CS 179: GPU Programming

LECTURE 5: SYNCHRONIZATION AND ILP

Announcement

Because of the large number of students enrolled in this course, we are changing how to submit labs for this course. Instead of emailing us with a zip file of your README and code, we are going to utilize Titan for set submission.

Instead of emailing us the solution, put a zip file in your home directory on Titan, in the format: **lab[N]_2019_submission.zip**

Your submission should be a single archive file (.zip) with your README file and all code.

Even if you've already emailed us with your lab 1 submission, you still need to put the zip file on Titan.

Last time…

- GPU Memory System
  - Global Memory: the slowest and largest form of memory on the GPU, shared by all grids
    - Coalesced memory access minimizes the number of cache lines read
  - Shared Memory: very fast memory, located on the SM and shared by the block
    - Setup as 32 banks that can be accessed in parallel. Each successive 32-bit words are assigned to successive banks
    - A bank conflict occurs when 2 threads in a warp access different elements in the same bank
  - Registers: fasted memory possible, located on the SM, scope is the thread
  - Local Memory: located on the Global Memory, scope is the thread
  - L1/L2/L3 Cache
  - Texture and Constant Cache

This lecture

- Synchronization
- Atomic Operations
- Instruction Dependencies
- Instruction Level Parallelism (ILP)
- Warp Scheduler
- Occupancy

Synchronization

Ideal case for parallelism:

- no resources shared between threads
- no communication needed between threads

However, many algorithms that require shared resources can still be accelerated by massive parallelism of the GPU.

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 synchronization issue:

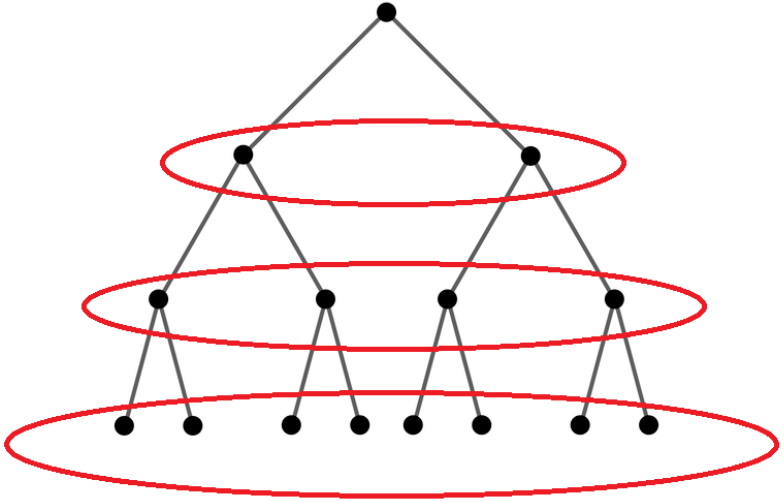
```
int x = 1;
Thread 1: x += 1;
Thread 2: x += 1;
```

- 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

Examples needing synchronization:

- Parallel BFS
- Summing a list of numbers
- Loading data into a GPU's shared memory



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++
- This __synchthreads() call is very useful for kernels using shared memory.

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

Atomic Operations

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

```
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

- Idea: we do not have to wait until instruction *n* has finished to start instruction *n + 1*
- In CUDA, also removes performances losses caused by how certain operations are handled by the hardware

ILP Example

```
z0 = x[0] + y[0];
z1 = x[1] + y[1];
```


```
x0 = x[0];
y0 = y[0];
z0 = x0 + y0;

x1 = x[1];
y1 = y[1];
z1 = x1 + y1;
```

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

ILP Example

```
z0 = x[0] + y[0];
z1 = x[1] + y[1];
```


```
x0 = x[0];
y0 = y[0];
x1 = x[1];
y1 = y[1];
z0 = x0 + y0;
z1 = x1 + y1;
```

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

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 in each SM and 2 dispatchers in each scheduler
  - Can start instructions in up to 4 warps each clock and up to 2 **subsequent, independent** instructions in each warp.

Occupancy

Idea: Need enough independent threads per SM to hide latencies

- Instruction latencies
- Memory access latencies

Occupancy: number of concurrent threads per SM

- Occupancy = active warps per SM / max warps per SM

Number of threads that fit per SM (max warps per SM) is determined by the hardware resources of the GPU.

Occupancy

The number of active warps per SM is determined by the limiting resources

- Registers per thread
  - SM registers are partitioned among the threads
- Shared memory per thread block
  - SM shared memory is partitioned among the blocks
- Threads per thread block
  - Threads are allocated at thread block granularity

Needed occupancy depends on the code

- More independent work per thread -> less occupancy is needed
- Memory-bound codes tend to need more occupancy Higher latency than for arithmetic, need more work to hide it

Don’t need for 100% occupancy for maximum performance

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!

GK110 Occupancy

100% occupancy

- 2 blocks of 1024 threads
- 32 registers/thread
- 24KB of shared memory / block

50% occupancy

- 1 block of 1024 threads
- 64 registers/thread
- 48KB of shared memory / block

Questions?

- Synchronization
- Atomic Operations
- Instruction Dependencies
- Instruction Level Parallelism (ILP)
- Warp Scheduler
- Occupancy

Resources

ILP
https://www.nvidia.com/content/GTC-2010/pdfs/2238_GTC2010.pdf

Warps and Occupancy
http://on-demand.gputechconf.com/gtc-express/2011/presentations/cuda_webinars_WarpsAndOccupancy.pdf

Next time…

Set 2 is out today and will be due next Wednesday (04/17)
Set 2 Recitation on Friday (04/12)
GPU based algorithms (next week, lectures will be given by George?)