University of Leeds

Lecture 18: Atomic opera

# Previous lectures



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

- 
- https://eduassistpro.github.i
- texes
- Single instructions can also be made perfor
  *[Lecture 6]*.



For instance, in OpenMP an atomic instruction looks like:

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

# Today's lecture



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



Ther
**synchronisation**.



Today we will see how GPUs support
similar way to a shared memory CPU.

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

# Atomic operations



**Definition**

**ut**



- Usually restricted to simple **arith** on, subtraction *etc.*)



- Implemented by a combination of compil
- Typically a (much) **smaller performance penalty** than using locks/mutexes *etc.*

# Load, compute, and store



Consider the following line:

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

Even t



1. 
2. 
3. **Stores** the updated value.



Two or more processing units might interfere wit
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:

1.
2. 
3.
4. A stores 8 to memory.
5. B performs its computation: 10 − 2 = 8
6. 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.

1. th
   - 
   - `device_data`, `device_hist` on the device.
2. Both initialised on the host and copied to the d
3. Build, initialise and enqueue a ke histogram.
   - **One work item per data element**, *e.g.* `data[i]`.
4. Copy the histogram back to the host using `clEnqueueReadBuffer()`.

# Kernel 1: Direct to global; no atomics



```
1  __kernel
2  void histogramNoAtomic(
3    __
4    __
5
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.
```

Man



In Op ;

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



This now works as expected.

# Atomic operations in OpenCL



There are many atomic operations in OpenCL[1]:

| | |
|---|---|
| `atomic_inc`, `atomic_dec` | Increment, decrement. |
| a _ _ | |
| a _ _ | |
| `atomic_and`, `atomic_or`, `atomic_xor` | Bitwi |
| `atomic_xchg`, `atomic_cmpxchg` | Exch chan |



[1] 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

.



Mor m

**in local memory**, then update the global hist .

- Fewer **competing** work items for the lo
- 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    in
5
6
7
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    in   v
5    if   v
6
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( &(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
```

This

Instead use as many work items as possible:

```
1 for( i=lid; i<maxValue; i+=size )
2   atomic_add( &(device_histo[i]), local_histo[i] );
```

- Each `i` in the range realised by **exactly** one work item.
- Spans full range even if `size<maxValue`.

# Atomic exchange and compare-and-exchange



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

1. Sets `old=*p`.
2. 
3. 



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

1. Sets `old=*p`
2. Sets `*p=val` if `*p==cmp`, otherwise d
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:

-
-



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.
-



A sim

```
1 int lock;        // 0 or 1. Accessible by all threads.
2 ...
3 while( atomic_xchg(&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 ...
```



1. One thread / work item sees `lock==0` and continues to line 2.
2. A second thread **also** sees `lock=`
   **before the first thread sets** `loc`
3. The first thread now sets `lock=1`
4. The second thread **also** sets `lock=1`.
5. **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.

**Spin**

Note

- 
- May seem this can be used to **sy** ps.

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 th



s



Example: **Prepending** an item to a singly **r**

- Need to ensure old and new `head`
- Use `atomic_cmpxchg()` in an infinite loop[1].

[1] McCool *et al.*, *Structured parallel programming* (Morgan-Kauffman, 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    wh
6    {
7
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.
2.
3. _ o a. loop.



This is the expected behaviour.



However, in a multi-threaded context:

1. Another thread may change `*head` b
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 emp



- Atomic **compare and exchange** can be used to implement a spinlock, lock-free data structures,



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