Objective

- To learn practical histogram programming techniques
  - Basic histogram algorithm using atomic operations
  - Atomic operation throughput
  - Privatization

Review: A Histogram Example

- In phrase “Programming Massively Parallel Processors” build a histogram of frequencies of each letter
  - A(4), C(1), E(1), G(1), …
  - How do you do this in parallel?
    - Have each thread to take a section of the input
    - For each input letter, use atomic operations to build the histogram

Iteration #1 – 1st letter in each section
Iteration #2 – 2\textsuperscript{nd} letter in each section

Iteration #3

Iteration #4

Iteration #5
What is wrong with the algorithm?

A better approach:
- Reads from the input array are not coalesced
  - Assign inputs to each thread in a strided pattern
  - Adjacent threads process adjacent input letters

Iteration 2
- All threads move to the next section of input

A Histogram Kernel
- The kernel receives a pointer to the input buffer
- Each thread processes the input in a strided pattern

```
__global__ void histo_kernel(unsigned char *buffer,
                           long size, unsigned int *histo)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    // stride is total number of threads
    int stride = blockDim.x * gridDim.x;
    // stride is total number of threads

    // stride is total number of threads
```
More on the Histogram Kernel

// All threads in the grid collectively handle
// blockDim.x * gridDim.x consecutive elements

while (i < size) {
    atomicAdd( &(histo[buffer[i]]), 1);
    i += stride;
}

Atomic Operations on DRAM

• An atomic operation starts with a read, with a latency of a few hundred cycles

• The atomic operation ends with a write, with a latency of a few hundred cycles

• During this whole time, no one else can access the location

• Each Load-Modify-Store has two full memory access delays
  – All atomic operations on the same variable (RAM location) are serialized
Latency determines throughput of atomic operations

- Throughput of an atomic operation is the rate at which the application can execute an atomic operation on a particular location.

- The rate is limited by the total latency of the read-modify-write sequence, typically more than 1000 cycles for global memory (DRAM) locations.

- This means that if many threads attempt to do atomic operation on the same location (contention), the memory bandwidth is reduced to < 1/1000!

You may have a similar experience in supermarket checkout

- Some customers realize that they missed an item after they started to check out
- They run to the isle and get the item while the line waits
  - The rate of check is reduced due to the long latency of running to the isle and back.
- Imagine a store where every customer starts the check out before they even fetch any of the items
  - The rate of the checkout will be 1 / (entire shopping time of each customer)

Hardware Improvements

- Atomic operations on Fermi L2 cache
  - medium latency, but still serialized
  - Global to all blocks
  - “Free improvement” on Global Memory atomics

Hardware Improvements (cont.)

- Atomic operations on Shared Memory
  - Very short latency, but still serialized
  - Private to each thread block
  - Need algorithm work by programmers (more later)
Atomics in Shared Memory Requires Privatization

- Create private copies of the histo[] array for each thread block

```c
__global__ void histo_kernel(unsigned char *buffer, long size, unsigned int *histo)
{
    __shared__ unsigned int histo_private[256];
    if (threadIdx.x < 256) histo_private[threadIdx.x] = 0;
    __syncthreads();

    int i = threadIdx.x + blockIdx.x * blockDim.x;
    // stride is total number of threads
    int stride = blockDim.x * gridDim.x;
    while (i < size) {
        atomicAdd( &(private_histo[buffer[i]], 1);
        i += stride;
    }
}
```

Build Private Histogram

```c
int i = threadIdx.x + blockIdx.x * blockDim.x;
// stride is total number of threads
int stride = blockDim.x * gridDim.x;
while (i < size) {
    atomicAdd( &histo[threadIdx.x], private_histo[threadIdx.x] );
    i += stride;
}
```

More on Privatization

- Privatization is a powerful and frequently used techniques for parallelizing applications

  - The operation needs to be associative and commutative
    - Histogram add operation is associative and commutative
  - The histogram size needs to be small
    - Fits into shared memory
  - What if the histogram is too large to privatize?
ANY MORE QUESTIONS?
READ CHAPTER 9