Distributed and Parallel Computing

Lecture 2

Alan P. Sexton

University of Birmingham

Spring 2019

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)

GTX-960

```
Device 0: "GeForce GTX 960"
CUDA Driver Version / Runtime Version          8.0 / 7.0
CUDA Capability Major/Minor version number:    5.2
Total amount of global memory:                 1996 MBytes (2092957696 bytes)
( 8) Multiprocessors, (128) CUDA Cores/MP:     1024 CUDA Cores
GPU Max Clock rate:                            1291 MHz (1.29 GHz)
Memory Clock rate:                             3600 Mhz
Memory Bus Width:                              128-bit
L2 Cache Size:                                 1048576 bytes
Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
Total amount of constant memory:               65536 bytes
Total amount of shared memory per block:       49152 bytes
Total number of registers available per block: 65536
Warp size:                                     32
Maximum number of threads per multiprocessor:  2048
Maximum number of threads per block:           1024
Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
Maximum memory pitch:                          2147483647 bytes
Texture alignment:                             512 bytes
Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
Run time limit on kernels:                     No
Integrated GPU sharing Host Memory:            No
Support host page-locked memory mapping:       Yes
Alignment requirement for Surfaces:            Yes
Device has ECC support:                        Disabled
Device supports Unified Addressing (UVA):      Yes
Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
Compute Mode: < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
```

GTX-960 Computation

Remember:

- 1 core is 1 processing unit, can execute 1 thread,
- Need 32 cores together for 1 warp

Critical details of the GTX-960:

- Maxwell micro-architecture
  - c.f.: `https://docs.nvidia.com/cuda/maxwell-tuning-guide/index.html`
  - 4 warp schedulers/Streaming Multiprocessor (SM)
  - Each scheduler can issue up to 2 (independent) instructions from the same warp each cycle (e.g. an ALU op simultaneously with a load/store op)
  - Maximum number of blocks/SM = 32
- 8 SMs, 128 cores/SM (= 1024 cores in total)
- Maximum 2048 threads/SM
- Maximum 1024 threads/block
- GPU maximum clock rate: 1.29GHz

GTX-960 Computation, cntd.

When a kernel is invoked (with config of Grid, Blocks and Threads):

- Each block is assigned to an SM based on availability
- Max 1024 threads/block ⇒ 2 blocks can be live ("in flight") on each SM
  - More if you have less threads per block
- When a block is assigned to an SM, the warps of the block are distributed statically among the 4 warp schedulers.
- With max 2048 threads/SM (64 warps), each scheduler gets at most 16 warps (possibly from different blocks).
- Maximum number of blocks/SM is 32, so if you have less than 64 threads per block you can not fill the full SM thread limit, which may lead to under occupancy.

GTX-960 Computation, cntd.

- Each scheduler can issue up to 2 (independent) instructions from the same warp each cycle
- Each scheduler issues 1 instruction from one warp per cycle, if it has a warp ready to execute.
  - If the current warp is not ready (e.g. waiting on a memory transaction or an FP function unit), the scheduler switches to an alternative warp assigned to this scheduler that is ready without costing any delay if such a warp exists.
  - The advantage of each scheduler having upto 16 warps to schedule means that you can cover quite a bit of latency while waiting for a delayed instruction to complete (mem, function unit etc.) by switching between the other 15 warps.
- Thus in the theoretical optimal case, if you have a kernel with $32 \times N$ threads in total, and the kernel is K instructions long, you could potentially execute the kernel in $\frac{N \times K}{8 \times 4}$ cycles at maximum 1.29 GHz (8 multiprocessors, 4 warp schedulers)

GTX960 Computation in Practice

Assume a kernel with 50,000 threads and a block size of 1024 threads/block

- Maximum 2 blocks/SM
- Need 50,000/1024 = 48.828125 blocks: round up to 49 block
    - Too few blocks and some threads won't execute
    - Too many blocks and some blocks will be scheduled but not do useful work (taking resources in the process)
- The first 16 blocks are assigned to the 8 SMs and start to run
- Each block has 32 warps, thus each SM has 64 warps and 4 out of the 64 can actually run at a time on each SM

GTX960 Computation in Practice, cntd.

- If a warp finishes, or has to wait for an operation to complete, the SM switches to a different warp without any delays if there is one ready to go
- If a warp has to wait and no other warps are ready to run, then the warp scheduler and the corresponding cores are sitting idle. This can happen if:
  - your configuration has blocks that are unnecessarily small or
  - the block is nearing completion and most of the warps have already finished
- An SM has high occupancy if there are many warps in flight, ready to switch in and use the cores if the current warp stalls
- Low occupancy means higher probability of not getting useful work done if a warp stalls

GTX960 Computation in Practice, cntd.

- Threads within a warp run in lock-step. Threads in different warps do not
- When all the warps in a block have finished, the next waiting block is loaded into the SM and starts running
- Blocks do NOT execute in lock step
  - Like a shared queue at a bank or postoffice: All the blocks are in a single queue, when any SM has a free slot that can accommodate a block, the first block on the queue takes it.

GTX960 Computation in Practice, cntd.

- At the end, there are too few blocks to fully occupy the SM
- Typically want the block size to be as large as possible to maximise occupancy, but memory constraints or problem structure may dictate a lower block size
- Sometimes a block size below the maximum possible may give greater perfomance: while occupancy is not optimal, it may be good enough that more benefits are gained by spreading the load over more SMs towards the end of the computation.

GTX960 Computation in Practice, cntd.

Back to our 50,000 thread example:

- Assume a wait-free warp can execute the kernel in 1 second
- Assume, because of waits, it actually takes 10s
- Further assume, with 16 or more warps per warp scheduler, latency is fully covered so, on average, and with 1 warp scheduler, each warp takes 1s
- For simplicity, assume that, "coincidentally", blocks that start together finish together, thus execution runs in "phases", with 16 blocks per phase

GTX960 Computation in Practice, cntd.

- Choose a block size of 1024:
  - 2 blocks/SM
  - $\lceil 50,000/1024 \rceil = 49$ blocks of 1024 threads (32 warps)

GTX960 Computation in Practice, cntd.

- Choose a block size of 1024:
  - 2 blocks/SM
  - $\lceil 50,000/1024 \rceil = 49$ blocks of 1024 threads (32 warps)
  - 16 blocks / phase $\Rightarrow \lceil 49/16 = 3.0625 \rceil = 4$ phases

GTX960 Computation in Practice, cntd.

- Choose a block size of 1024:
    - 2 blocks/SM
    - $\lceil 50,000/1024 \rceil = 49$ blocks of 1024 threads (32 warps)
    - 16 blocks / phase $\Rightarrow \lceil 49/16 = 3.0625 \rceil = 4$ phases
    - 2 warp schedulers/block $\Rightarrow$ each block takes 16s
    - Total time: $4 \times 16 =$ 64s

GTX960 Computation in Practice, cntd.

- Choose a block size of 1024:
  - 2 blocks/SM
  - $\lceil 50,000/1024 \rceil = 49$ blocks of 1024 threads (32 warps)
  - 16 blocks / phase $\Rightarrow \lceil 49/16 = 3.0625 \rceil = 4$ phases
  - 2 warp schedulers/block $\Rightarrow$ each block takes 16s
  - Total time: $4 \times 16 =$ 64s
  - Actually, last phase has 1 block with 26.5 warps and 4 warp schedulers$\Rightarrow$ only takes 12s (7 warps/scheduler, some waits)$\Rightarrow$ final total: 60s

GTX960 Computation in Practice, cntd.

- Choose a block size of 1024:
  - 2 blocks/SM
  - $\lceil 50,000/1024 \rceil = 49$ blocks of 1024 threads (32 warps)
  - 16 blocks / phase $\Rightarrow \lceil 49/16 = 3.0625 \rceil = 4$ phases
  - 2 warp schedulers/block $\Rightarrow$ each block takes 16s
  - Total time: $4 \times 16 =$ 64s
  - Actually, last phase has 1 block with 26.5 warps and 4 warp schedulers$\Rightarrow$ only takes 12s (7 warps/scheduler, some waits)$\Rightarrow$ final total: 60s
- Choose a block size of 512:
  - 4 blocks/SM
  - $\lceil 50,000/512 \rceil = 98$ blocks of 512 threads (16 warps)

GTX960 Computation in Practice, cntd.

- Choose a block size of 1024:
  - 2 blocks/SM
  - $\lceil 50,000/1024 \rceil = 49$ blocks of 1024 threads (32 warps)
  - 16 blocks / phase $\Rightarrow \lceil 49/16 = 3.0625 \rceil = 4$ phases
  - 2 warp schedulers/block $\Rightarrow$ each block takes 16s
  - Total time: $4 \times 16 =$ 64s
  - Actually, last phase has 1 block with 26.5 warps and 4 warp schedulers$\Rightarrow$ only takes 12s (7 warps/scheduler, some waits)$\Rightarrow$ final total: 60s
- Choose a block size of 512:
  - 4 blocks/SM
  - $\lceil 50,000/512 \rceil = 98$ blocks of 512 threads (16 warps)
  - 32 blocks / phase $\Rightarrow \lceil 98/32 = 3.0625 \rceil = 4$ phases

GTX960 Computation in Practice, cntd.

- Choose a block size of 1024:
  - 2 blocks/SM
  - $\lceil 50,000/1024 \rceil = 49$ blocks of 1024 threads (32 warps)
  - 16 blocks / phase $\Rightarrow \lceil 49/16 = 3.0625 \rceil = 4$ phases
  - 2 warp schedulers/block $\Rightarrow$ each block takes 16s
  - Total time: $4 \times 16 =$ 64s
  - Actually, last phase has 1 block with 26.5 warps and 4 warp schedulers$\Rightarrow$ only takes 12s (7 warps/scheduler, some waits)$\Rightarrow$ final total: 60s
- Choose a block size of 512:
  - 4 blocks/SM
  - $\lceil 50,000/512 \rceil = 98$ blocks of 512 threads (16 warps)
  - 32 blocks / phase $\Rightarrow \lceil 98/32 = 3.0625 \rceil = 4$ phases
  - 1 warp scheduler/block $\Rightarrow$ each block takes 16s
  - Total time: $4 \times 16 =$ 64s

GTX960 Computation in Practice, cntd.

- Choose a block size of 1024:
  - 2 blocks/SM
  - $\lceil 50,000/1024 \rceil = 49$ blocks of 1024 threads (32 warps)
  - 16 blocks / phase $\Rightarrow \lceil 49/16 = 3.0625 \rceil = 4$ phases
  - 2 warp schedulers/block $\Rightarrow$ each block takes 16s
  - Total time: $4 \times 16 =$ 64s
  - Actually, last phase has 1 block with 26.5 warps and 4 warp schedulers$\Rightarrow$ only takes 12s (7 warps/scheduler, some waits)$\Rightarrow$ final total: 60s
- Choose a block size of 512:
  - 4 blocks/SM
  - $\lceil 50,000/512 \rceil = 98$ blocks of 512 threads (16 warps)
  - 32 blocks / phase $\Rightarrow \lceil 98/32 = 3.0625 \rceil = 4$ phases
  - 1 warp scheduler/block $\Rightarrow$ each block takes 16s
  - Total time: $4 \times 16 =$ 64s
  - Last phase has 1 full block plus a block with 10.5 warps
  - The two blocks can be assigned to different SMs
  - 4 warp schedulers -> 8 warps/warp scheduler -> 8s plus some wait time, say total of 12s for last phase $\Rightarrow$ final total: 60s

GTX-1080Ti

```
Device 0: "GeForce GTX 1080 Ti"
CUDA Driver Version / Runtime Version          9.1 / 9.1
CUDA Capability Major/Minor version number:    6.1
Total amount of global memory:                 11172 MBytes (11715084288 bytes)
(28) Multiprocessors, (128) CUDA Cores/MP:     3584 CUDA Cores
GPU Max Clock rate:                            1683 MHz (1.68 GHz)
Memory Clock rate:                             5505 Mhz
Memory Bus Width:                              352-bit
L2 Cache Size:                                 2883584 bytes
Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
Total amount of constant memory:               65536 bytes
Total amount of shared memory per block:       49152 bytes
Total number of registers available per block: 65536
Warp size:                                     32
Maximum number of threads per multiprocessor:  2048
Maximum number of threads per block:           1024
Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
Maximum memory pitch:                          2147483647 bytes
Texture alignment:                             512 bytes
Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
Run time limit on kernels:                     No
Integrated GPU sharing Host Memory:            No
Support host page-locked memory mapping:       Yes
Alignment requirement for Surfaces:            Yes
Device has ECC support:                        Disabled
Device supports Unified Addressing (UVA):      Yes
Supports Cooperative Kernel Launch:            Yes
Supports MultiDevice Co-op Kernel Launch:      Yes
Device PCI Domain ID / Bus ID / location ID:   0 / 23 / 0
Compute Mode: < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
```

GTX1080Ti Computation

Critical details of the GTX-1080Ti:

- Pascal micro-architecture
  - c.f.: `https://docs.nvidia.com/cuda/pascal-tuning-guide/index.html`
  - 4 warp schedulers/SM
  - Each scheduler can issue up to 2 (independent) instructions from the same warp each cycle
- 28 SMs, 128 cores/SM (= 1024 cores in total)
- Maximum 2048 threads/SM
- Maximum 1024 threads/block
- GPU maximum clock rate: 1.68GHz

GTX1080Ti Computation cntd.

- Maximum number of blocks/SM is 64
- so if you have less than 32 threads/block you can not fill the full SM thread limit, which may lead to under occupancy
- but you shouldn't really have less than 32 threads per block anyway because then you are underusing warps.
- Thus in the theoretical optimal case, if you have a kernel with $32 \times N$ threads in total, and the kernel is K instructions long, you could potentially execute the kernel in $\frac{N \times K}{28 \times 4}$ cycles at maximum 1.68 GHz
  - = approx 4.56 times the speed of the GTX 960

GTX1080Ti Computation cntd.

- Maximum number of blocks/SM is 64
- so if you have less than 32 threads/block you can not fill the full SM thread limit, which may lead to under occupancy
- but you shouldn't really have less than 32 threads per block anyway because then you are underusing warps.
- Thus in the theoretical optimal case, if you have a kernel with $32 \times N$ threads in total, and the kernel is K instructions long, you could potentially execute the kernel in $\frac{N \times K}{28 \times 4}$ cycles at maximum 1.68 GHz
  - = approx 4.56 times the speed of the GTX 960
  - Single precision GLOPS rating of GTX960 = 2308
  - Single precision GLOPS rating of GTX1080Ti = 10609
  - Ratio = approx 4.60

Synchronisation

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 ⇒ *NO DIVERGENCE*

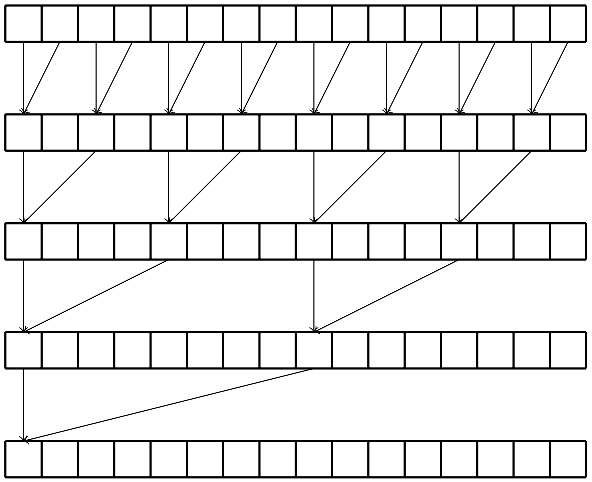
Loops where different threads in the warp execute different numbers of iterations also form divergences: The threads that execute the fewest number of iterations wait on the threads that execute the most number.

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] ;
}
```

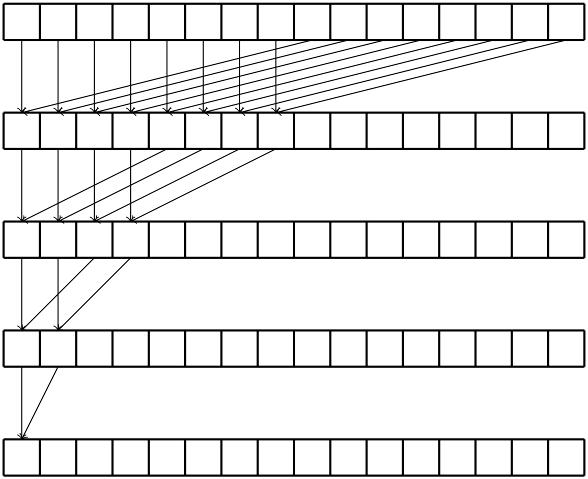
- Note the `__syncthreads()`: necessary to make sure all threads have completed previous stage.

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, 1024 threads = 32 warps all active, but only half of those threads are doing useful work and all of them are taking two passes
- 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 > 0; stride >> 1)
{
    __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 > 0; stride >> 1)
{
    __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 > 0; stride >> 1)
{
    __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

CUDA Memories

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 band width (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	`__device__ __shared__ int var;`
Global	`__device__ int var;`
Constant	`__device__ __constant__ int var;`

Parallel Reduction and Memory

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 *tiled* 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.