CS 179: GPU Programming

LECTURE 5: GPU COMPUTE ARCHITECTURE
Last time...

GPU Memory System
- Different kinds of memory pools, caches, etc
- Different optimization techniques
Warp Schedulers

Warp schedulers find a warp that is ready to execute its next instruction and available execution cores and then start execution

- GK110: 4 warp schedulers, 2 dispatchers in each SM
- Starts instructions in up to 4 warps each clock,
- and starts up to 2 instructions in each warp.
GK110 (Kepler) numbers

- max threads / SM = 2048 (64 warps)
- max threads / block = 1024 (32 warps)
- 32 bit registers / SM = 64k
- max shared memory / SM = 48KB

The number of blocks that run concurrently on a SM depends on the resource requirements of the block!
Occupancy

occupancy = warps per SM / max warps per SM

max warps / SM depends only on GPU

warps / SM depends on warps / block, registers / block, shared memory / block.
<table>
<thead>
<tr>
<th><strong>GK110 Occupancy</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>100% occupancy</strong></td>
</tr>
<tr>
<td>● 2 blocks of 1024 threads</td>
</tr>
<tr>
<td>● 32 registers/thread</td>
</tr>
<tr>
<td>● 24KB of shared memory / block</td>
</tr>
<tr>
<td><strong>50% occupancy</strong></td>
</tr>
<tr>
<td>● 1 block of 1024 threads</td>
</tr>
<tr>
<td>● 64 registers/thread</td>
</tr>
<tr>
<td>● 48KB of shared memory / block</td>
</tr>
</tbody>
</table>
This lecture

- Synchronization
- Atomic Operations
- Instruction Dependencies
- Instruction Level Parallelism (ILP)
Synchronization

**Synchronization** is a process by which multiple threads must indirectly communicate with each other in order to make sure they do not clash with each other.

- Example of a synchronization issue:
  - `int x = 1;`
  - Thread 1 wants to add 1 to `x;`
  - Thread 2 wants to add 1 to `x;`
  - Thread 1 reads in the value of `x` (which is 1) into a register
  - Thread 2 reads in the value of `x` (which is still 1) into a register
  - Both threads increment the values they read in but they both think the final value is 2
  - They write `x` back out and the final result is 2
Synchronization

On a CPU, you can solve synchronization issues using Locks, Semaphores, Condition Variables, etc. On a GPU, these solutions introduce too much memory and process overhead.

- We have simpler solutions better suited for parallel programs
CUDA Synchronization

Use the __syncthreads() function to sync threads within a block
- Only works at the block level
- SMs are separate from each other so can't do better than this
- Similar to barrier() function in C/C++
Atomic Operations

Atomic Operations are operations that ONLY happen in sequence
- For example, if you want to add up N numbers by adding the numbers to a variable that starts in 0, you must add one number at a time
- Don't do this though. We'll talk about better ways to do this in the next lecture. Only use when you have no other options

CUDA provides built in atomic operations
- Use the functions: atomic<op>(float *address, float val);
  - Replace <op> with one of: Add, Sub, Exch, Min, Max, Inc, Dec, And, Or, Xor
  - e.g. atomicAdd(float *address, float val) for atomic addition
  - These functions are all implemented using a function called atomicCAS(int *address, int compare, int val)
  - CAS stands for compare and swap. The function compares *address to compare and swaps the value to val if the values are different
Instruction Dependencies

An **Instruction Dependency** is a requirement relationship between instructions that force a sequential execution
- In the example on the right, each summation call must happen in sequence because the value of `acc` depends on the previous summation as well
Can be caused by direct dependencies or requirements set by the execution order of code
- I.e. You can't start an instruction until all previous operations have been completed in a single thread

```plaintext
acc += x[0];
acc += x[1];
acc += x[2];
acc += x[3];
...
```
Instruction Level Parallelism (ILP)

**Instruction Level Parallelism** is when you avoid performances losses caused by instruction dependencies

- In CUDA, also removes performances losses caused by how certain operations are handled by the hardware
ILP Example

\[
\begin{align*}
  z_0 &= x[0] + y[0]; \\
  z_1 &= x[1] + y[1]; \\
  x_0 &= x[0]; \\
  y_0 &= y[0]; \\
  z_0 &= x_0 + y_0; \\
  x_1 &= x[1]; \\
  y_1 &= y[1]; \\
  z_1 &= x_1 + y_1;
\end{align*}
\]

- The second half of the code can't start execution until the first half completes.
ILP Example

\[ z_0 = x[0] + y[0]; \]
\[ z_1 = x[1] + y[1]; \]

\[ x_0 = x[0]; \]
\[ y_0 = y[0]; \]
\[ x_1 = x[1]; \]
\[ y_1 = y[1]; \]
\[ z_0 = x_0 + y_0; \]
\[ z_1 = x_1 + y_1; \]

- Sequential nature of the code due to instruction dependency has been minimized.
- Additionally, this code minimizes the number of memory transactions required.
Questions?

- Synchronization
- Atomic Operations
- Instruction Dependencies
- Instruction Level Parallelism (ILP)
Next time...

Set 2 Rec on Friday (04/06)
GPU based algorithms (next week)