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:

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

Today's 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:


```
x -= 2;
```

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

1. **Loads** the value of x.
2. Performs the **computation** (*i.e.* subtracts 2).
3. **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:

1. `A` loads the value of `x` as 10.
2. `B` loads the value of `x` as 10.
3. `A` performs its computation: `10 - 2 = 8`.
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. 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.
2. Both initialised on the host and copied to the device.

3. Build, initialise and enqueue a **kernel** to construct the device 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


```
__kernel
void histogramNoAtomic(
  __global int *device_hist,
  __global int *device_data,
           int maxValue )
{
  int gid = get_global_id(0);

  // Data value.
  int val = device_data[gid];

  // Check range before updating.
  if( val>=0 && val<maxValue )
    device_hist[val]++;
}
```

Kernel 2: Direct to global; atomic.

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

```
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()`:

```
__kernel
void histogramAtomic( ... )
{
  ...  // As before.
  if( val>=0 && val<maxValue )
    atomic_inc( &(device_hist[val]) );
}
```


This now works as expected.

Atomic operations in OpenCL

There are many atomic operations in OpenCL[1]:

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

[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 **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)


```
__kernel
void localHistogram( ..., __local int *local_hist )
{
  int
    i,
    lid  = get_local_id (0),
    gid  = get_global_id (0),
    size = get_local_size(0);

  // Clear the histogram
  for( i=lid; i<maxValue; i+=size )
    local_hist[i] = 0;

  // Ensure histogram fully initialised.
  barrier(CLK_LOCAL_MEM_FENCE);

  // (cont'd next slide).
```

Kernel 3: Local histogram (2)


```
  // (from previous slide)

  // Add to the local histogram.
  int val = device_data[gid];
  if( val>=0 && val<maxValue )
    atomic_inc( &(local_hist[val]) );

  // Ensure local histogram calculation complete
     before moving on.
  barrier(CLK_LOCAL_MEM_FENCE);

  // Atomic add the local histogram to the global one.
  for( i=lid; i<maxValue; i+=size )
    atomic_add( &(device_hist[i]), local_hist[i] );
}
```

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.*:


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

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

Instead use as many work items as possible:

```
for( i=lid; i<maxValue; i+=size )
  atomic_add( &(device_hist[i]), local_hist[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. Sets `*p=val`.
3. Returns `old`.

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

1. Sets `old=*p`.
2. Sets `*p=val` <u>if</u> `*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 ...
```


1. One thread / work item sees `lock==0` and continues to line 2.
2. A second thread **also** sees `lock==0` and proceeds to line 2, **before the first thread sets** `lock=1`.
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.

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[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    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**.