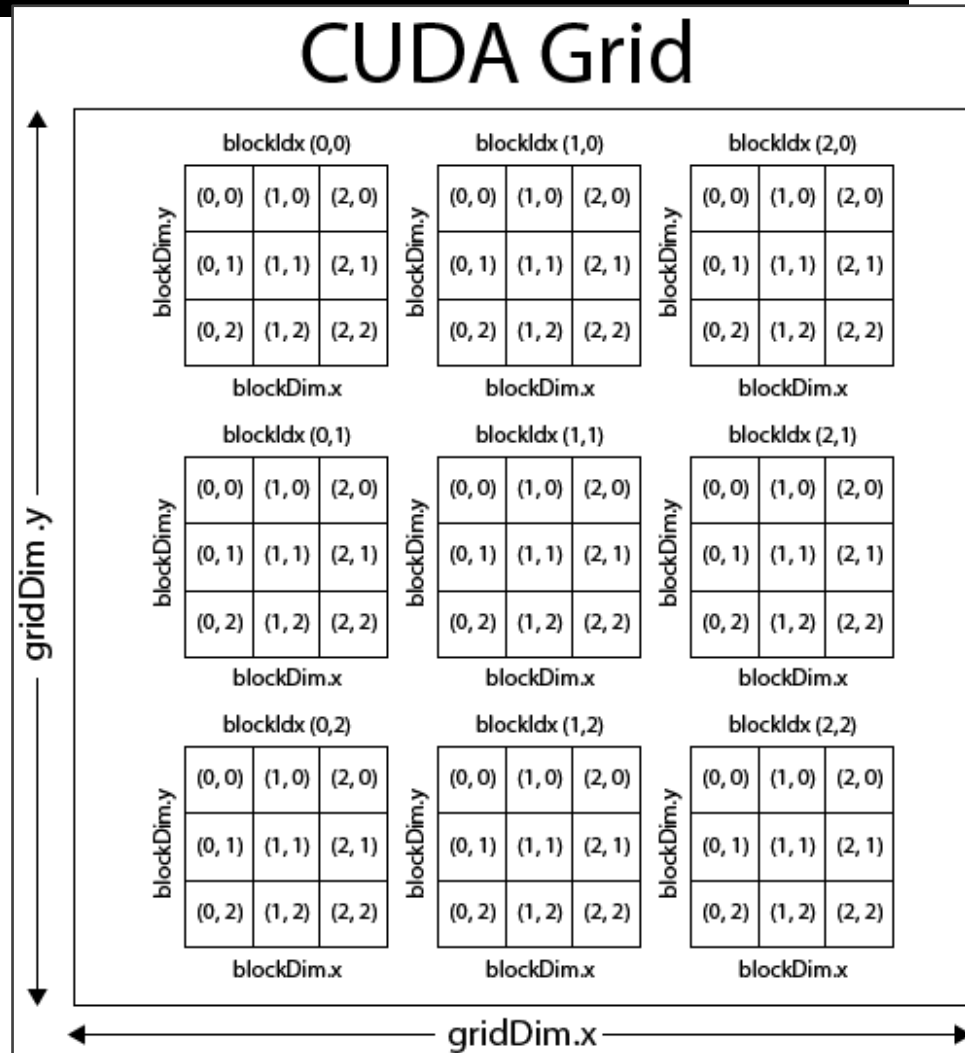
gridDim.x

More Example Kernel (2D)   cont'd

```
__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;
}
```

CUDA Grid

Communication Among Threads

- How do you do global communication?
- Finish a grid and start a new one

Global Communication

- Finish a kernel and start a new one
- All writes from all threads complete before a kernel finishes

```
step1<<<grid1,blk1>>>(...);
// The system ensures that all
// writes from step1 complete.
step2<<<grid2,blk2>>>(...);
```

Race Conditions

- Or, write to a predefined memory location
  - Race condition! Updates can be lost

Race Conditions

```
threadId:0                          threadId:1917
    // vector[0] was equal to 0
vector[0] += 5;                     vector[0] += 1;
...                                 ...
a = vector[0];                      a = vector[0];
```

- What is the value of `a` in thread 0?
- What is the value of `a` in thread 1917?

Race Conditions

- Thread 0 could have finished execution before 1917 started

- Or the other way around

- Or both are executing at the same time

Race Conditions

- Answer: not defined by the programming model, can be arbitrary

Atomics

- CUDA provides atomic operations to deal with this problem

Atomics

- An atomic operation guarantees that only a single thread access a memory address

- No race condition, but ordering is still arbitrary

- Different types of atomic instructions

- `atomic{Add, Sub, Exch, Min, Max, Inc, Dec, CAS, And, Or, Xor}`

- More types in Fermi

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions

Example: Histogram

```
// Determine frequency of colors in a picture
// Each thread looks at one pixel and increments
// a counter atomically
__global__ void histogram(int* color,
                                int* buckets){
  int i = threadIdx.x
        + blockDim.x * blockIdx.x;
  int c = colors[i];
  atomicAdd(&buckets[c], 1);
}
```

Example: Workqueue

```
// each thread gets a task from a shared queue
__global__
void workq(int* work_q, int* q_counter,
           int* output, int queue_max){
    int i = threadIdx.x
         + blockDim.x * blockIdx.x;
    int q_index =
      atomicInc(q_counter, queue_max);
    int result = do_work(work_q[q_index]);
    output[i] = result;
}
```

Atomics

- Atomics are slower than normal load/store

- You can have the whole machine queuing on a single location in memory

- Atomics unavailable on (very) old GPUs (G80)!

Example: Global Min/Max (Naive)

```
// If you require the maximum across all threads
// in a grid, you could do it with a single global
// maximum value, but it will be VERY slow
__global__
void global_max(int* values, int* gl_max){
    int i = threadIdx.x
            + blockDim.x * blockIdx.x;
    int val = values[i];
    atomicMax(gl_max,val);
}
```

Example: Global Min/Max (Better)

```
// uses intermediate/local maximum per thread
__global__
void global_max(int* values, int* max,
                int *regional_maxes,
                int num_regions){
  // i and val as before …
  int region = i % num_regions;
  if(atomicMax(&reg_max[region],val) < val)
  {
    atomicMax(max,val);
  }
}
```

Global Min/Max

- Single value causes serial bottleneck
- Create hierarchy of values for more parallelism
- Performance will still be slow, so use judiciously
- Even better version in the future!

Summary on Atomic Operations

- Can't use normal load/store for inter-thread communication because of race conditions

- Use atomic instructions for sparse and/or unpredictable global communication
  - See next lectures for shared memory and scan for other communication patterns

- Decompose data (very limited use of single global sum/max/min/etc.) for more parallelism