Atomic Operations and Applications
Objectives

- Understand atomic operations
  - Read-modify-write in parallel computation
  - Use of atomic operations in CUDA
  - Why atomic operations reduce memory system throughput
- Histogramming as an example application of atomic operations
  - Basic histogram algorithm
  - Privatization
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.
A Common Parallel Coordination Pattern

- Multiple customer service agents serving customers
  - Each customer gets a number
  - A central display shows the number of the next customer who will be served
  - When an agent becomes available, he/she calls the number and he/she adds 1 to the display

- Bad outcomes
  - Multiple customers get the same number
  - Multiple agents serve the same number
A Common Arbitration Pattern

- Multiple customers booking air tickets, each
  - Brings up a flight seat map
  - Decides on a seat
  - Update the seat map, mark the seat as taken

- A bad outcome
  - Multiple passengers ended up booking the same seat
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.
Bad Timing

<table>
<thead>
<tr>
<th>Time</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(0) Old ← Mem[x]</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(1) New ← Old + 1</td>
<td>(0) Old ← Mem[x]</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>(1) Mem[x] ← New</td>
</tr>
<tr>
<td>4</td>
<td>(1) Mem[x] ← New</td>
<td>(1) New ← Old + 1</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>(1) Mem[x] ← New</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>(1) Mem[x] ← New</td>
</tr>
</tbody>
</table>

- Thread 1 Old = 0
- Thread 2 Old = 0
- Mem[x] = 1 after the sequence
Avoid Bad Timing: Atomic Operations

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

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

Or

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

thread2: Old ← Mem[x]
New ← Old + 1
Mem[x] ← New
Atomic Operation in General

- Performed by a single ISA instruction on a memory location *address*
  - Read the old value, modify the value, and write the new value to the location

- The hardware ensures that no other threads can access the location until the atomic operation is complete
  - Any other threads that access the location will typically be held in a queue until its turn
  - All threads perform the atomic operation *serially*
CUDA Atomic Functions

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

- For example: Atomic Add

  ```c
  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**

  ```c
  unsigned int atomicAdd(unsigned int* address, unsigned int val);
  ```

- **Unsigned 64-bit integer atomic add**

  ```c
  unsigned long long int atomicAdd(unsigned long long int* address, unsigned long long int val);
  ```

- **Single-precision floating-point atomic add (capability > 2.0)**

  ```c
  float atomicAdd(unsigned int* 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
In sentence “Advanced Parallel Computation” build a histogram of frequencies of each letter

Result: A(5), C(2), D(1), E(2), …

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
Example: Iteration 1

Advanced Parallel Computation

Thread 1  Thread 2  Thread 3  Thread 4

ABCDEFGHIJKLMNOPQRSTUVWXYZ

Copyright © 2013 by Yong Cao, Referencing UIUC ECE408/498AL Course Notes
Example: Iteration 2

ADVANCEDPARALLELCOMPUTATION

Thread 1

1 2 1

Thread 2

2 1

Thread 3

1

Thread 4

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 W X Y Z
Example: Iteration 3

Advanced Parallel Computation

Thread 1

Thread 2

Thread 3

Thread 4

1 2 2

2 1 2

1 1

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Copyright © 2013 by Yong Cao, Referencing UIUC ECE408/498AL Course Notes
Issue: None coalesced access

- Assign inputs to each thread in a strided pattern
- Adjacent threads process adjacent input letters
Solution: Coalesced access

- Iteration 1
Solution: Coalesced access

- Iteration 2
Solution: Coalesced access

- Iteration 3
A Histogram Kernel

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

```c
__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;
    // All threads handle blockDim.x * gridDim.x consecutive elements
    while (i < size) {
        atomicAdd( &(histo[buffer[i]]), 1);
        i += stride;
    }
}
```
Atomic Operation on Global Memory

- 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.
- All atomic operations on the same variable (global memory address) are serialized.
Atomic Operations

Atomic Operations on Shared Memory

- Very short latency, but still serialized
- Private to each thread block
- Need algorithm work by programmers for the coordination on the global memory access
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;
```
Build Private Histogram

// stride is total number of threads
int stride = blockDim.x * blockDim.x;
while (i < size) {
    atomicAdd(&private_histo[buffer[i]], 1);
    i += stride;
}
// wait for all other threads in the block to finish
__syncthreads();
if (threadIdx.x < 256)

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

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

The operation needs to be associative and commutative

- True for all uses of atomic operations, because they do not guarantee ordering
- Histogram add operation is associative and commutative

The histogram size needs to be small

- How small does it need to be? How small should it be?