Lec12 Synchronize threads within Block and Device Memories

Yun Tian (Tony) (Ph.D)

CSCD 439/539 GPU Computing

Summary We Learned

- GPU computing history and GPU device evolvement.
- Programming model
  - Global memory
  - Host memory vs. Device Memory
  - Map Thread Grid to 1D and 2D dataset
  - Simple Kernel with 1D and 2D grid
- Hardware features
  - Architecture of modern GPUs
- cuPrintf() and error handling
- Pointers to pointers on device

Recall Some Details

- Recall that threads:
  - Execute within blocks
  - Grouped into warp of 32 threads within blocks
  - Block warps are executed in turn on a multiprocessor
    - can have more than 32 threads per block
    - more is often better (1024 is max for new hardware now per block)
  - Threads in same block can all access global memory and a small fast **shared memory** (used to be 16KB, newer cards 48KB/block)

Outline for Today

- Synchronize threads within a block
- Different GPU memory types

Memory Model

- Recall host responsibility
  - allocate global memory on card device,

    float *d_dataB

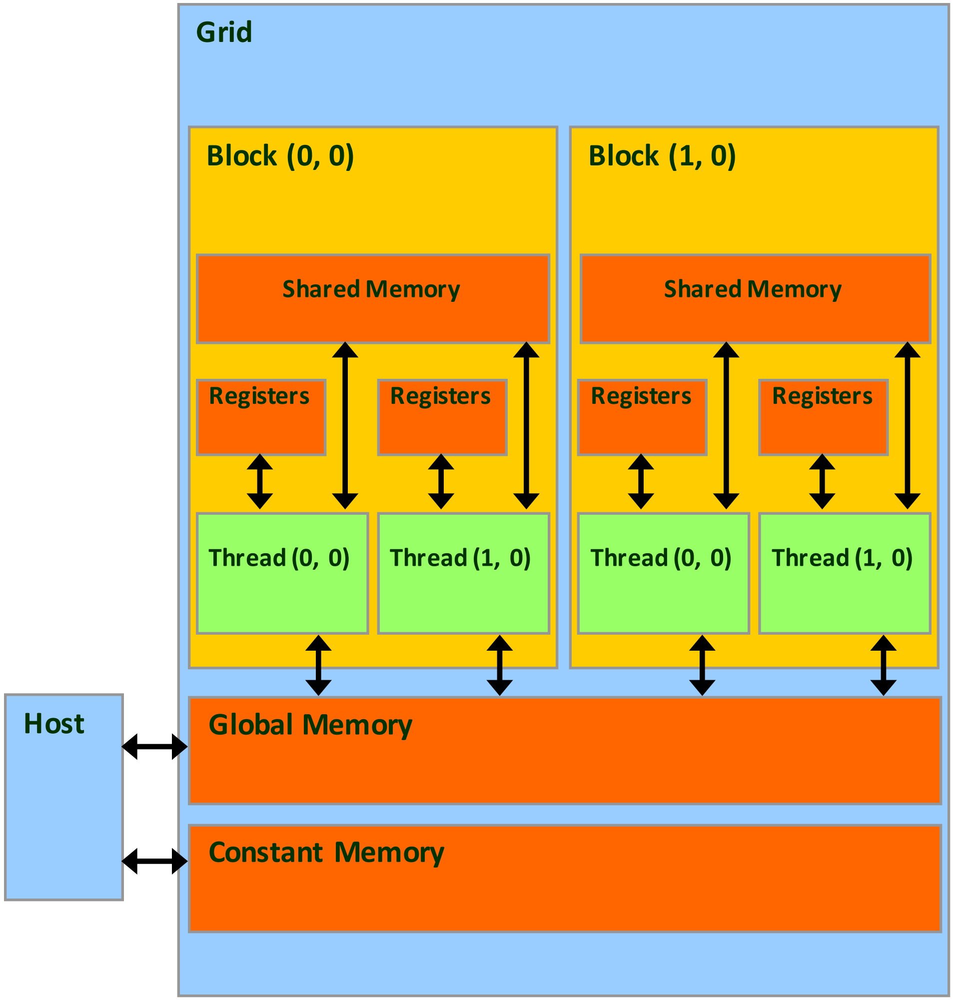
    cudaMalloc(&d_dataB, numDeviceBytes)
  - copy required data from host to device and vice versa.
  - choose block dimensions:
    - dim3 threads(blockwidth, blockheight,1)

Memory Model

- Recall host responsibility
  - choose grid dimensions:
    - dim3 grid(1, numBlocks, 1)
  - compute total dynamic shared memory needed by a block, int sharedMemorySize = ??
  - execute kernel:
  - kernel_2<<<grid, threads, sharedMemorySize>>>(d_dataA, d_dataB)

Hardware Implementation of CUDA Memories

- **Each thread can:**
  - **Read/write per-thread registers**
  - **Read/write per-thread local memory**
  - **Read/write per-block shared memory**
  - **Read/write per-grid global memory**
  - **Read/only per-grid constant memory**



CUDA Variable Type Qualifiers

Variable declaration	Memory	Scope	Lifetime
`int var;`	register	thread	thread
`int array_var[10];`	local	thread	thread
`__shared__ int shared_var;`	shared	block	block
`__device__ int global_var;`	global	grid	application
`__constant__ int constant_var;`	constant	grid	application

- **“automatic” scalar variables without qualifier reside in a register**
  - **compiler will spill to thread local memory**
- **“automatic” array variables without qualifier reside in thread-local memory. But physically use a piece of global memory.**

CUDA Variable Type Performance

Variable declaration	Memory	Penalty
`int var;`	register	1x
`int array_var[10];`	local	100x
`__shared__ int shared_var;`	shared	1x
`__device__ int global_var;`	global	100x
`__constant__ int constant_var;`	constant	1x

- scalar variables reside in fast, on-chip registers
- shared variables reside in fast, on-chip memories
- thread-local arrays & global variables reside in uncached off-chip memory
- constant variables reside in cached off-chip memory

CUDA Variable Type Scale

Variable declaration	Instances	Visibility
`int var;`	100,000s	1
`int array_var[10];`	100,000s	1
`__shared__ int shared_var;`	1000s	100s
`__device__ int global_var;`	1	100,000s
`__constant__ int constant_var;`	1	100,000s

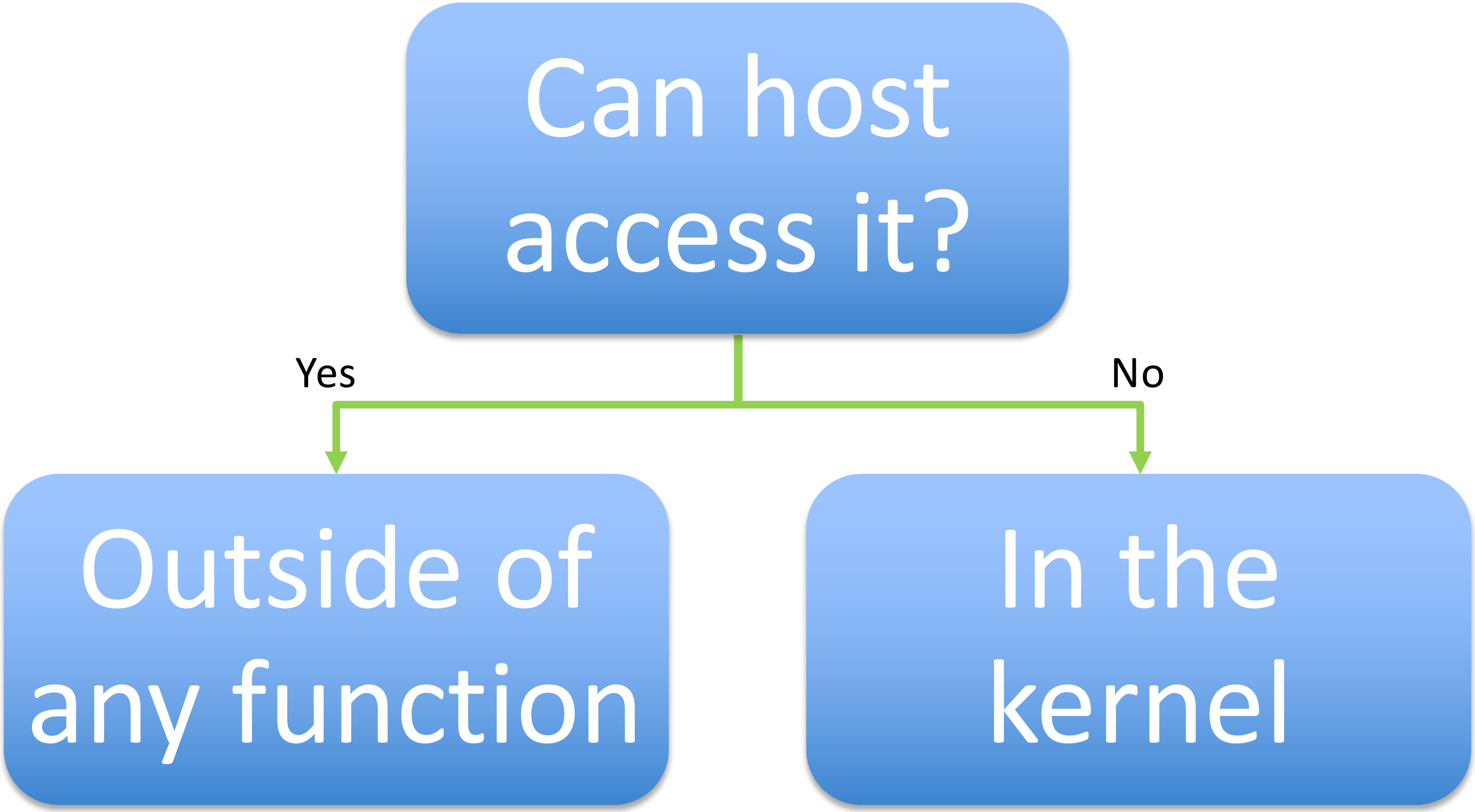
If we have 1000 blocks and each block has 100 threads.

- 100Ks per-thread variables, R/W by 1 thread
- 1000s shared variables, each R/W by 100s of threads
- 1 global variable is R/W by 100Ks threads
- 1 constant variable is readable by 100Ks threads

Constant Memory

- http://cuda-programming.blogspot.com/2013/01/what-is-constant-memory-in-cuda.html
- Ideally, for threads in a warp(or half), reading from the constant cache is as fast as reading from a register as long as all threads read the **same or nearby addresses**.
- Cost scales linearly with the number of different addresses read by all threads within a warp(half).
- Demo of Constant Memory

Where to declare variables?




```
__constant__ int constant_var;
__device__   int global_var;

int var;
int array_var[10];
__shared__   int shared_var;
```

Shared Memory

- Per-Block Shared memory
  - global memory is vey slow, costs 100-200 simple instructions to access.
  - shared memory costs only 1 or 2 single instruction.
  - Threads within a block can be synchronized with a barrier.

Shared Memory

- Synchronization of threads within a block

  __syncthreads();

  - barrier synchronization between all threads in a block.
  - This does not apply between blocks.
  - Remember how to synchronize threads between blocks?
    - No built-in mechanism in Cuda for threads from different blocks waiting for each other.
    - Decompose one kernel into multiple ones.
    - Launch them one after another.

synchthreads()

- Barrier synchronization for threads within a block.
- Allows threads in a block to wait for each other until **all** threads in that block reached that barrier points, then they can move forward.
    - Acts like Barrier Object in Pthread and Java threads.

synchthreads()

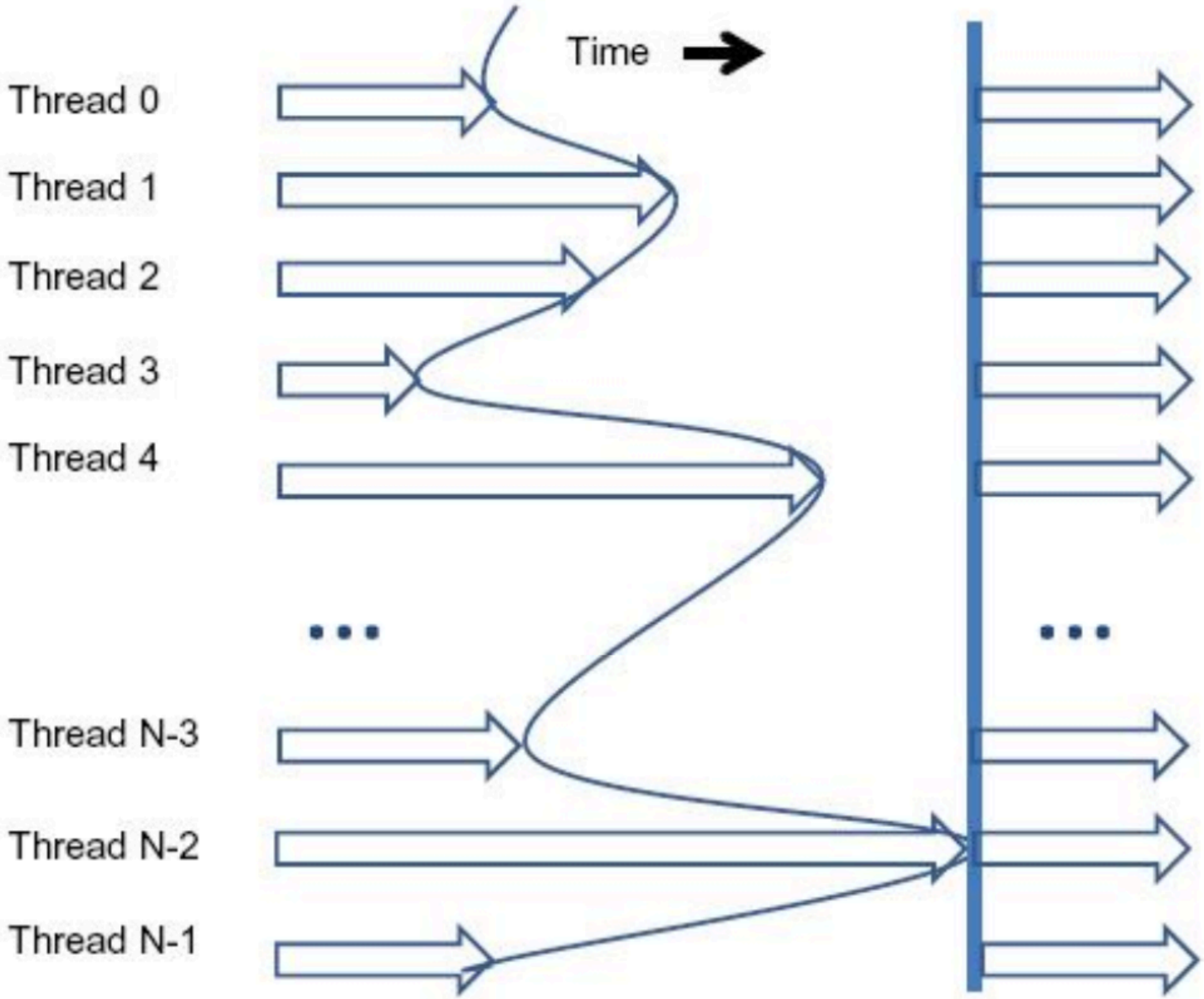


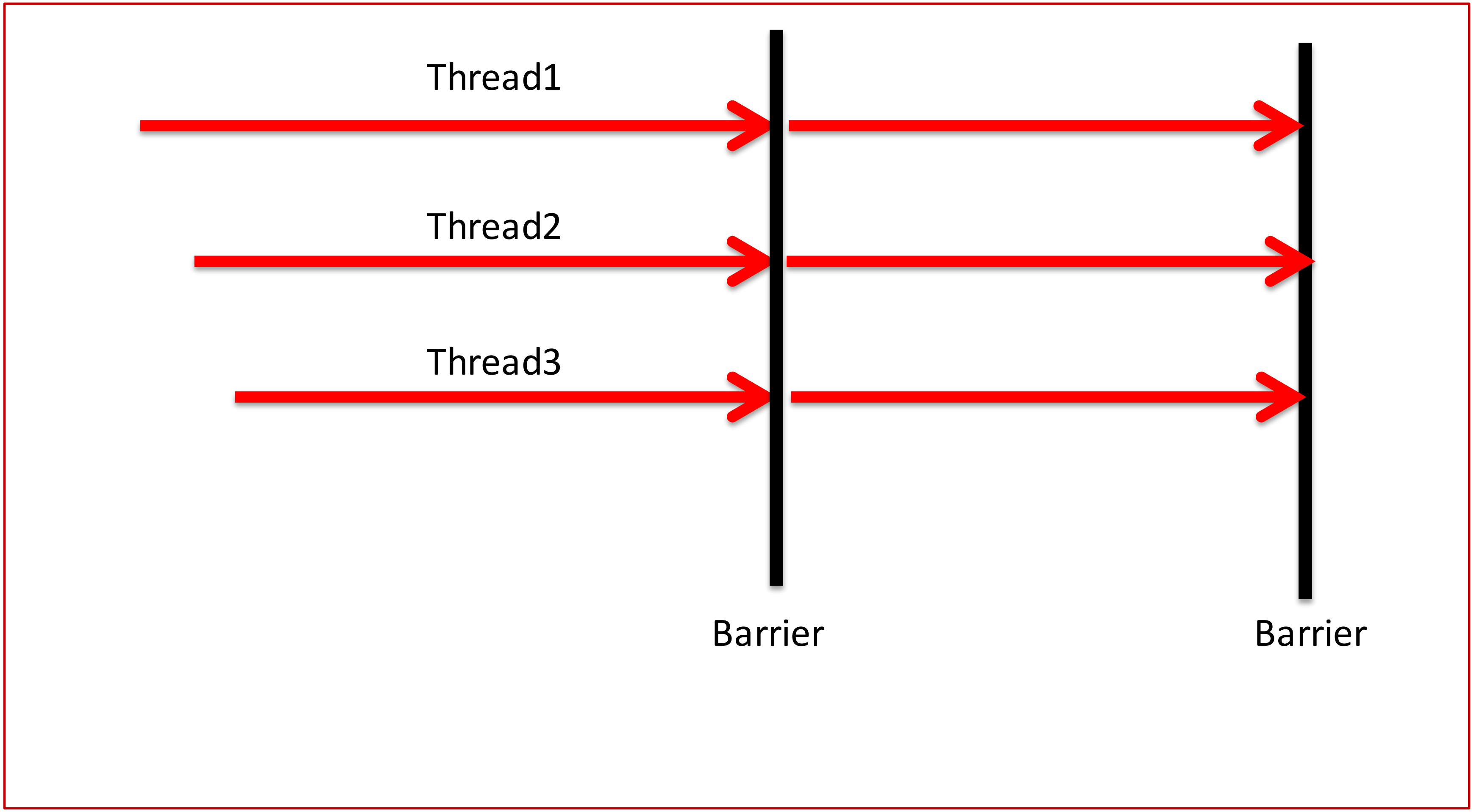
FIGURE 4.11

An example execution timing of barrier synchronization.

Barrier Synchronization

```
//inside kernel method shared by multiple threads
for(int i = 0; i < 500; i ++) {
    //do work here
    //
    if( i == 100) {
        __syncthreads(); //barrier pointer
    }
}
```

Barrier Synchronization



syncthreads()

- **All** threads in that block **must** execute __syncthreads(), or we will deadlock.
  - makes sure all threads involved in the barrier eventually get the resources and arrive at the barrier.
- Be careful when use it inside if statement
  - You have to make sure ALL threads will evaluate the if condition either true or false.
  - If some threads go true branch and some go false branch,
    - Deadlock, Why?

synchthreads()

```
//inside a kernel
if ( a[idx] % 2 == 0 )
{
       dowork1();
       __syncthreads();
}
else
{
      dowork2();
      __syncthreads();
}
//what will happen?
```

We have two barrier points, all threads have to go either if true branch and dowork1,
Or all threads have to go else branch dowork2().
Otherwise, we deadlock.

syncthreads()

- When is __syncthreads() necessary or useful?

When one thread needs a result computed by another thread.
This thread has to wait until the result is ready.

- Note: There is a natural synch happening after an if-statement or similar conditional.

Wrap Up

- Different types of GPU memory
- __syncthreads() for threads within blocks.
- Next Class, case study of shared memory and why we use shared memory?