

# XJCO3221 Parallel Computation

Peter Jimack

University of Leeds

Lecture 18: Atomic operations

## Previous lectures

Whenever multiple processing units had read-write access to the same memory location, there are potential **data races**:

- If at least one unit **writes** to the memory [*Lecture 5*].
- Can solve using **critical regions** guarded by **locks/mutexes** [*Lectures 6 and 7*].
- Single instructions can also be made performed **atomically** [*Lecture 6*].

For instance, in OpenMP an atomic instruction looks like:

```
1 #pragma omp atomic
2 count++;
```

# This lecture

GPUs also have memory accessible by multiple **work items** / **threads**:

- **Global** memory, accessible to all work groups.
- **Local** memory, only accessible within a work group.

There is therefore potential data races and the need for **synchronisation**.

Today we will see how GPUs support **atomic operations** in a similar way to a shared memory CPU.

- Also consider an atomic **compare and swap**.

# Atomic operations

## Definition

An **atomic operation** is one that is completed **without interruption** by any other processing unit.

- Usually restricted to simple **arithmetic operations** (addition, subtraction *etc.*)
- Implemented by a combination of compiler and hardware.
- Typically a **much smaller performance penalty** than using locks/mutexes *etc.*

# Load, compute, and store

Consider the following line:

```
1 x -= 2;
```

Even this single instruction performs **three** sub-operations:

- ➊ **Loads** the value of `x`.
- ➋ Performs the **computation** (*i.e.* subtracts 2).
- ➌ **Stores** the updated value.

Two or more processing units might interfere with each other, resulting in a different result to the **serial** equivalent.

This could not happen if the operation was **atomic**.

# Example

Suppose  $x=10$  initially, and two processing units A and B both subtract 2 from  $x$ . Depending on the scheduler, this may happen:

- ➊ A loads the value of  $x$  as 10.
- ➋ B loads the value of  $x$  as 10.
- ➌ A performs its computation:  $10 - 2 = 8$ .
- ➍ A stores 8 to memory.
- ➎ B performs its computation:  $10 - 2 = 8$ .
- ➏ B stores 8 to memory.

The result is  $x = 8$ , rather than  $x = 6$  as expected.

# Constructing a histogram on a GPU

Code on Minerva: `histogram.c`, `histogram.cl`, `helper.h`

Have an array of integers in the range 0 to `maxValue-1` inclusive; want the **histogram** showing the frequency of each value.

- ❶ Memory allocated on the **host** and on the **device**, for both the data and the histogram.
  - `data`, `hist` on the host.
  - `device_data`, `device_hist` on the device.
- ❷ Both initialised on the host and copied to the device.
- ❸ Build, initialise and enqueue a **kernel** to construct the device histogram.
  - **One work item per data element**, e.g. `data[i]`.
- ❹ Copy the histogram back to the host using `clEnqueueReadBuffer()`.

# Kernel 1: Direct to global; no atomics

```
1  __kernel
2  void histogramNoAtomic(
3      __global int *device_hist,
4      __global int *device_data,
5          int  maxValue )
6  {
7      int gid = get_global_id(0);
8
9      // Data value.
10     int val = device_data[gid];
11
12     // Check range before updating.
13     if( val >= 0 && val < maxValue )
14         device_hist[val]++;
15 }
```



## Kernel 2: Direct to global; atomic.

Code **fails** because the update of `device_hist` is **not atomic**

```
1 device_hist[val]++;           // Load, compute, store.
```

Many additions to the histogram are lost, resulting in lower totals.

In OpenCL, can make this **atomic** by using `atomic_inc()`:

```
1 __kernel
2 void histogramAtomic( ... )
3 {
4     ... // As before.
5     if( val >= 0 && val < maxValue )
6         atomic_inc( &(amp;device_hist[val]) );
7 }
```

This now works as expected.

# Atomic operations in OpenCL

There are many atomic operations in OpenCL<sup>1</sup>:

<code>atomic_inc</code> , <code>atomic_dec</code>	Increment, decrement.
<code>atomic_add</code> , <code>atomic_sub</code>	Addition, subtraction.
<code>atomic_min</code> , <code>atomic_max</code>	Smaller or larger of two arguments.
<code>atomic_and</code> , <code>atomic_or</code> , <code>atomic_xor</code>	Bitwise operations.
<code>atomic_xchg</code> , <code>atomic_cmpxchg</code>	Exchange, compare and exchange ( <i>see later</i> ).

---

<sup>1</sup>Similar in CUDA, *i.e.* `atomicAdd()`, `atomicInc()` etc.

# Optimising with local memory

Having a **single** histogram in global memory is not efficient:

- Potential for **very many** work items attempting to access the **same** global memory location **almost simultaneously**.

More efficient for each **work group** to calculate its own histogram **in local memory**, then update the global histogram **at the end**.

- Fewer **competing** work items for the local histogram.
- Local memory is faster anyway.

Aside: Could use a similar strategy for a **multi-threaded CPU** (*i.e.* each **thread** constructs its own histogram).

## Kernel 3: Local histogram (1)

```
1  __kernel
2  void localHistogram( ..., __local int *local_hist )
3  {
4      int
5          i,
6          lid  = get_local_id  (0),
7          gid  = get_global_id (0),
8          size = get_local_size(0);
9
10     // Clear the histogram.
11     for( i=lid; i<maxValue; i+=size )
12         local_hist[i] = 0;
13
14     // Ensure histogram fully initialised.
15     barrier(CLK_LOCAL_MEM_FENCE);
16
17     // (cont'd next slide).
```

## Kernel 3: Local histogram (2)

```
1  // (from previous slide).
2
3  // Add to the local histogram.
4  int val = device_data[gid];
5  if( val >= 0 && val < maxValue )
6      atomic_inc( &(amp;local_hist[val]) );
7
8  // Ensure local histogram calculation complete
   before moving on.
9  barrier(CLK_LOCAL_MEM_FENCE);
10
11 // Atomic add the local histogram to the global one.
12 for( i=lid; i<maxValue; i+=size )
13     atomic_add( &(amp;device_hist[i]), local_hist[i] );
14 }
```

You should see a performance improvement using this method.

Could have had **one** work item in each group initialise and update the **entire** local histogram, e.g.:

```
1 if( lid==0 )  
2   for( i=0; i<maxValue; i++ )  
3     atomic_add( &(amp;device_hist[i]), local_hist[i] );
```

This would work, but would be slower (*check*).

Instead use as many work items as possible:

```
1 for( i=lid; i<maxValue; i+=size )  
2   atomic_add( &(amp;device_hist[i]), local_hist[i] );
```

- Each *i* in the range realised by **exactly** one work item.
- Spans full range even if  $\text{size} < \text{maxValue}$ .

# Atomic exchange and compare-and-exchange

```
int atomic_xchg(int *p,int val):
```

- ① Sets old=\*p.
- ② Sets \*p=val.
- ③ Returns old.

```
int atomic_cmpxchg(int *p,int cmp,int val):
```

- ① Sets old=\*p.
- ② Sets \*p=val **if** \*p==cmp; otherwise does not change.
- ③ Returns old.

In both cases, p can be in local or global memory, and the data type can be int, unsigned int or float.

# Uses of compare and exchange

Many low-level parallel frameworks provide similar functionality, sometimes referred to as CAS = Compare And Swap:

- `atomicExch()`, `atomicCAS()` in CUDA;
- `std::atomic::compare_exchange_weak()`,  
`std::atomic::exchange()` in C++11.

Common uses include:

- 1 Implementing a **lock** or **mutex**.
- 2 **Lock-free** implementations.

Examples below are for OpenCL, but just as relevant for CUDA and multi-core CPUs.



# Spinlock

Suppose an `int` variable `lock` was accessible to multiple threads.

- `lock` takes **two** values, 0 and 1.
- Take 0 to be **unlocked**, 1 to be **locked**.

A simple 'spinlock' can be implemented as follows:

```
1 int lock;           // 0 or 1. Accessible by all threads.  
2 ...  
3 while( atomic_cmpxchg(&lock,0,1)==1 );
```

- Infinite while loop, until `lock==0`.
- Then sets `lock=1` and continues past line 3.
- Does all this **atomically**.

# Why atomic?

Consider what could happen **without** atomicity:

```
1 while( lock==1 );  
2 lock = 1;  
3 ...
```

- ➊ One thread / work item sees `lock==0` and continues to line 2.
- ➋ A second thread **also** sees `lock==0` and proceeds to line 2, **before the first thread sets** `lock=1`.
- ➌ The first thread now sets `lock=1`.
- ➍ The second thread **also** sets `lock=1`.
- ➎ **Both** continue to line 3!

## Spinlocks vs. locks/mutexes:

- **Pro:** Spinlocks **faster**, as do not put the thread to sleep.
- **Con:** Spinlocks waste CPU/GPU cycles.

## Spinlocks on GPUs:

Note lock could be in **global** memory.

- Accessible to work items from **different** work groups.
- May seem this can be used to **synchronise between** groups.

But recall the warning from Lecture 17:

Cannot guarantee all work groups are active on the device **at the same time** (as some may be queued), so this not a **robust** synchronisation mechanism.

# Lock-free data structures

Atomic compare-and-exchange can also be used to implement **thread-safe** access to data structures **without requiring locks**, and the associated overhead.

Such **lock free** data structures, if they can be achieved, are desirable for good parallel performance.

Example: **Prepending** an item to a singly **linked list**.

- Need to ensure old and new head nodes updated **together**
- Use `atomic_cmpxchg()` in an infinite loop<sup>1</sup>.

---

<sup>1</sup>McCool *et al.*, *Structured parallel programming* (Morgan-Kaufman, 2012).

## Basic idea of a lock-free data linked list

```
1 struct node { ... };           // Some data structure.
2 node *head;                    // Head of list.
3 void prependToList( node *a ) // 'a' becomes head.
4 {
5     while(true)
6     {
7         // Take a copy of current head pointer.
8         node *b = head;
9
10        // Link to the node being added.
11        a->next = b;
12
13        // Only update head if not just changed by another
14        // work item/thread; else try again from line 6.
15        if( atomic_cmpxchg(head,b,a)==b ) break;
16    }
17 }
```

```
if( atomic_cmpxchg(head,b,a)==b ) break;
```

If only a single thread was involved:

- 1 old=\*head, *i.e.* old==b, the first item in the list.
- 2 **Compare-exchange:** \*head==b, so changes \*head to a.
- 3 atomic\_cmpxchg returns b, so will break from while loop.

This is the expected behaviour.

However, in a multi-threaded context:

- 1 Another thread may change \*head from b before line 15.
- 2 Since \*head!=b, will **not** change it.
- 3 Will return some value !=b, so will try again.

## Summary and next lecture

This lecture we have revisited **atomic operations** with an emphasis on GPUs:

- Atomics used to ensure correct updates of memory accessible by multiple work items.
- Atomic **compare and exchange** can be used to implement a spinlock, lock-free data structures, *etc.*

The next lecture is the last on GPU programming when we will look at events and **task parallelism**.