Synchronization

Ideal case for parallelism:

- no resources shared between threads
- no communication between threads

Many algorithms that require just a little bit of resource sharing can still be accelerated by massive parallelism of GPU
Examples needing synchronization

(1) Block loads data into shared memory before processing
(2) Summing a list of numbers
__syncthreads() synchronizes all threads in a block.

Useful for “load data into shared memory example”

No global equivalent of __syncthreads()
Atomic instructions: motivation

Two threads try to increment variable $x=42$ concurrently. Final value should be 44.

Possible execution order:

thread 0 load $x$ (=42) into register r0
thread 1 load $x$ (=42) into register r1
thread 0 increment r0 to 43
thread 1 increment r1 to 43
thread 0 store r0 (=43) into x
thread 1 store r1 (=43) into x

Actual final value of $x$: 43 :(
An atomic instruction executes as a single unit, cannot be interrupted.
Atomic instructions on CUDA

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

Syntax: atomicAdd(float *address, float val)

Work in both global and shared memory!
The fun world of parallelism

All of the atomic instructions can be implemented given 
**compare and swap**:

```c
atomicCAS(int *address, int compare, int val)
```

CAS is very powerful, can also implement locks, lock-free data structures, etc.

Recommend **Art of Multiprocessor Programming** to learn more
What if I only need to synchronize between all threads in a warp?
Warps are already synchronized!

Can save unneeded __syncthreads() use, but code is fragile and can be broken by compiler optimizations.
Warp vote & warp shuffle

Safer warp-synchronous programming (and doesn’t use shared memory)

Warp vote: \texttt{\_all}, \texttt{\_any}, \texttt{\_ballot}

\begin{verbatim}
int x = threadIdx.x; // goes from 0 to 31
\_any(x < 16) == true;
\_all(x < 16) == false;
\_ballot(x < 16) == (1 << 16) - 1;
\end{verbatim}
Warp shuffle

Read value of register from another thread in warp.

```
int __shfl(int var, int srcLane, int width=warpSize)
```

Extremely useful to compute sum of values across a warp (and other reductions, more next time)

First available on Kepler (no Fermi, only CC >= 3.0)
Do more cheap things and fewer expensive things!

Example: computing sum of list of numbers

Naive:
each thread atomically increments each number to accumulator in global memory
Smarter solution:

- each thread computes its own sum in register
- use warp shuffle to compute sum over warp
- each warp does a single atomic increment to accumulator in global memory
(1) Questions on latency hiding, thread divergence, coalesced memory access, bank conflicts, instruction dependencies

(2) Putting it into action: optimizing matrix transpose. Need to comment on all non-coalesced memory accesses and bank conflicts in code.
Matrix transpose

A great IO problem, because you have a stride 1 access and a stride n access.

Transpose is just a fancy memcpy, so memcpy provides a great performance target.
Matrix Transpose

```c
__global__
void naiveTransposeKernel(const float *input, float *output, int n) {
    // launched with (64, 16) block size and (n / 64, n / 64) grid size
    // each block transposes a 64x64 block

    const int i = threadIdx.x + 64 * blockIdx.x;
    int j = 4 * threadIdx.y + 64 * blockIdx.y;
    const int end_j = j + 4;

    for (; j < end_j; j++) {
        output[j + n * i] = input[i + n * j];
    }
}
```
Shared memory & matrix transpose

Idea to avoid non-coalesced accesses:

- Load from global memory with stride 1
- Store into shared memory with stride $x$
- __syncthreads()
- Load from shared memory with stride $y$
- Store to global memory with stride 1

Choose values of $x$ and $y$ perform the transpose.
Avoiding bank conflicts

You can choose $x$ and $y$ to avoid bank conflicts.

A stride $n$ access to shared memory avoids bank conflicts iff $\gcd(n, 32) = 1$. 
Two versions of the same kernel

You have to write 2 kernels for the set:

(1) `shmemTransposeKernel`. This should have all of the optimizations with memory access I just talked about.

(2) `optimalTransposeKernel`. Build on top of `shmemTransposeKernel`, but include any optimizations tricks that you want.
Possible optimizations

- Reduce & separate instruction dependencies (and everything depends on writes)
- Unroll loops to reduce bounds checking overhead
- Try rewriting your code to use 64-bit or 128-bit loads (with float2 or float4)
- Take a warp-centric approach rather than block-centric and use warp shuffle rather than shared memory (will not be built on top of shmemTranspose). I’ll allow you to use this as a guide
Profiling

profiling = analyzing where program spends time
Putting effort into optimizing without profiling is foolish.

There is a great visual (GUI) profiler for CUDA called nvpp, but using it is a bit of a pain with a remote GPU.

nvprof is the command line profiler (works on minuteman) so let’s check that out!
If you didn’t catch the demo in class (or even if you did), this blog post and the guide will be useful.
Profiler tips

List all profiler events with `nvprof --query-events` and all metrics with `nvprof --query-metrics`.

Many useful metrics, some good ones are `acheived_occupancy`, `ipc`, `shared_replay_overhead`, all of the utilizations, all of the throughputs.

Some useful events are `global_ld_mem_divergence_replays`, `global_st_mem_divergence_replays`, `shared_load_replay`, `shared_store_replay`
Profile example (1)

[emartin@minuteman:~/set2]$ nvprof --events
global_ld_mem_divergence_replays,global_st_mem_divergence_replays --metrics achieved_occupancy ./transpose 4096 naive

==11043== NVPROF is profiling process 11043, command: ./transpose 4096 naive

==11043== Warning: Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.

Size 4096 naive GPU: 33.305279 ms

==11043== Profiling application: ./transpose 4096 naive

==11043== Profiling result:

==11043== Event result:
<table>
<thead>
<tr>
<th>Invocations</th>
<th>Event Name</th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device &quot;GeForce GTX 780 (0)&quot;</td>
<td>Kernel: naiveTransposeKernel(float const <em>, float</em>, int)</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>global_ld_mem_divergence_replays</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>global_st_mem_divergence_replays</td>
<td>16252928</td>
<td>16252928</td>
<td>16252928</td>
</tr>
</tbody>
</table>

==11043== Metric result:

<table>
<thead>
<tr>
<th>Invocations</th>
<th>Metric Name</th>
<th>Metric Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device &quot;GeForce GTX 780 (0)&quot;</td>
<td>achieved_occupancy</td>
<td>Achieved Occupancy</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.862066</td>
<td>0.862066</td>
<td>0.862066</td>
</tr>
</tbody>
</table>
Profiling interpretation

Lots of non-coalesced stores, 83% occupancy for naive kernel transpose
Submit jobs with qsub. You must specify -l gpu=1 for cuda

> qsub -l gpu=1 job.sh

Binaries can be run directly if the binary option is specified

> qsub -l gpu=1 -b y ./program

Script and binaries are run in your homedir by default. Use the -cwd option to run in the current folder

~/.set10/files/>qsub -l gpu=1 -cwd job.sh
Amazon Cluster (2)

View running jobs with qstat

```
>qstat

The stdout and stderr are in stored in files after the job completes. These files have the job number appended.

>qsub -l gpu=1 -cwd job.sh

>ls

job.sh.o5

job.sh.e5
```
Login requires ssh keys

```bash
>ssh -i user123.rsa user123@54.163.37.252
```

Can also use the url instead of the ip

`ec2-54-163-37-252.compute-1.amazonaws.com`

Windows users must use puttygen to convert to putty format

Keys will be distributed to each haru account