CUDA Programming

Week 3. Cuda Threads

Outline

- Thread, warp, and scheduling
- Branch divergence
- Instruction unrolling
- Homework

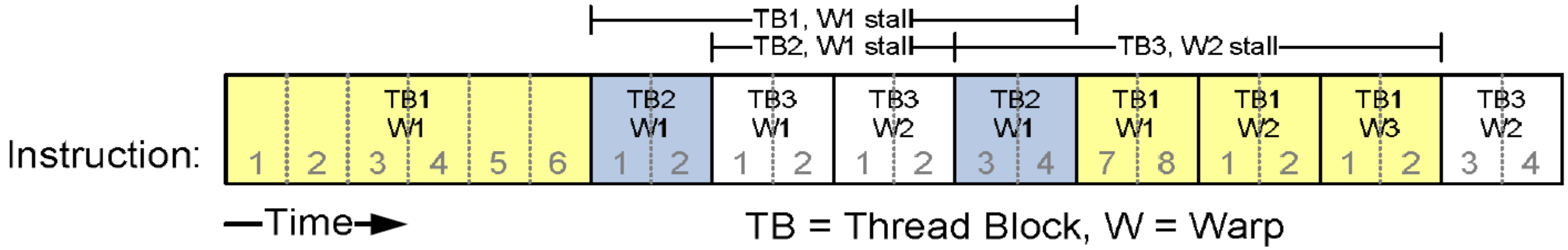
Thread assignment

- In execution, a block is assigned to an SM
- An SM can have up to 8 blocks, as long as there are enough resources for all blocks.
  - SMs dynamically partition hardware resources to threads and blocks during the runtime.
  - In G80, up to 768 threads per SM.
    - Need >=768/8=96 threads to fully utilize both resources
  - In G80, up to 512 threads per block.

Warps

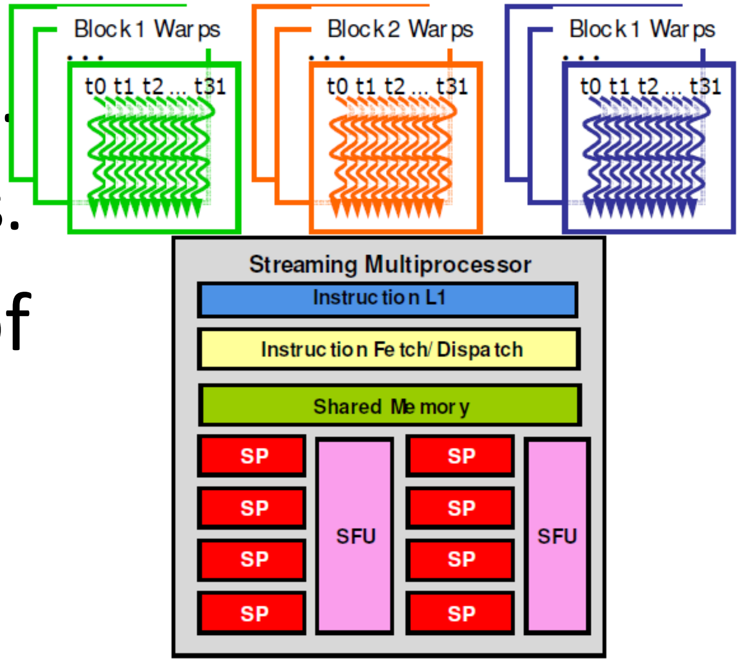
- In execution, threads are divided into warps.
  - All threads in a warp execute the same instruction.
  - In G80, each warp has 32-threads.
  - Thread 0-31 form the first warp, 32-63 the second warp, and so on.

- Warps are the unit of thread scheduling in SM
  - Context switch is zero-overhead.



Number of warps

- Suppose 3 blocks are assigned to an SM.
- If each block has 256 threads, how many warps in the SM?
  - Each block has 256/32 warps.
  - 3 blocks have 8*3 = 24 warps.
- 24 is the maximal number of warps in an SM in G80
  - Max 768 threads in each SM

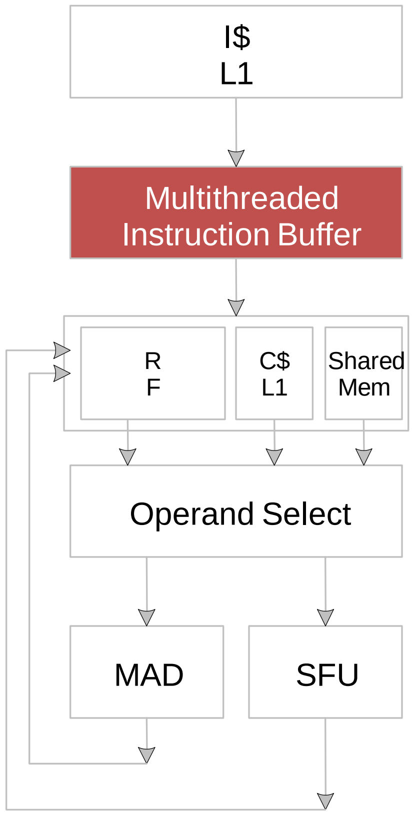


Thread scheduling

- Warps whose next instruction has its operands ready for consumption are eligible for execution
- 4 clock cycles needed to dispatch the same instruction for all threads in a Warp in G80
  - Why?
- Question: If one global memory access is needed for every 4 instructions, and each memory access takes 200 cycles, how many warps are needed to fully cover the latency
  - 13 Warps

SM Instruction Buffer:Warp Scheduling

- Fetch one warp instruction/cycle
  - from instruction L1 cache
  - into any instruction buffer slot
- Issue one “ready-to-go” warp instruction/cycle from any warp - instruction buffer slot
  - Issue selection based on round-robin/age of warp
  - Use operand scoreboarding to prevent hazards



Scoreboarding

- Determine if a thread is ready to execute
- Old concept from CDC 6600 (1960s) to separate memory and computation
- A scoreboard is a table in hardware that tracks
  - instructions being fetched, issued, executed
  - resources (functional units and operands) needed by instructions
  - which instructions modify which registers

GPU Scoreboarding

- All register operands of all instructions in the instruction Buffer are scoreboarded
  - Instruction becomes ready after the needed values are deposited
  - Prevents hazards
  - Cleared instructions are eligible for issue
- Decoupled Memory/Processor pipelines
  - Any thread can continue to issue instructions until scoreboarding prevents issue
  - Allows Memory/Processor ops to proceed in shadow of other waiting Memory/Processor ops

BRANCH DIVERGENCE

Branch divergence

- Threads within a single warp take different paths
  - Different execution paths are serialized in G80
  - The control paths taken by the threads in a warp are traversed one at a time until there is no more.
- Example with divergence: If (threadIdx.x > 2) { }
  - This creates two control paths for threads in a block: Threads 0, 1 and 2 follow different path than the rest of the threads in the first warp

Avoid branch divergence

- Example without divergence:
  - If (threadIdx.x / BLOCK_SIZE > 2) { }
  - Also creates two different control paths for threads in a block
  - Branch granularity is a whole multiple of warp size; all threads in any given warp follow the same path
    - BLOCK_SIZE = 16, and there are 16*3=48 threads are in the different paths, but they are in 3 warps.

EX: Parallel Reduction

- Given an array of values, “reduce” them to a single value in parallel
  - Sum reduction: sum of all values in the array
  - Max reduction: maximum of all values in the array
- Typically parallel implementation:
  - Recursively halve number of threads, reduce two values per thread
  - For n/2 threads, it takes log(n) steps for n elements

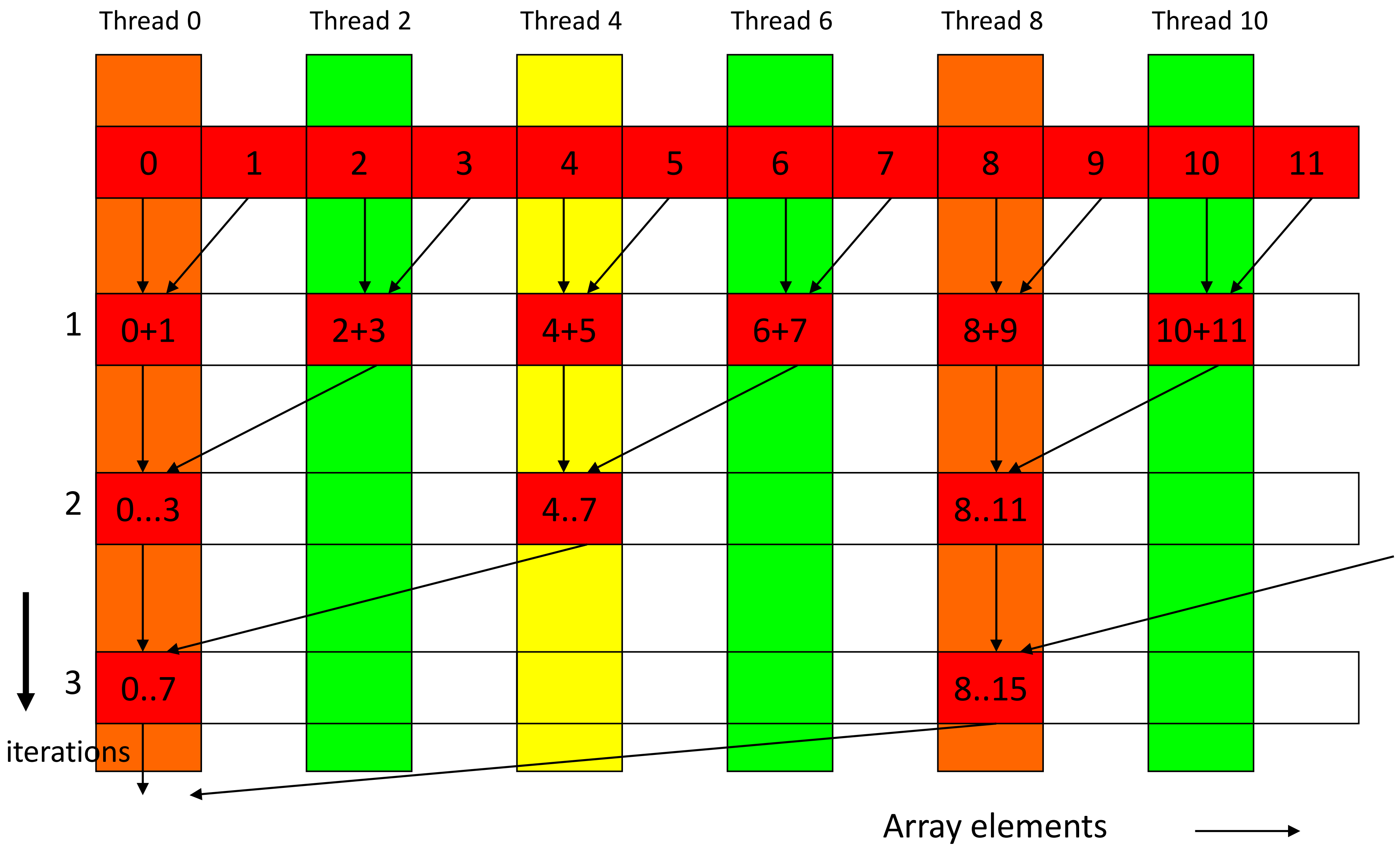
Implementation details

- Assume an in-place reduction using shared memory
- The original vector is in global memory
- The shared memory used to hold a partial sum vector
- Each iteration brings the partial sum vector closer to the final sum
- The final solution will be in element 0

Implementation

- Assume we have already loaded array into __shared__ float partialSum[]

```
unsigned int t = threadIdx.x;
for (unsigned int stride = 1;
            stride < blockDim.x;  stride *= 2)
{
  __syncthreads();
  if (t % (2*stride) == 0)
            partialSum[t] += partialSum[t+stride];
}
```


Thread 0
Thread 2
Thread 4
Thread 6
Thread 8
Thread 10
0
1
2
3
4
5
6
7
8
9
10
11
1
0+1
2+3
4+5
6+7
8+9
10+11
2
0...3
4..7
8..11
3
0..7
8..15
iterations
Array elements

Problems

- In each iterations, two control flow paths will be sequentially traversed for each warp.
- No more than half of threads will be executing at any time.
- After the 5th iteration, entire warps in each block will be disabled (no divergence).
- This can go on for a while, up to 4 more iterations ($512/32=16= 2^4$), where each iteration only has one thread activated until all warps retire .

Review the code

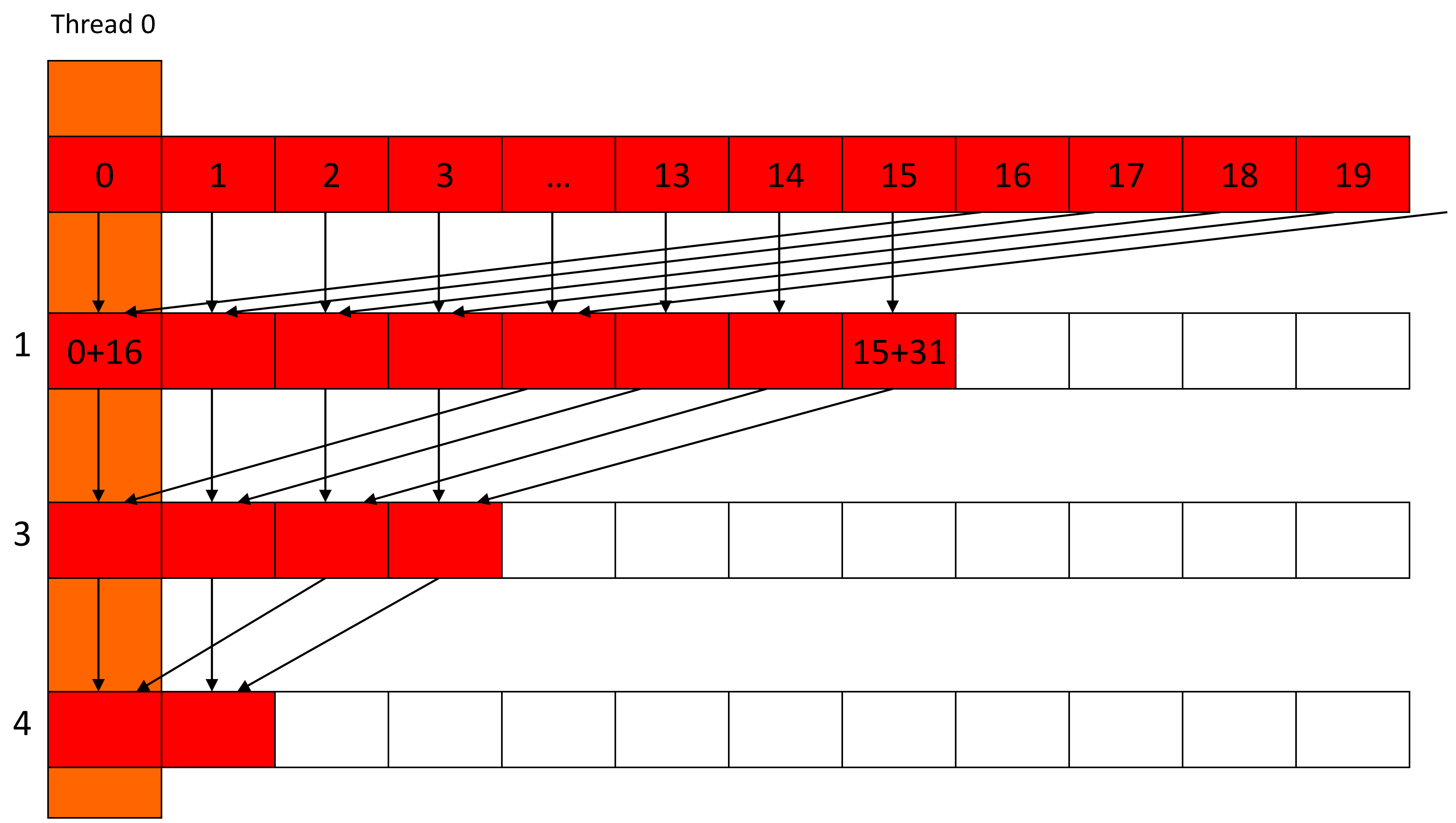
```
unsigned int t = threadIdx.x;
for (unsigned int stride = 1;
     stride<blockDim.x; stride*=2)
{
  __syncthreads();
  if (t % (2*stride) == 0)
     partialSum[t]+=partialSum[t+stride];
}
```

BAD: Divergence due to interleaved branch decisions

A better implementation

```
unsigned int t = threadIdx.x;
for (unsigned int stride = blockDim.x;
    stride > 1;  stride >> 1)
{
  __syncthreads();
  if (t < stride)
    partialSum[t] += partialSum[t+stride];
}
```

No Divergence until < 16 sub-sums



UNROLLING

Example

```
for(int k=0; k<16; ++k)
    Pvalue+=Ms[ty][k]*Ns[k][tx];
```

- There are very few mul/add between branches and address calculation.
- Loop unrolling

```
Pvalue+=Ms[ty][k]*Ns[k][tx]+…
     Ms[ty][k+15]*Ns[k+15][tx];
```

  – More adds for address calculation

#pragma unroll

- Compiler can help the unrolling using the #pragma directives

```
    #pragma unroll 5
    for (int i=0;i<n;i++)
```

- The loop will be unrolled 5 times
  - Need be careful for boundary case
- Use #pragma unroll 1 to prevent the compiler from ever unrolling a loop

HOMEWORK

Homework

- Since we have a break next time, let’s have more assignments
- 1. Implement summation using two different parallel ordering, and compare their performance result
- 2. Continue the matrix-matrix multiplication
  - Add tiling (more next slides)
  - Add unrolling

Block Granularity Considerations

- How many threads per block should be used?
  - For 8X8, we have 64 threads per Block. Since each SM can take up to 768 threads, there are 12 Blocks. However, each SM can only take up to 8 Blocks, only 512 threads will go into each SM!
  - For 16X16, we have 256 threads per Block. Since each SM can take up to 768 threads, it can take up to 3 Blocks and achieve full capacity unless other resource considerations overrule.
  - For 32X32, we have 1024 threads per Block. Not even one can fit into an SM!

- Each SM in G80 has 16KB shared memory
  - For TILE_WIDTH = 16, each thread block uses 2*256*4B = 2KB of shared memory.
  - Can potentially have up to 8 Thread Blocks
    - This allows up to 8*512 = 4,096 pending loads.
  - The next TILE_WIDTH 32 would lead to 2*32*32*4B= 8KB shared memory usage per thread block, allowing only 2 blocks
- Using 16x16 tiling, we reduce the accesses to the global memory by a factor of 16
  - The 86.4B/s bandwidth can now support (86.4/4)*16 = 347.6 GFLOPS!