ECE408 Spring 2019

Applied Parallel Programming

Lecture 17: Atomic Operations and Histogramming - Part 2

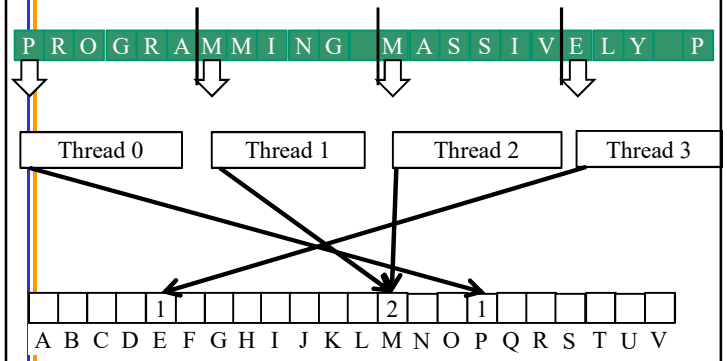
Objective

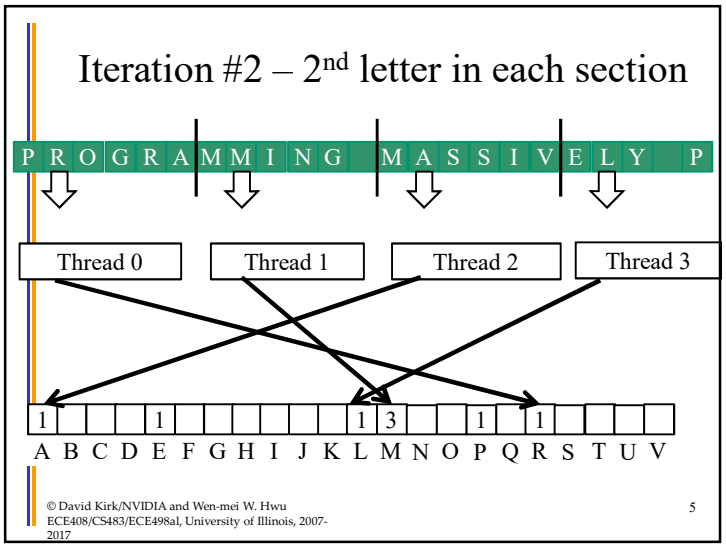
- To learn practical histogram programming techniques
  - Basic histogram algorithm using atomic operations
  - Atomic operation throughput
  - Privatization

Review: A Histogram Example

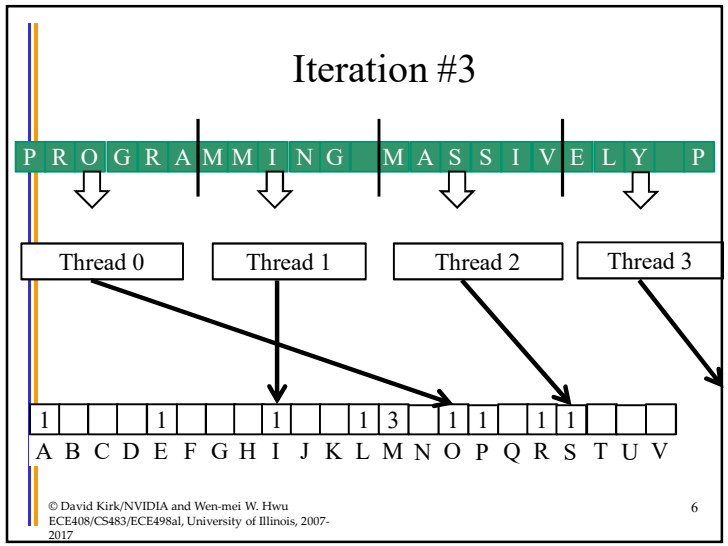
- In phrase "Programming Massively Parallel Processors" build a histogram of frequencies of each letter
- A(4), C(1), E(1), G(1), …
- How do you do this in parallel?
  - Have each thread to take a section of the input
  - For each input letter, use atomic operations to build the histogram

Iteration #1 – 1st letter in each section

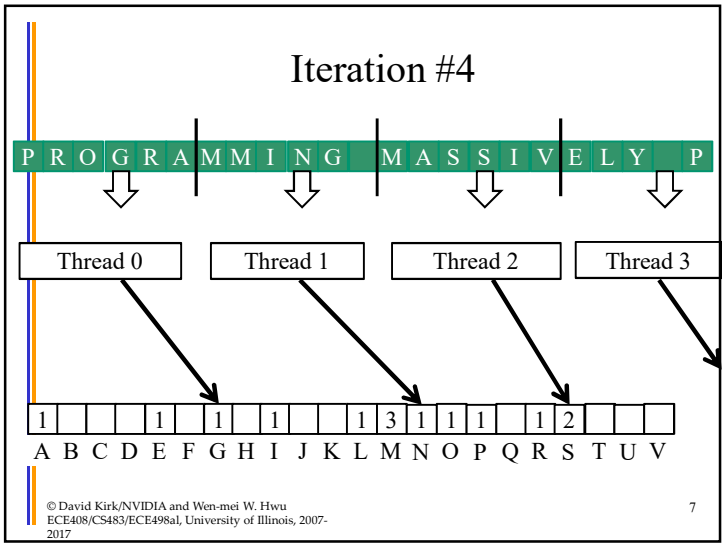



Iteration #2 – 2nd letter in each section
P R O G R A M M I N G M A S S I V E L Y P
Thread 0
Thread 1
Thread 2
Thread 3
1 1 1 3 1 1
A B C D E F G H I J K L M N O P Q R S T U V
© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al, University of Illinois, 2007-2017
5

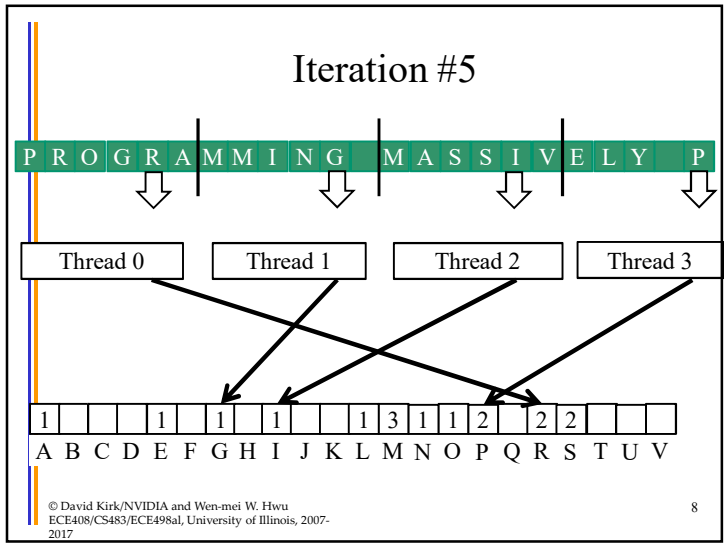
5


Iteration #3
P R O G R A M M I N G M A S S I V E L Y P
Thread 0
Thread 1
Thread 2
Thread 3
1 1 1 1 3 1 1 1 1
A B C D E F G H I J K L M N O P Q R S T U V
© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al, University of Illinois, 2007-2017
6

6


Iteration #4
P R O G R A M M I N G M A S S I V E L Y P
Thread 0
Thread 1
Thread 2
Thread 3
1 1 1 1 1 3 1 1 1 1 2
A B C D E F G H I J K L M N O P Q R S T U V
© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al, University of Illinois, 2007-2017
7

7

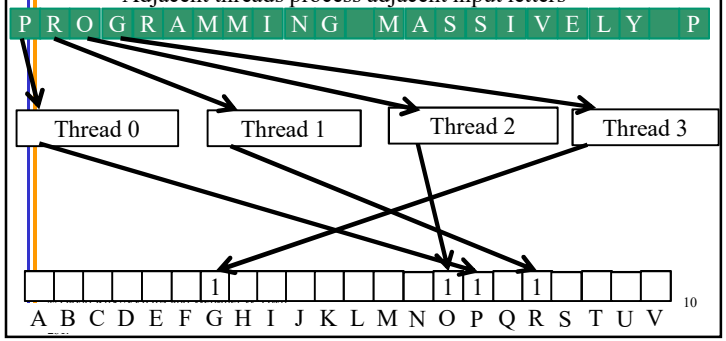

Iteration #5
P R O G R A M M I N G M A S S I V E L Y P
Thread 0
Thread 1
Thread 2
Thread 3
1 1 1 1 1 3 1 1 2 2 2
A B C D E F G H I J K L M N O P Q R S T U V
© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al, University of Illinois, 2007-2017
8

8

What is wrong with the algorithm?

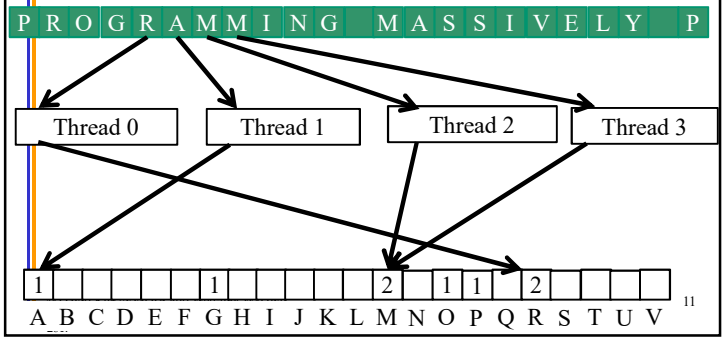
What is wrong with the algorithm? A better approach:

- 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

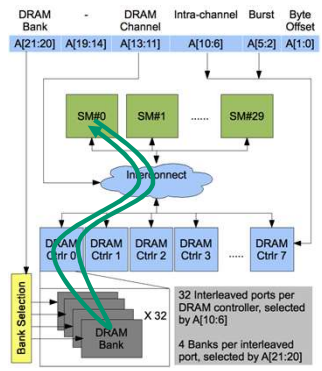
- The kernel receives a pointer to the input buffer
- Each thread process the input in a strided pattern

```
__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;
```

More on the Histogram Kernel

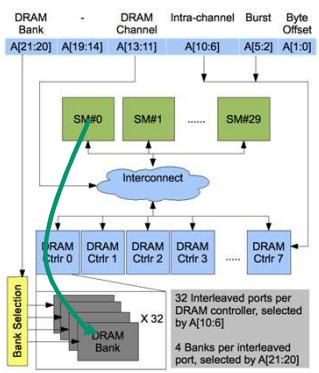
```
// All threads in the grid collectively handle
// blockDim.x * gridDim.x consecutive elements

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

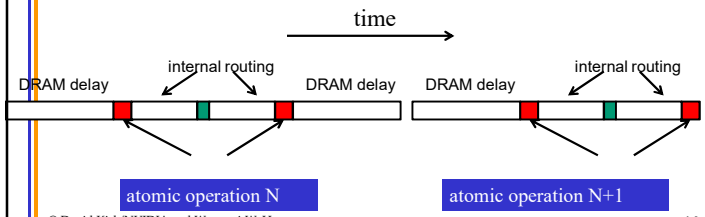
Atomic Operations on DRAM



- 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

Atomic Operations on DRAM

- 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

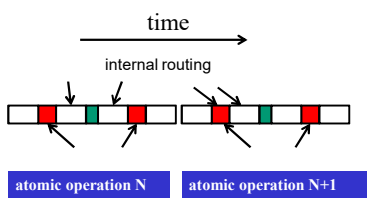
- Throughput of an atomic operation is the rate at which the application can execute an atomic operation on a particular location.
- The rate is limited by the total latency of the read-modify-write sequence, typically more than 1000 cycles for global memory (DRAM) locations.
- This means that if many 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)

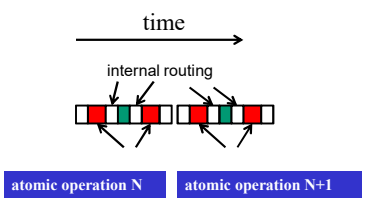
Hardware Improvements (cont.)

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



Hardware Improvements

- 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

```
__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();
```


Build Private Histogram

```
  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 Final Histogram

```
// wait for all other threads in the block to finish
__syncthreads();

if (threadIdx.x < 256)
  atomicAdd( &(histo[threadIdx.x]),
                        private_histo[threadIdx.x] );

}
```


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