ECE408 Spring 2020

Applied Parallel Programming

Lecture 17
Atomic Operations and Histogramming

1

Objective

- To understand atomic operations
  - Read-modify-write in parallel computation
  - A primitive form of "critical regions" in parallel programs
  - Use of atomic operations in CUDA
  - Why atomic operations reduce memory system throughput
  - How to avoid atomic operations in some parallel algorithms
- To learn practical histogram programming techniques
  - Basic histogram algorithm using atomic operations
  - Atomic operation throughput
  - Privatization

2

A Common Collaboration Pattern

- Multiple bank tellers count the total amount of cash in the safe
- Each grab a pile and count
- Have a central display of the running total
- Whenever someone finishes counting a pile, add the subtotal of the pile to the running total
- A bad outcome
  - Some of the piles were not accounted for

3

A Common Arbitration Pattern

- Multiple customers booking air tickets
- Each
  - Brings up a flight seat map
  - Decides on a seat
  - Update the the seat map, mark the seat as taken
- A bad outcome
  - Multiple passengers ended up booking the same seat

4

Read-Modify-Write Operations

thread1: Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

thread2: Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

If Mem[x] was **initially 0**, what would the value of Mem[x] be after threads 1 and 2 have completed?

– What does each thread get in their Old variable?

The answer may vary due to data races. To avoid data races, you should use atomic operations

5

Timing Scenario #1

Time	Thread 1	Thread 2
1	(0) Old ← Mem[x]	
2	(1) New ← Old + 1	
3	(1) Mem[x] ← New	
4		(1) Old ← Mem[x]
5		(2) New ← Old + 1
6		(2) Mem[x] ← New

- Thread 1 Old = 0
- Thread 2 Old = 1
- Mem[x] = 2 after the sequence

6

Timing Scenario #2

Time	Thread 1	Thread 2
1		(0) Old ← Mem[x]
2		(1) New ← Old + 1
3		(1) Mem[x] ← New
4	(1) Old ← Mem[x]	
5	(2) New ← Old + 1	
6	(2) Mem[x] ← New	

- Thread 1 Old = 1
- Thread 2 Old = 0
- Mem[x] = 2 after the sequence

7

Timing Scenario #3

Time	Thread 1	Thread 2
1	(0) Old ← Mem[x]	
2	(1) New ← Old + 1	
3		(0) Old ← Mem[x]
4	(1) Mem[x] ← New	
5		(1) New ← Old + 1
6		(1) Mem[x] ← New

- Thread 1 Old = 0
- Thread 2 Old = 0
- Mem[x] = 1 after the sequence

8

Timing Scenario #4

Time	Thread 1	Thread 2
1		(0) Old ← Mem[x]
2		(1) New ← Old + 1
3	(0) Old ← Mem[x]	
4		(1) Mem[x] ← New
5	(1) New ← Old + 1	
6	(1) Mem[x] ← New	

- Thread 1 Old = 0
- Thread 2 Old = 0
- Mem[x] = 1 after the sequence

9

Atomic Operations Prevent Interleaving

thread1: Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

thread2: Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

Or

thread2: Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

thread1: Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

10

Without Atomic Operations

Mem[x] initialized to 0

thread1: Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

thread2: Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

- Both threads receive 0
- Mem[x] becomes 1

11

Needed When Threads Write to Same Location

When **two threads**

- **may write to** the **same memory** location,
- the program may **need atomic operations**.

Sharing is **not always easy to recognize**…

- Do two insertions into a hash table share data?
- What about two graph node updates based on all of the nodes' neighbors?
- What if nodes are on same side of bipartite graph?

12

What Exactly is "Atomic?"

To a high-energy photon, atoms are not.

Atomicity is ALWAYS with respect to something.

Two **sections of code**
- **that execute atomically with respect to one another**
- **appear** to the software **as though**
- the programs' **execution did not interleave** at all.

What Can Go Wrong?

Common failure mode:
- **Programmer *thinks* operations** are **independent**.
- Hasn't considered input data for which they are not.
- Or another programmer reuses code without understanding assumptions that imply independence.

Also: **atomicity does not constrain relative order**.

Implementing Atomic Operations

- Many ISAs offer synchronization primitives,
  - **instructions with** one (or more) **address operands**
  - **that execute atomically with respect to one another** when used on the same address.
- Mostly read, modify, write operations
  - Bit test and set
  - Compare and swap / exchange
  - Swap / exchange
  - Fetch and increment / add

Atomicity Enforced by Microarchitecture

When synchronization primitives execute,
- **hardware ensures** that **no other** thread
- **accesses** the location **until** the operation is **complete**.

Other threads that access the location
- are typically stalled or held in a queue until their turn.
- **Threads perform atomic operations serially.**

Atomic Operations in CUDA

- Function calls that are translated into single ISA instructions (a.k.a. *intrinsics*)
  - Atomic add, sub, inc, dec, min, max, exch (exchange), CAS (compare and swap)
  - Read CUDA C programming Guide for more details
- Atomic Add

  int atomicAdd(int **address**, int **val**);*

  reads the 32-bit word **old** pointed to by **address** in global or shared memory, computes **(old + val)**, and stores the result back to memory at the same address. The function returns **old**.

More Atomic Adds in CUDA

- Unsigned 32-bit integer atomic add

  unsigned int atomicAdd(unsigned int address, unsigned int val);*
- Unsigned 64-bit integer atomic add

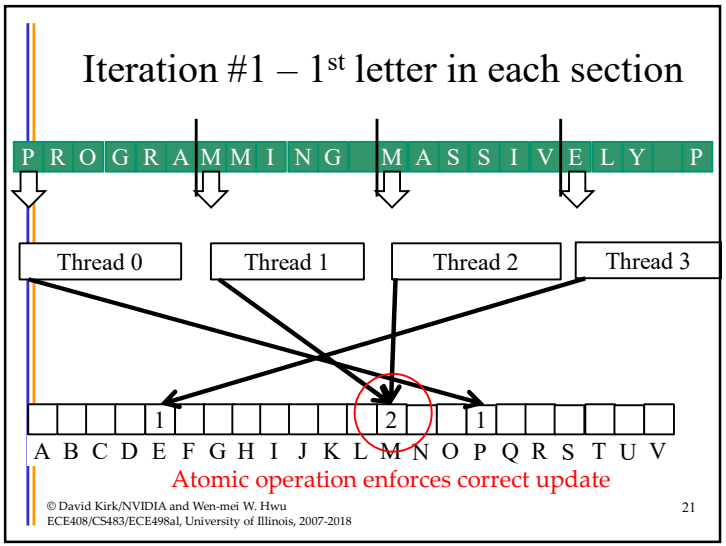
  unsigned long long int atomicAdd(unsigned long long int address, unsigned long long int val);*
- Single-precision floating-point atomic add (capability > 2.0)
  - float atomicAdd(float* address, float val);

Histogramming

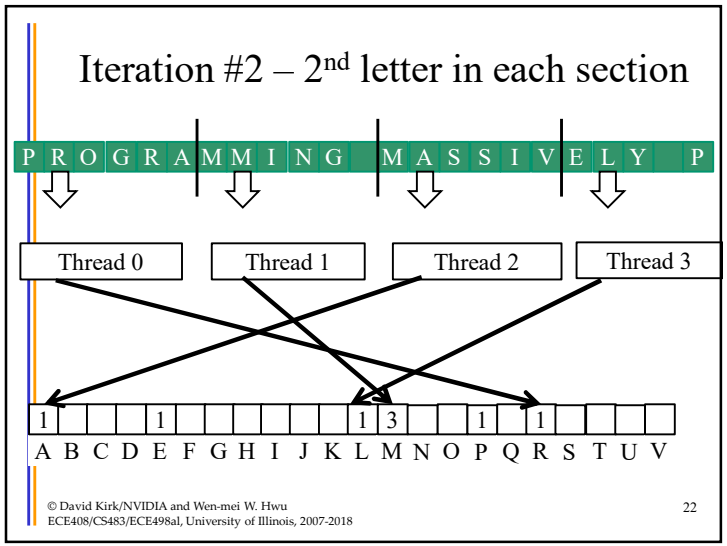
- A method for extracting notable features and patterns from large data sets
  - Feature extraction for object recognition in images
  - Fraud detection in credit card transactions
  - Correlating heavenly object movements in astrophysics
  - …
- Basic histograms - for each element in the data set, use the value to identify a "bin" to increment

A Histogram Example

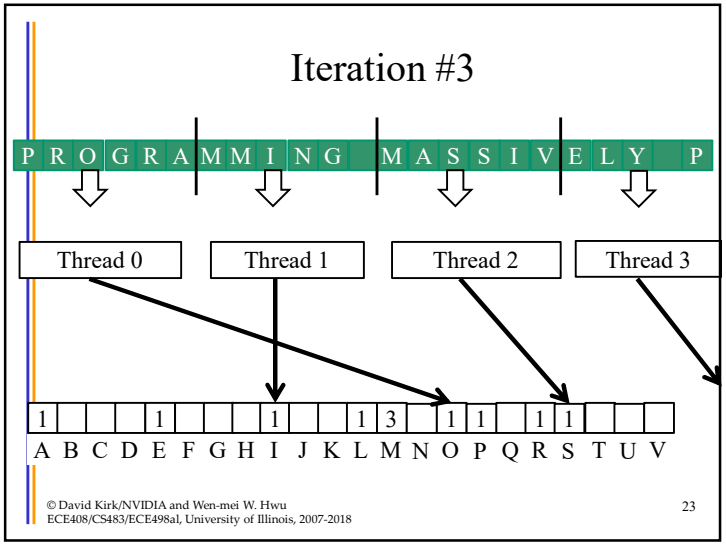
- In sentence "Programming Massively Parallel Processors" build a histogram of frequencies of each letter
- A(4), C(1), E(1), G(1), …
- How do you do this in parallel?


Iteration #1 – 1st 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 2 1
A B C D E F G H I J K L M N O P Q R S T U V
Atomic operation enforces correct update
© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al, University of Illinois, 2007-2018
21

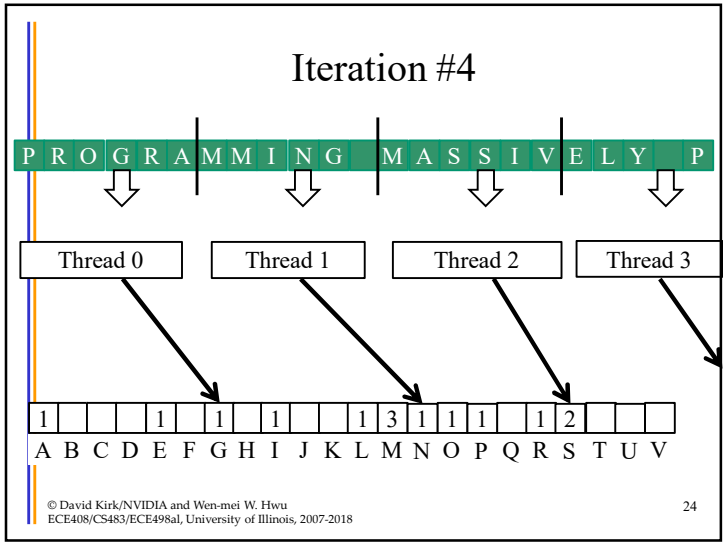
21


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, 2007-2018
22

22

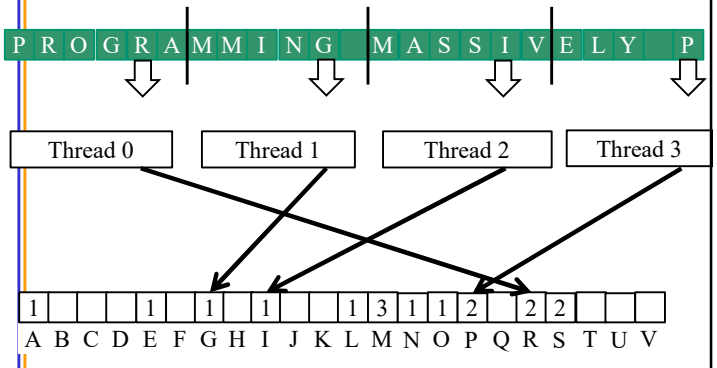

Iteration #3
P R O G R A M M I N G M A S S I V E L Y P
Thread 0
Thread 1
Thread 2
Thread 3
1 1 1 1 3 1 1 1 1
A B C D E F G H I J K L M N O P Q R S T U V
© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al, University of Illinois, 2007-2018
23

23


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 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, 2007-2018
24

24

Iteration #5



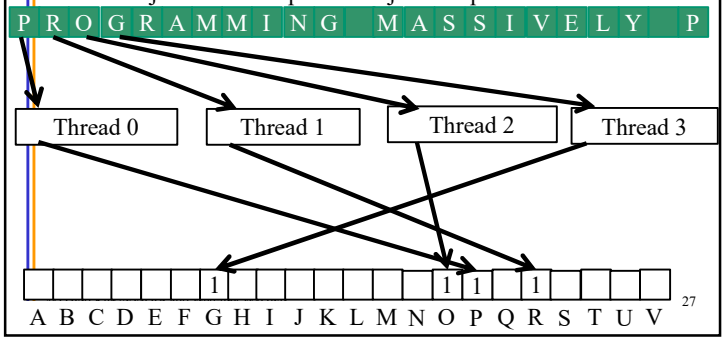
25

What is wrong with the algorithm?

26

What is wrong with the algorithm? A better approach:

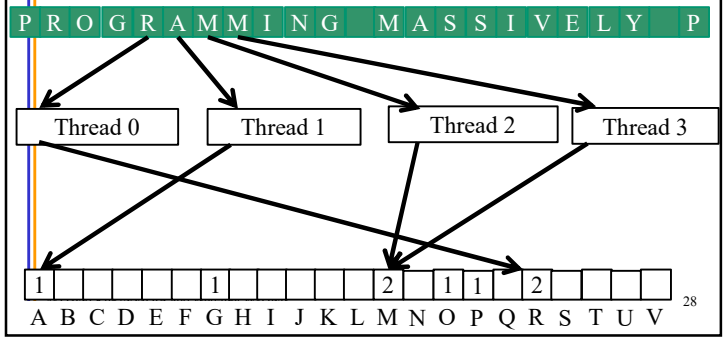
- Reads from the input array are not coalesced
  - Assign inputs to each thread in a strided pattern
  - Adjacent threads process adjacent input letters



27

Iteration 2

- All threads move to the next section of input



28

A Histogram Kernel

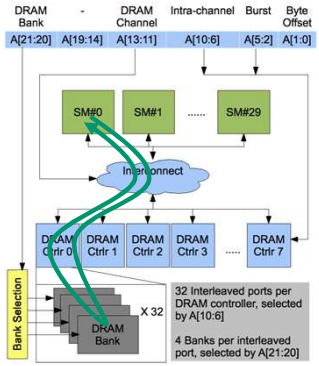
- The kernel receives a pointer to the input buffer
- Each thread process the input in a strided pattern

```
__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

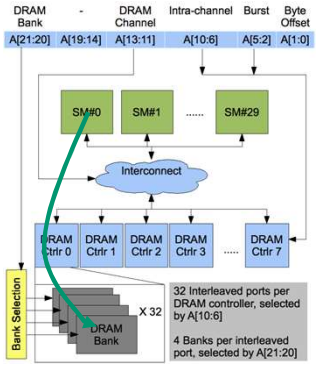
```
// All threads in the grid collectively handle
// blockDim.x * gridDim.x consecutive elements

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

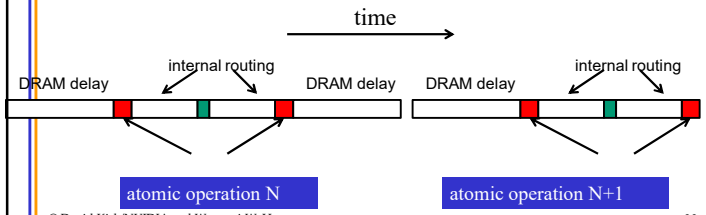
Atomic Operations on DRAM



- 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



33

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!

34

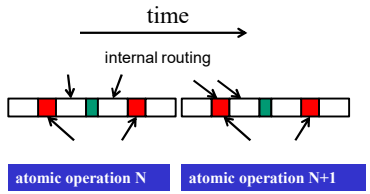
You may have a similar experience in supermarket checkout

- 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)

35

Hardware Improvements

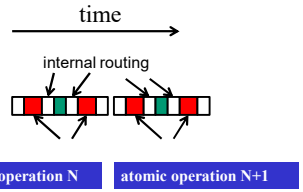
- Atomic operations on L2 cache
  - medium latency, but still serialized
  - Global to all blocks
  - "Free improvement" on Global Memory atomics



36

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)



37

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();
```


38

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;
    }
```


39

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] );

}
```


40

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?

41

ANY MORE QUESTIONS READ CHAPTER 9

42