

# Distributed and Parallel Computing

## Lecture 03

Alan P. Sexton

University of Birmingham

Spring 2018

# SMs, Cores and Warps

A GPU has a number of *Streaming Multiprocessors (SMs)*, which have a number of *Cores*. Threads are scheduled in units of *Warps*. Each SM has a number of resources. For example, each SM of a Fermi architecture GPU (GeForce 400 and 500 series) might have 2 instruction dispatch units and

- 3 banks of 16 cores (i.e. 48 cores)
- 1 bank of 16 Load Store Units (for calculating source and destination addresses for 16 threads per clock cycle)
- 1 bank of 4 Special Functional Units (hardware support for calculating sin, cos, reciprocals and square roots - a warp executes over 8 clocks).
- 1 bank of 4 Texture Units

The 2 instruction dispatch units can start, on each clock cycle, processing on any 2 banks at a time. The 3 banks of 16 cores means that 2 sequential instructions from one thread warp can be executing simultaneously if they are not dependent on each other (superscalar instruction parallelism)

Conceptually, Warps execute in lock step, so synchronisation within a warp is (mostly) automatic but can be tricky (data variables should be marked *volatile*)

- In practice, it is much more complicated.
- We need to be able to synchronise threads in a block
- Also need to be able to synchronise threads in a warp
- Cannot synchronise across different blocks
- **Barrier synchronisation:** `__syncthreads()`
- Must **NEVER** have `__syncthreads()` in a branch of a conditional that some threads in the block will not execute
  - Otherwise deadlock may occur!
- Cannot even fix it by making sure each branch has a `__syncthreads()` call: the different calls are **NOT** necessarily to the same barrier!

# More on Warps

For the threads in a block,

- Warp 0 consists of threads 0 to 31
- Warp 1 consists of threads 32 to 63
- ...

For threads in a multi-dimensional block

- Multidimensional threads are linearized in row-major order
- Thus all the threads with z value 0 come first, followed by those with z value 1, etc.
- Within each group of threads with the same z value, the threads with y value 0 come first, followed by those with y value 1 etc.
- Within each group of threads with the same z and y value, the thread with x value 0 comes first followed by that with x value 1 etc.
- Within this linearized order, the first 32 threads belong to the first warp etc.

# Warp Execution and Divergence

The whole warp is handled by a single controller. Consider what happens if some threads (A) in a warp take one branch (1) of an `if` statement, and others (B) take the other branch (2):

- All threads in the warp must execute the same instructions
- First execute branch 1: all threads execute the branch 1 instructions but the B threads are disabled (think of de-clutching in a car) so they have no effect.
- Then execute branch 2: all threads execute the branch 2 instructions but now the A threads are disabled.
- This is called a *divergence*
- Whole warp execute same branch  $\Rightarrow$  **NO DIVERGENCE**

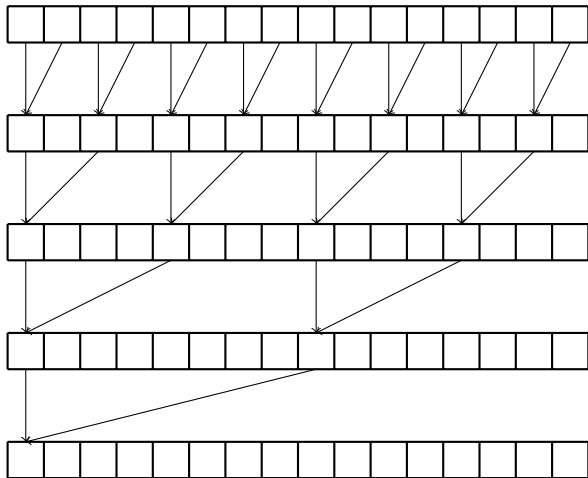
Loops where different threads in the warp execute different numbers of iterations also form divergences: The threads that execute the fewest number of iterations wait on the threads that execute the most number.

# Divergence and Reduction

Let's see the consequences of divergence for reduction: the reduction of a set of numbers to one e.g. finding the sum of a vector of length 1024

- Sequentially: iterate through a vector accumulating the sum in a variable (register)
- In parallel, one approach: use a binary tournament:
  - Round 1:
    - thread 0 executes  $A[0] += A[1]$
    - thread 2 executes  $A[2] += A[3]$
    - ...
    - thread  $2i$ , where  $0 < 2i < 1024$ , executes  $A[2*i] += A[2*i+1]$
  - Round 2:
    - thread  $4i$ , where  $0 < 4i < 1024$ , executes  $A[4*i] += A[4*i+2]$
  - ...
  - Round  $n$ :
    - thread  $2^n i$  executes  $A[2^n * i] += A[2^n * i + 2^{n-1}]$

# Naive Parallel Reduction



# Naive Parallel Reduction Code

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



# Naive Parallel Reduction Code

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

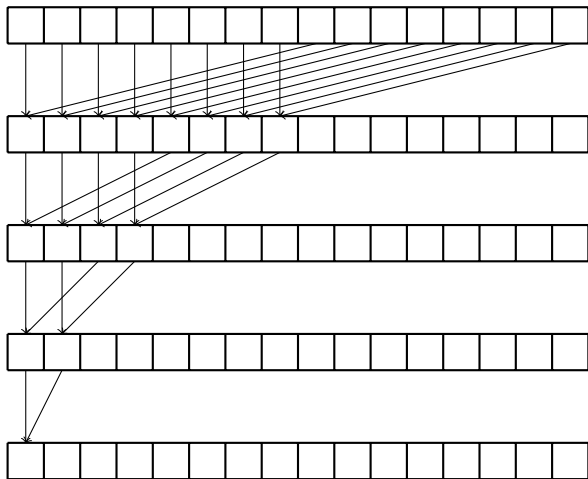
- Note the `__syncthreads()`: necessary to make sure all threads have completed previous stage.

# Naive Parallel Reduction Code

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

- Note the `__syncthreads()`: necessary to make sure all threads have completed previous stage.
- Assume `blockDim.x` is 1024
- Iteration 1: Even threads execute add: 1 pass for true branch, 1 pass for false branch, 512 threads = 16 warps all active
- All iterations: 2 passes each iteration
- In progressive iterations, fewer threads doing real work

# Parallel Reduction, Alternative Strategy



# Alternative Parallel Reduction Code

```
float partialSum[]  
...  
uint t = threadIdx.x;  
for (uint stride = blockDim.x/2 ; stride > 1; stride /= 2)  
{  
    __syncthreads();  
    if (t < stride)  
        partialSum[t] += partialSum[t+stride] ;  
}
```

# Alternative Parallel Reduction Code

```
float partialSum[]
...
uint t = threadIdx.x;
for (uint stride = blockDim.x/2 ; stride > 1; stride /= 2)
{
    __syncthreads();
    if (t < stride)
        partialSum[t] += partialSum[t+stride] ;
}
```

- Assume blockDim.x is 1024
- Threads 0-511 = warps 0-15 execute true branch, threads 512-1023 = warps 16-31 execute false branch, thus no divergence  $\Rightarrow$  1 pass each iteration

# Alternative Parallel Reduction Code

```
float partialSum[]
...
uint t = threadIdx.x;
for (uint stride = blockDim.x/2 ; stride > 1; stride /= 2)
{
    __syncthreads();
    if (t < stride)
        partialSum[t] += partialSum[t+stride] ;
}
```

- Assume blockDim.x is 1024
- Threads 0-511 = warps 0-15 execute true branch, threads 512-1023 = warps 16-31 execute false branch, thus no divergence  $\Rightarrow$  1 pass each iteration
- Continues 1 pass each iteration until less than 32 threads executing

Global memory on a GPU is separated from the SMs by a bus and is of DRAM type (i.e. based on capacitors holding charges). This makes data access slow (100s of clock cycles) and limited bandwidth (can't get many words at a time)

GPU	Mem Clock Rate	Mem Bus	Peak Bandwidth
GT 610	535 MHz	64 bits	4.280 GB/s
GTX 960	3600 MHz	128 bits	57.600 GB/s
GTX 1050 Ti	3504 MHz	128 bits	56.064 GB/s
GTX 1080 Ti	5505 MHz	352 bits	242.220 GB/s
Titan V	850 MHz	3072 bits	326.400 GB/s

# Memory Bandwidth as a Performance Barrier

- For simple operations (e.g. `vectorAdd`) the *compute to global memory access* ratio (CGMA) is 1/3 (1 flop to 3 memory accesses - 2 reads and a write).
- Assume the global memory access bandwidth is of the order of 200GB/s, and the processor can execute of the order of 1500 GFLOPS, then memory bandwidth is limiting us as follows:
  - 3 read/writes = 12 bytes
  - 12 bytes memory access at 200GB/s for each flop =  $200/12$  GFLOPS = 17 GFLOPS
  - Thus though the hardware is capable of 1500 GFLOPS, our global memory latencies for this application limits us to 17 GFLOPS.



# CUDA Memories

Limits quoted for the GeForce GTX 960:

Memory	Scope	Lifetime	Speed	Limits
Register	Thread	Kernel	Ultra fast	65536/block
Local	Thread	Kernel	Very slow	(part of Global)
Shared	Block	Kernel	Very fast	49152 bytes/block
Global	Grid	Application	Very slow	1996 MBytes
Constant	Grid	Application	Very fast	65536 bytes (in Global but cached)

Using different memory types:

Memory	Variable Declaration
Register	Automatic variables other than arrays
Local	Automatic array variables
Shared	<code>__device__ __shared__ int var;</code>
Global	<code>__device__ int var;</code>
Constant	<code>__device__ __constant__ int var;</code>

Now reconsider parallel reduction:

- Each reduction iteration reads two words from global memory and writes one word back to global memory per working thread
- If instead the first read copied from global to shared memory, the remainder of the operation would be hugely faster
- In general, many operation can be executed in a *tiling* fashion, where a large problem in global memory can be broken into small tiles, each of which fits in shared memory, the tiles can be solved and then the results recombined.