ECE408 Spring 2019
Applied Parallel Programming
Lecture 16
Atomic Operations and Histogramming

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

- 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 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
Read-Modify-Write Operations

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.

Timing Scenario #1

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

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

Timing Scenario #2

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

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

Timing Scenario #3

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

• Thread 1 Old = 0
• Thread 2 Old = 0
• Mem[x] = 1 after the sequence
Timing Scenario #4

<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></td>
<td>(1) New ⇔ Old + 1</td>
</tr>
<tr>
<td>3</td>
<td>(0) Old ⇔ Mem[x]</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>(1) New ⇔ Old + 1</td>
<td>(1) Mem[x] ⇔ New</td>
</tr>
<tr>
<td>5</td>
<td>(1) New ⇔ Old + 1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>(1) Mem[x] ⇔ New</td>
<td></td>
</tr>
</tbody>
</table>

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

Without Atomic Operations

Mem[x] initialized to 0

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

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

Atomic Operations –
To Ensure Good Outcomes

<table>
<thead>
<tr>
<th>thread1: Old ⇔ Mem[x]</th>
<th>New ⇔ Old + 1</th>
<th>Mem[x] ⇔ New</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread2: Old ⇔ Mem[x]</td>
<td>New ⇔ Old + 1</td>
<td>Mem[x] ⇔ New</td>
</tr>
</tbody>
</table>

Or

<table>
<thead>
<tr>
<th>thread1: Old ⇔ Mem[x]</th>
<th>New ⇔ Old + 1</th>
<th>Mem[x] ⇔ New</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread2: Old ⇔ Mem[x]</td>
<td>New ⇔ Old + 1</td>
<td>Mem[x] ⇔ New</td>
</tr>
</tbody>
</table>

Atomic Operations in General

- Typically performed by a single ISA instruction on a memory location **address**
  - Read the old value, calculate a new 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 stalled or held in a queue until its turn
  - All threads perform the atomic operation **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

  ```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(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

Thread 0  Thread 1  Thread 2  Thread 3

Atomic operation enforces correct update

Iteration #2 – 2nd letter in each section

Thread 0  Thread 1  Thread 2  Thread 3

Iteration #3

Thread 0  Thread 1  Thread 2  Thread 3

Iteration #4

Thread 0  Thread 1  Thread 2  Thread 3
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

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

More on the Histogram Kernel

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

```c
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

```plaintext
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

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 $1 / (\text{entire shopping time of each customer})$

Hardware Improvements

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

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 $<1/1000$!
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)

ANY MORE QUESTIONS
READ CHAPTER 9