Lecture 4
Warp shuffles, reduction and scan operations

Prof Wes Armour

wes.armour@eng.ox.ac.uk

Oxford e-Research Centre

Department of Engineering Science

Learning outcomes

In this fourth lecture we will learn about warp shuffle instructions, reduction and scan operations.

You will learn about:

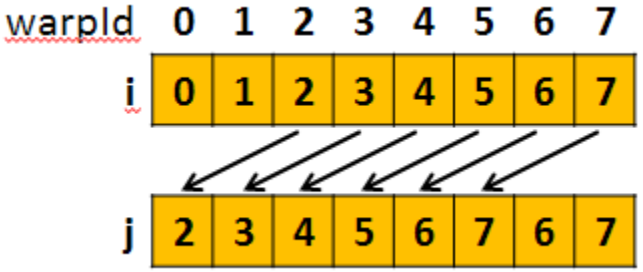
- Different types of warp shuffle instructions and why they are useful.
- How warp shuffles can be used to construct different memory access patterns.
- The reduction algorithm and implementation on a GPU.
- The scan algorithm and implementation on a GPU.

Warp shuffles - Introduction

Warp shuffles provide a mechanism for fast exchange of data between threads within the same warp.

There are four variants:

- `__shfl_up_sync`
  Copy from a lane with lower ID relative to caller.
- `__shfl_down_sync`
  Copy from a lane with higher ID relative to caller.
- `__shfl_xor_sync`
  Copy from a lane based on bitwise XOR of own lane ID.
- `__shfl_sync`
  Copy from indexed lane ID.



Here the lane ID is the position within the warp (threadIdx.x%32 for 1D blocks)

https://devblogs.nvidia.com/faster-parallel-reductions-kepler/

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#warp-shuffle-functions

Warp shuffles – up / down

`T shfl_up_sync(unsigned mask, T var, unsigned int delta);`

- `T` can be `int, unsigned int, long, unsigned long, long long, unsigned long long, float` or `double`. With the `cuda_fp16.h` header included, `T` can also be `__half` or `__half2`
- `mask` controls which threads are involved — usually set to `-1` or `0xffffffff`, equivalent to all 1's
- `var` is a local register variable, the data to be "shuffled". This can be `int, unsigned int, long, unsigned long, long long, unsigned long long, float` or `double`.
- `delta` is the offset within the warp (e.g. how many lanes to shuffle across) – if the appropriate thread does not exist (i.e. it's off the end of the warp) then the value is taken from the current thread.

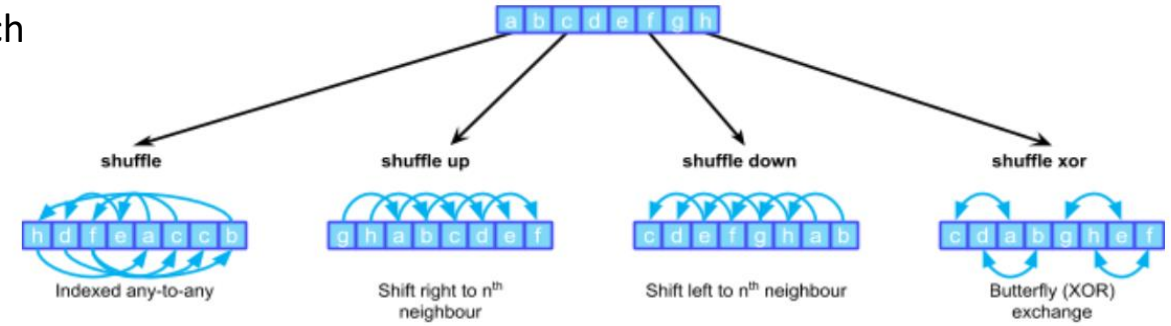
`T shfl_down_sync(unsigned mask, T var, unsigned int delta);` is defined similarly

Warp shuffles – XOR / sync

```
T shfl xor sync(unsigned mask, T var, int laneMask);
```

This performs an XOR (exclusive or) operation between `laneMask` and the calling thread's `laneID` to determine the lane from which to copy the value (`laneMask` controls which bits of `laneID` are "flipped").

This might seem abstract and confusing, but think of it as providing a "butterfly" type of addressing, which is **very** useful for reduction operations and FFTs.




```
T shfl sync(unsigned mask, T var, int srcLane);
```

This is the most general form of shuffle instruction, it copies data from `srcLane`

Warp shuffles – Warning!

When using shuffles, it's really important to remember…

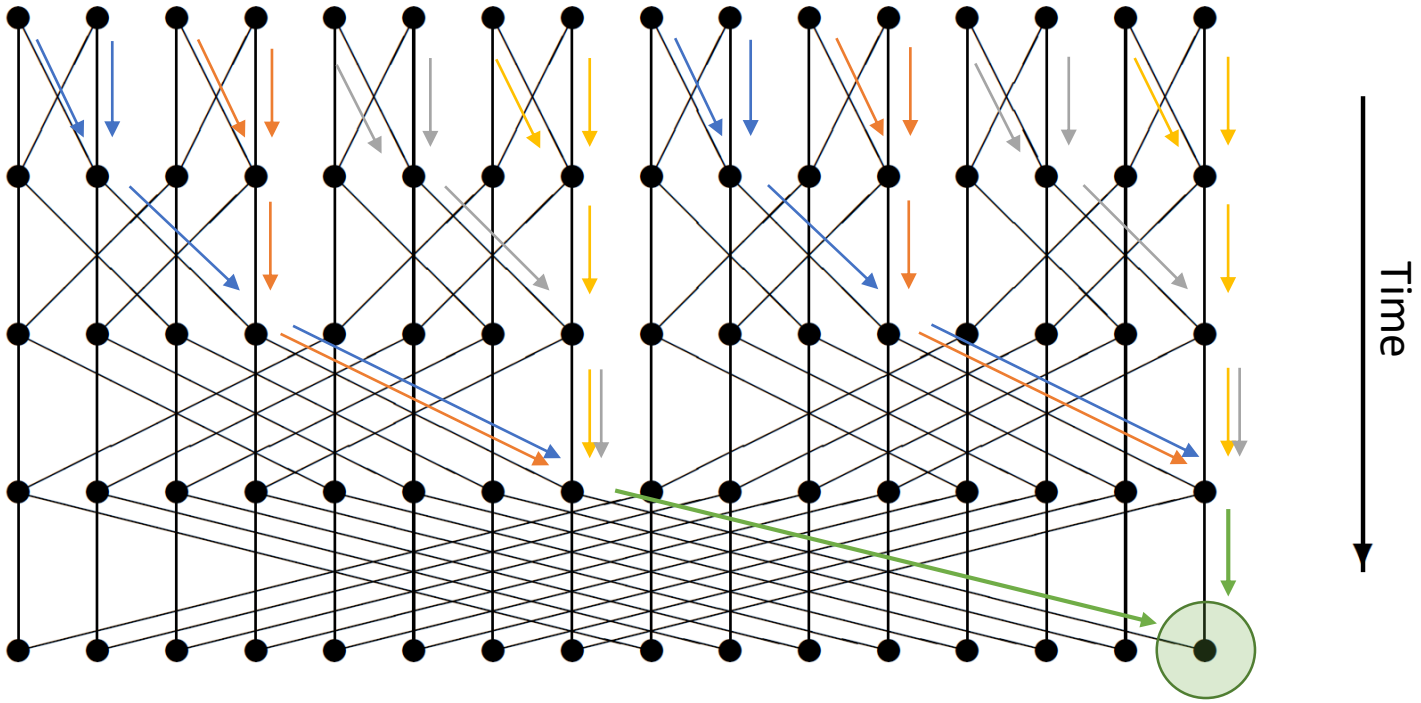
Threads may only read data from another thread which is actively participating in the shuffle command. If the target thread is inactive, the retrieved value is undefined.

This means you must be very careful with <u>conditional code</u>.

Warp shuffles – sum within a warp

Let's consider two different ways to sum up all of the elements within a warp. Method 1…

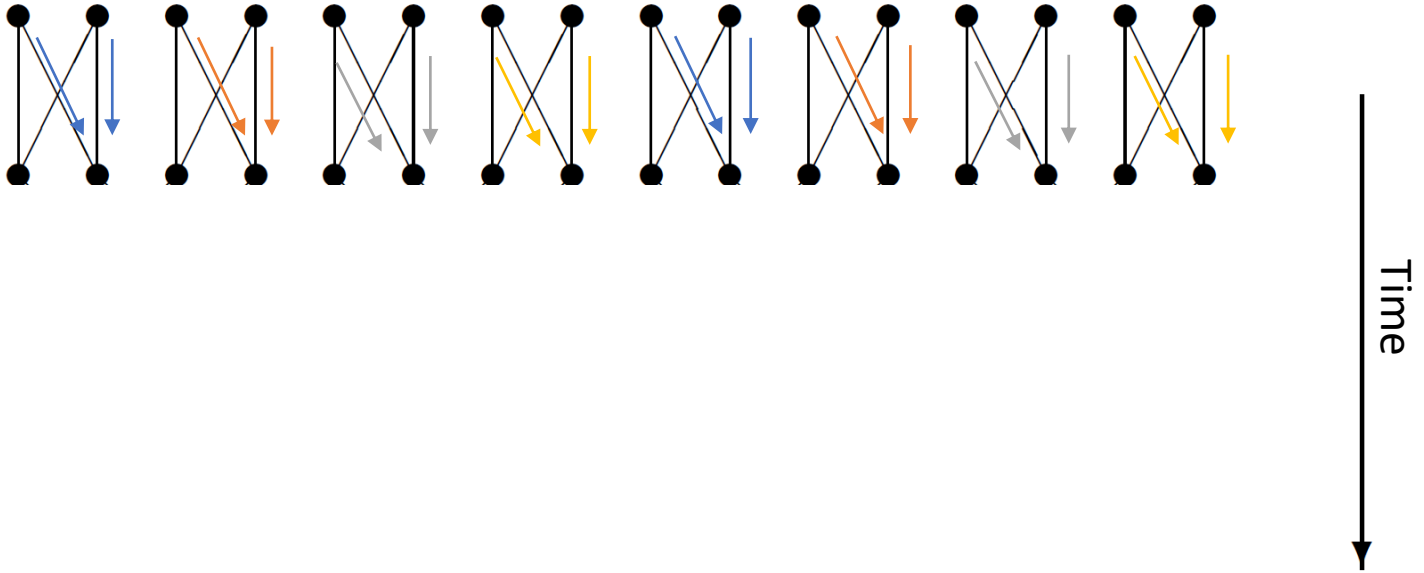
```
for (int i=1; i<32; i*=2)
    value += __shfl_xor_sync(-1, value, i);
```



Warp shuffles – sum within a warp

Let's consider two different ways to sum up all of the elements within a warp. Method 1…

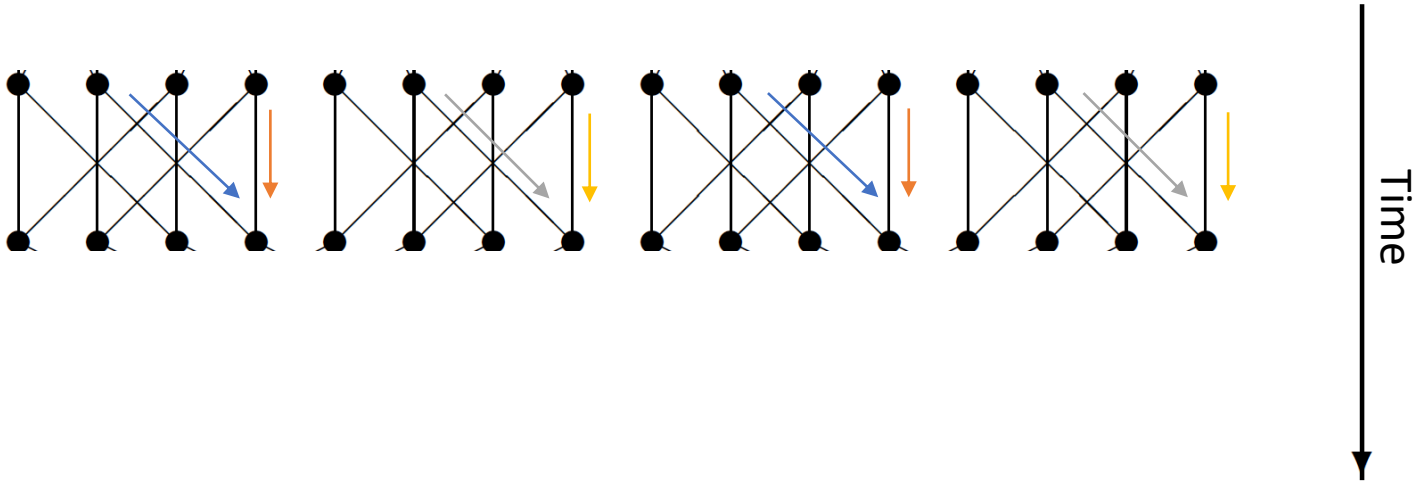
```
for (int i=1; i<32; i*=2)
    value += __shfl_xor_sync(-1, value, i);
```



Warp shuffles – sum within a warp

Let's consider two different ways to sum up all of the elements within a warp. Method 1…

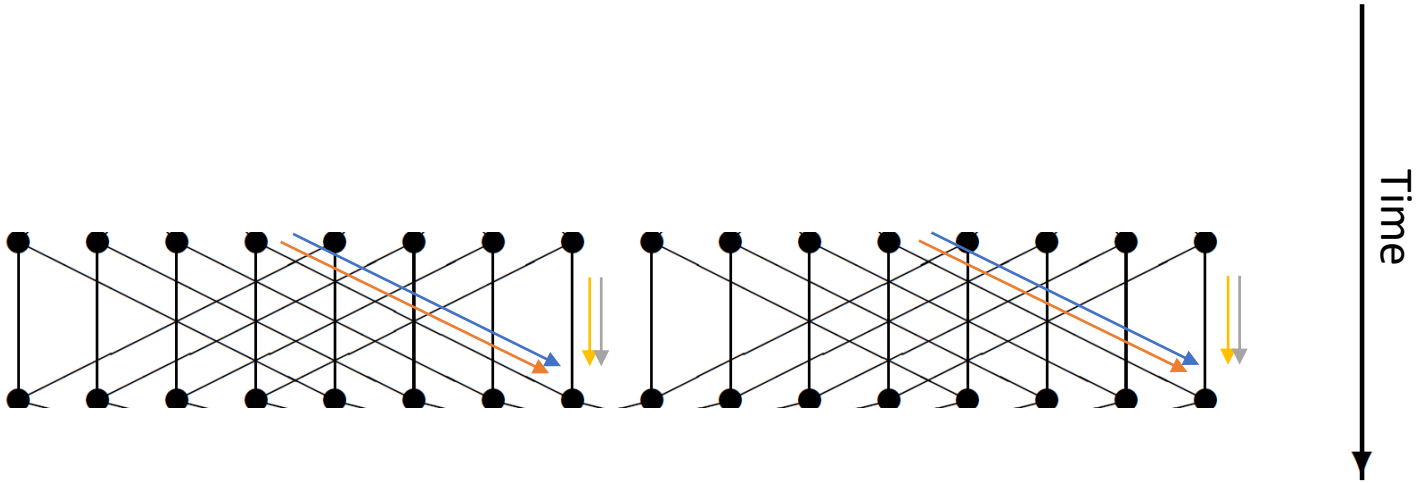
```
for (int i=1; i<32; i*=2)
    value += __shfl_xor_sync(-1, value, i);
```



Warp shuffles – sum within a warp

Let's consider two different ways to sum up all of the elements within a warp. Method 1…

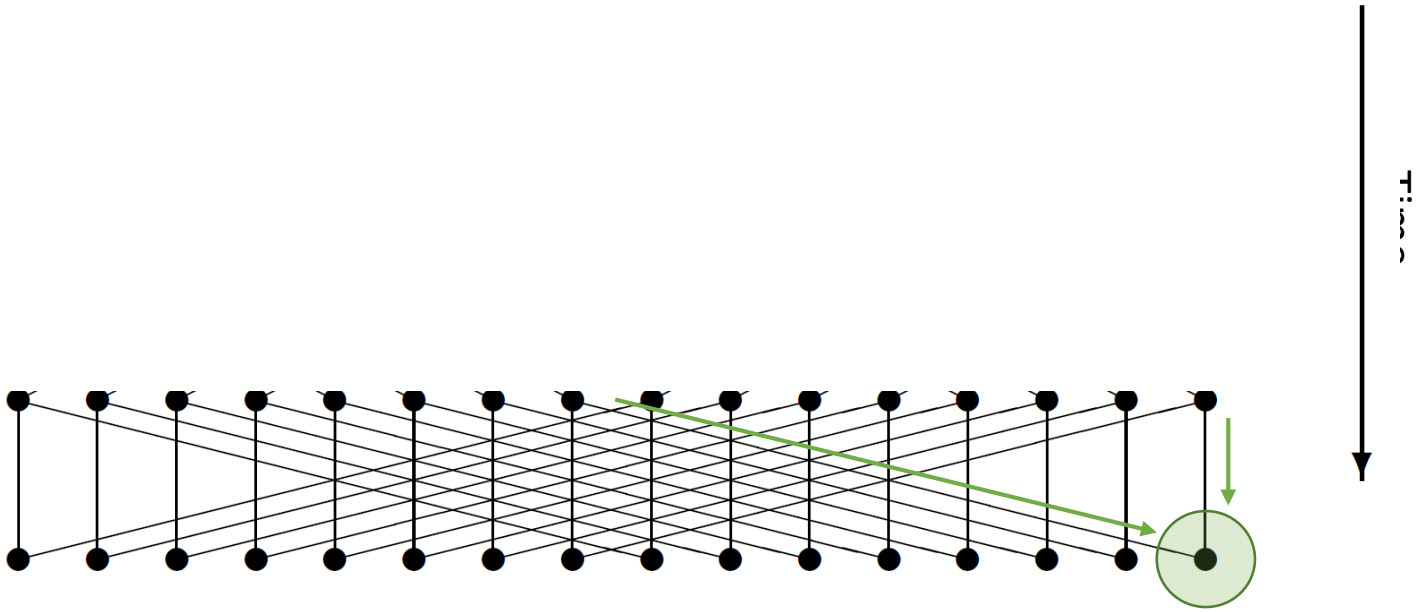
```
for (int i=1; i<32; i*=2)
    value += __shfl_xor_sync(-1, value, i);
```



Warp shuffles – sum within a warp

Let's consider two different ways to sum up all of the elements within a warp. Method 1…

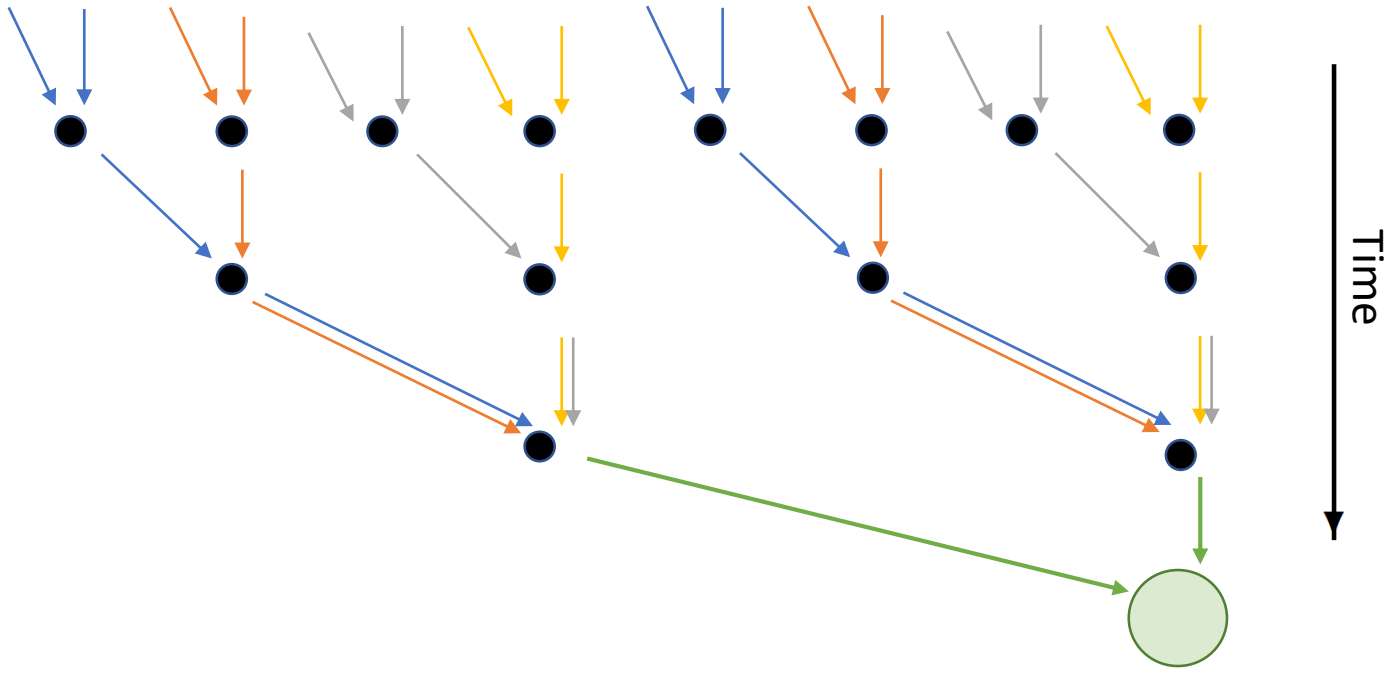
```
for (int i=1; i<32; i*=2)
    value += __shfl_xor_sync(-1, value, i);
```



Warp shuffles – sum within a warp

Let's consider two different ways to sum up all of the elements within a warp. Method 1…

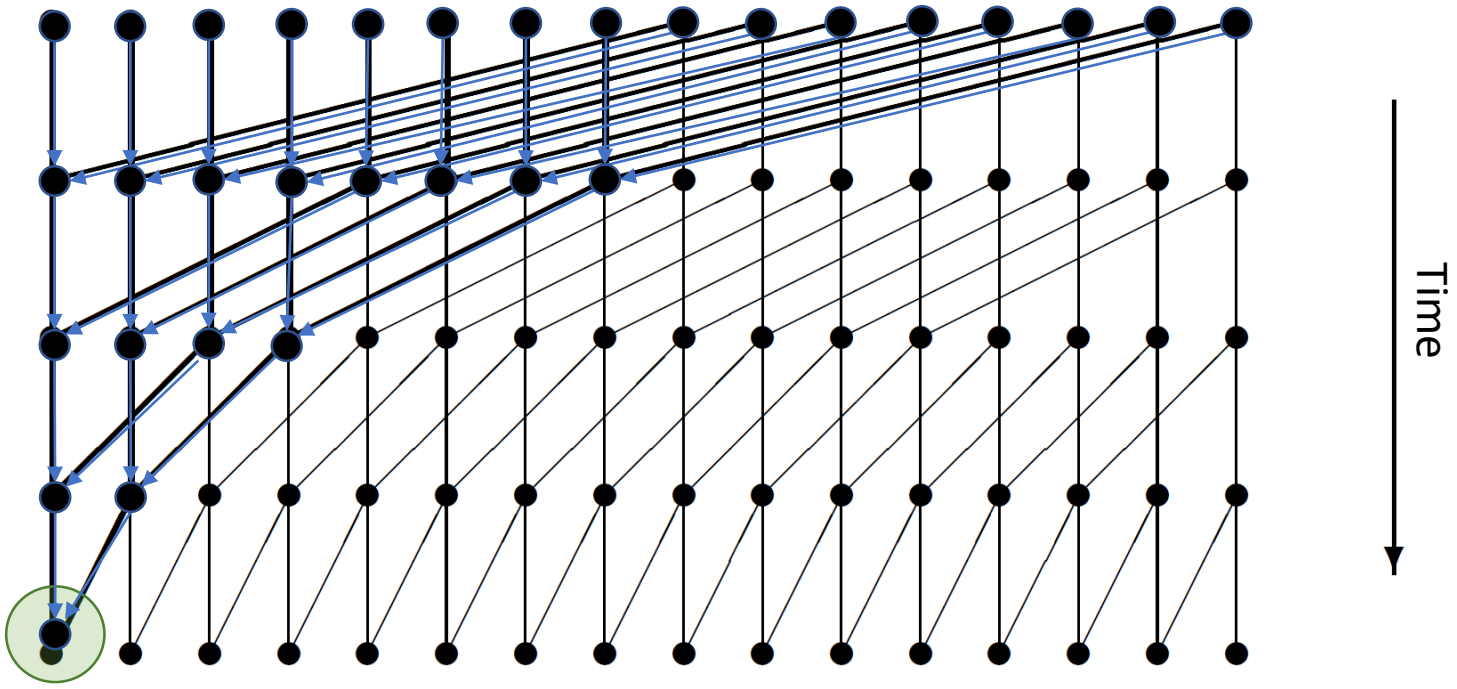
```
for (int i=1; i<32; i*=2)
    value += __shfl_xor_sync(-1, value, i);
```



Warp shuffles – sum within a warp

Method 2…

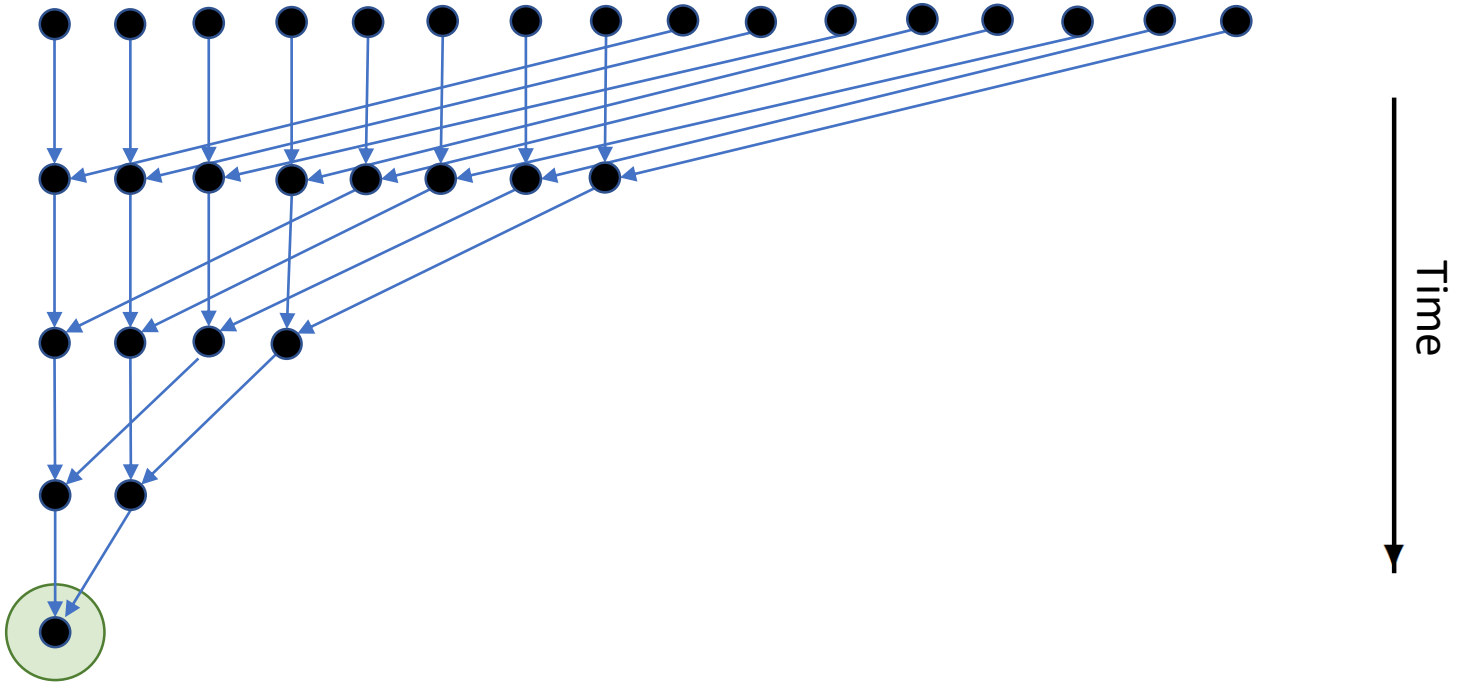
```
for (int i=16; i>0; i=i/2)
    value += __shfl_down_sync(-1, value, i);
```



Warp shuffles – sum within a warp

Method 2…

```
for (int i=16; i>0; i=i/2)
    value += __shfl_down_sync(-1, value, i);
```



Reduction

The most common reduction operation is computing the sum of a large array of values, some examples of where this is needed are:

- Averaging in Monte Carlo simulation.
- Computing RMS change in finite difference computation or an iterative solver.
- Computing a vector dot product in a CG or GMRES iteration.
- Computing mean and standard deviations in signal processing.

$$\mu = \frac{1}{n}\sum_{i=0}^{n-1} x_i$$

Reduction

Other common reduction operations are to compute a minimum or maximum (here you would just retain the min or max value when comparing two elements).

Key requirements for a reduction operator $\circ$ are:

- commutative: $a \circ b = b \circ a$
- associative: $a \circ (b \circ c) = (a \circ b) \circ c$

Together, they mean that the elements can be re-arranged and combined in any order.

(Note: in MPI there are special routines to perform reductions over distributed arrays.)

GPU reduction approach

Over the next few slides we will look at a summation reduction, as mentioned in the last slide, to extend this to min or max is easy.

We begin by assuming that each thread starts with a one value, the approach is then to…

- First add the values within each thread block, to form a partial sum.
- Then add together the partial sums from all of the blocks.

Lets look at these stages in turn.

Local (thread block) reduction

The first phase is constructing a partial sum of the values within a thread block (locally).

The first question we must answer is –
Q1: Where is the parallelism?

"Standard" summation uses an accumulator, adding one value at a time, it **is sequential**.

$$n\mu = (\dots(((x_0 + x_1) + x_2) + x_3) + x_4) + x_5 \dots$$

Parallel summation of N values:

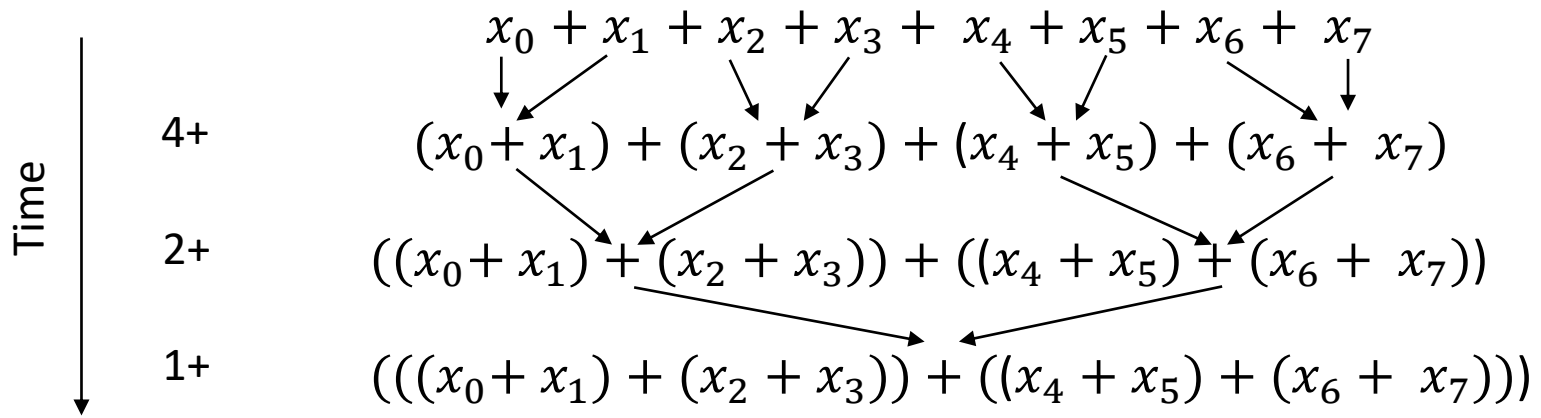
- First sum them in pairs to get N/2 values
- Repeat the procedure until we have only one value

$$n\mu = (x_0 + x_1) + (x_2 + x_3) + (x_4 + x_5) \dots x_n$$

Local reduction

The schematic below gives an example of the algorithm that we outlined on the previous slide for eight elements.

We see that 4 + 2 + 1 = 7 additions are required to reach the final summation. However (within a row) these can all be performed in parallel.



Local reduction

The next thing to consider is -
Q2: Does our algorithm will have any problems with wrap divergence?

Note that not all threads can be busy all of the time:

- N/2 operations in first phase
- N/4 in second
- N/8 in third
- etc.

For efficiency, we want to make sure that each warp is either fully active or fully inactive (as far as possible).

Local reduction

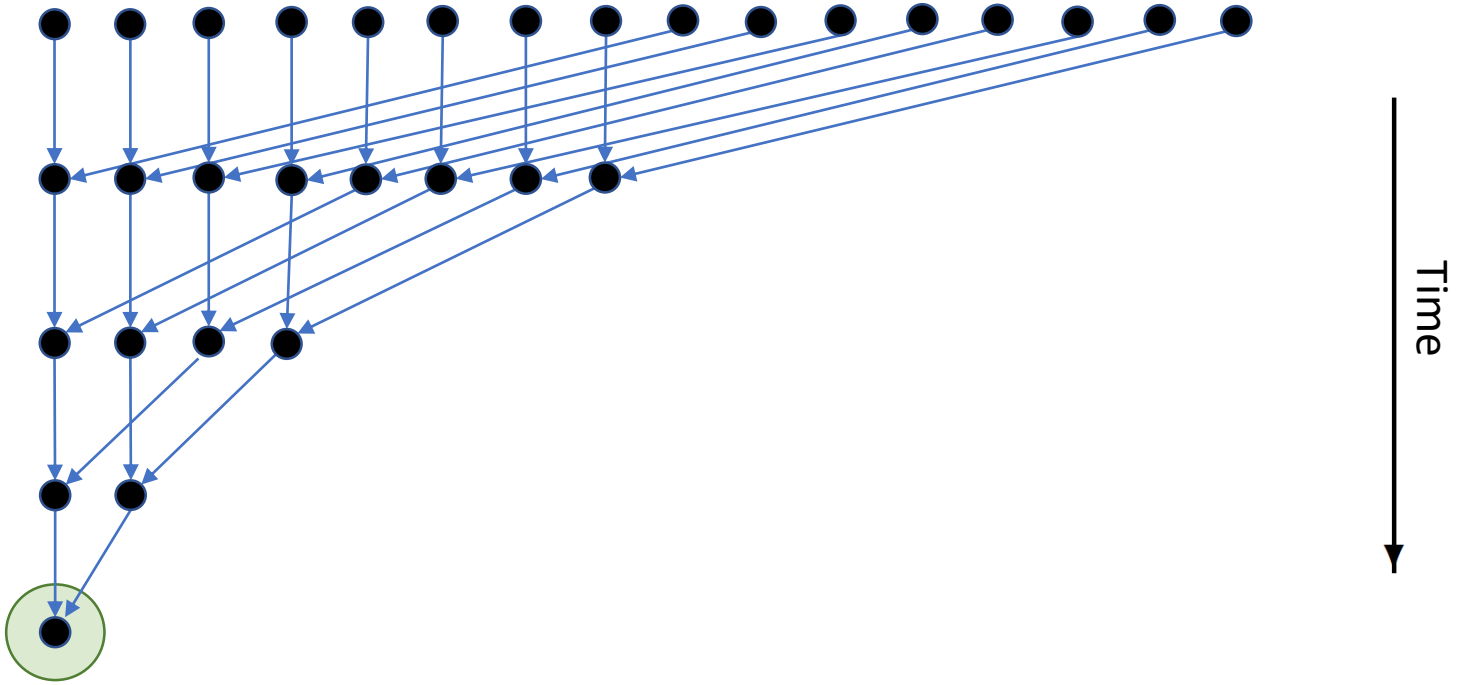
Next we need to think about where to hold our data -
Q3: Where should data be held?

Threads need to access results produced by other threads:

- Can’t be in registers, so we have to use either global memory or shared memory (or a combination).
- Global device arrays would be too slow, so use shared memory.
- Because we are not working within a warp, and data is being shared across warps, we need to think about synchronisation.

Local reduction

We will use the pervious data access pattern that we outlined for reduction within a warp using shuffles (method 2). The second half of our data will be added pairwise to first half of our data by the leading set of threads.



Local reduction

Here is an example kernel that will perform the reduction algorithm that we have just outlined.

Things to note are:

- The use of dynamic shared memory – size has to be declared when the kernel is called.
- The use of `__syncthreads()` to make sure previous operations have completed.
- The first thread outputs final partial sum into specific place for that block.
- Could use shuffles when only one warp still active.
- Alternatively, could reduce each warp, put partial sums in shared memory, and then the first warp could reduce the sums – requires only one `__syncthreads()`.

```
__global__ void sum(float *d_sum,float *d_data)
{
 extern __shared__  float temp[];
 int tid = threadIdx.x;

 temp[tid] = d_data[tid+blockIdx.x*blockDim.x];

 for (int d=blockDim.x>>1; d>=1; d>>=1) {
   __syncthreads();
   if (tid<d) temp[tid] += temp[tid+d];
 }

 if (tid==0) d_sum[blockIdx.x] = temp[0];
}
```

Global (device memory) reduction

One method to do this is to easily extend our previously outlined local reduction so that the result of each local reduction is then reduced…

Each local reduction puts the partial sum for each block in a different array element in a global array.

These partial sums can then be transferred back to the host for the final summation.

Practical 4 will look at this method.

Global (device memory) reduction

An alternative method (which is even easier) is to use the atomic add discussed in the previous lecture, and replace…

```
if (tid==0) d_sum[blockIdx.x] = temp[0];
```

With

```
if (tid==0) atomicAdd(&d_sum,temp[0]);
```

Global (device memory) reduction

More generalised reduction operations could use the atomic lock mechanism (discussed in the previous lecture). To do this we would replace:

```
if (tid==0) d_sum[blockIdx.x] = temp[0];
```

with

```
if (tid==0) {
        do {} while(atomicCAS(&lock,0,1)); // set lock

        *d_sum += temp[0];
        __threadfence();                   // wait for write completion

        lock = 0;                          // free lock
}
```

The scan operation

The scan operation can be defined as follows…

Given an input vector u_i $i = 0, \ldots, I-1$, compute:

$$v_j = \sum_{i<j} u_i \quad \text{for all} \quad j < I$$

$$v_1 = u_0$$
$$v_2 = u_0 + u_1$$
$$v_3 = u_0 + u_1 + u_2$$

Why is this algorithm important?

- It is a key part of many sorting routines.
- It arises in particle filter methods in statistics.
- It's related to solving long recurrence equations:

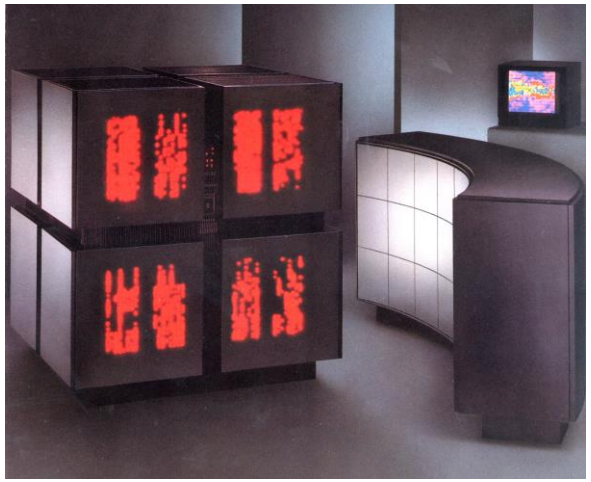
$$v_{n+1} = (1 - \lambda_n)v_n + \lambda_n u_n$$

- Finally, it's a good example that looks impossible to parallelise!

The scan operation

Before we look at how the algorithm for a GPU works, here are the “take home” points…

- Many parallel algorithms are tricky – don’t expect them all to be obvious.
- Do a good literature search! Check the examples in the CUDA SDK, Use Google – don’t put lots of effort into re-inventing the wheel.
- Sometimes relevant literature may be 25–30 years old, lots of algorithmic work was done for computers like CRAY vector machines and Thinking Machines’ massively-parallel CM5.



https://www.flickr.com/photos/argonne/8734933251/in/photostream/

The scan operation

The parallel scan algorithm is similar to the global reduction that we outlined previously.

The strategy is…

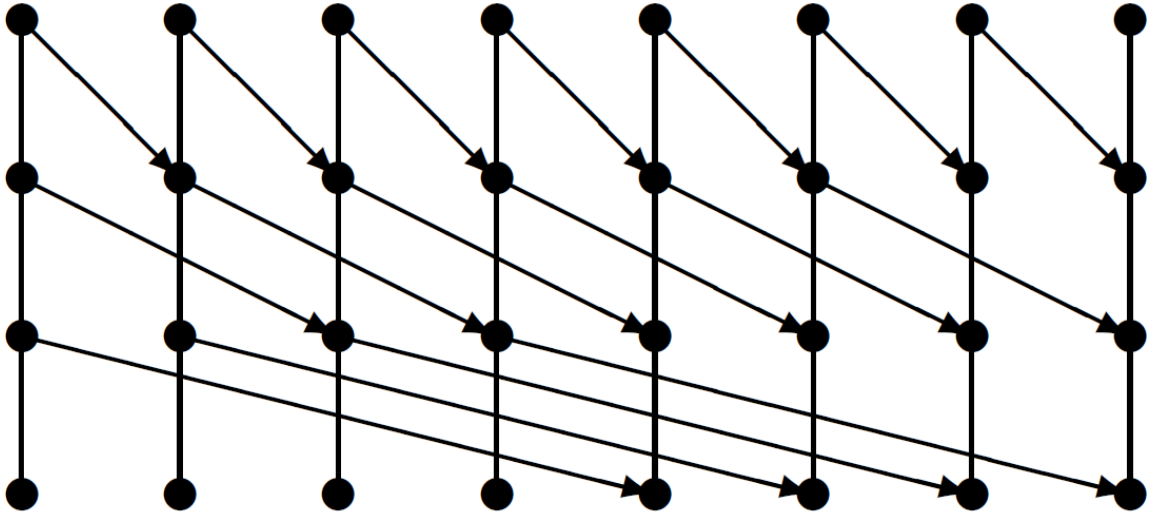
- Perform a local scan within each block.
- Add on the sum of all preceding blocks.

Let's look at two different approaches to the local scan, both are similar to the local reduction discussed previously.

The first approach is very simple using shared memory, but requires $O(N\ logN)$ operations.

The second approach is more efficient, it employs warp shuffles and uses a recursive structure, so more complicated, but only requires with $O(N)$ operations.

Local scan – first algorithm



- After n passes, each sum has it's local value plus preceding $2^n - 1$ values.
- $\log_2 N$ passes, and $O(N)$ operations per pass, so $O(N\ logN)$ operations in total.

Local scan – first algorithm

Points to note:

- Data is stored into shared memory in `temp[]`
- For loop steps through `1, 2, 4, 8, 16…`
- Ternary operator, `a ? b : c` evaluates to `b` if the value of `a` is true, `c` otherwise.
- So set `temp2` to `temp[tid-d]` if `tid >= d`, zero otherwise!
- Finally add `temp2` to `temp[tid]`.

Note both `__syncthreads()` are needed… all threads need to use the same value of `d`, with a synchronised `temp[]` array and the `tid-d` will cause inter-warp accesses.

```
__global__ void scan(float *d_data) {

 extern __shared__ float temp[];
 int tid   = threadIdx.x;
 temp[tid] = d_data[tid+blockIdx.x*blockDim.x];

 for (int d=1; d<blockDim.x; d<<=1) {
   __syncthreads();
   float temp2 = (tid >= d) ? temp[tid-d] : 0;
   __syncthreads();
   temp[tid] += temp2;
 }

 ...
}
```

Local scan – second algorithm

Our second algorithm has the same data access pattern as our first, however this version begins by using warp shuffles to perform a scan within each warp, it then stores the warp sum.

```
__global__ void scan(float *d_data) {
  __shared__ float temp[32];
  float temp1, temp2;
  int tid = threadIdx.x;
  temp1   = d_data[tid+blockIdx.x*blockDim.x];

  for (int d=1; d<32; d<<=1) {
    temp2 = __shfl_up_sync(-1, temp1,d);
    if (tid%32 >= d) temp1 += temp2;
  }

  if (tid%32 == 31) temp[tid/32] = temp1;
  __syncthreads();
  ...
```

Local scan – second algorithm

Next we perform a scan of the warp sums (assuming no more than 32 warps).

```
if (tid < 32) {
  temp2 = 0.0f;
  if (tid < blockDim.x/32)
    temp2 = temp[tid];

  for (int d=1; d<32; d<<=1) {
    temp3 = __shfl_up_sync(-1, temp2,d);
    if (tid%32 >= d) temp2 += temp3;
  }
  if (tid < blockDim.x/32) temp[tid] = temp2;
}
```

Local scan – second algorithm

Finally, we add the sum of previous warps:

```
        __syncthreads();

        if (tid >= 32) temp1 += temp[tid/32 - 1];
        ...
}
```

Global scan – first algorithm

We can take two approaches to perform a global scan.

The first is:

use one kernel to do a local scan and compute partial sum for each block.

Use host code to perform a scan of the partial sums.

Use another kernel to add sums of preceding blocks

Global scan – second algorithm

The second alternative is to perform the whole operation using just one kernel call.

However, this needs the sum of all preceding blocks to add to the local scan values.

This presents a problem… blocks are not necessarily processed in order, so could end up in deadlock waiting for results from a block which doesn't get a chance to start.

The solution to this is to use atomic increments.

Global scan – second algorithm

The way to do this is to declare a global device variable

```
__device__ int my_block_count = 0;
```

and at the beginning of the kernel code use

```
__shared__ unsigned int my_blockId;
if (threadIdx.x==0) {
        my_blockId = atomicAdd( &my_block_count, 1 );
}
__syncthreads();
```

which returns the old value of my_block_count and increments it, all in one operation.

This gives us a way of launching blocks in strict order.

Global scan – second algorithm

In this approach to the global scan, the kernel code does the following:

- Get in-order block ID.
- Perform scan within the block.
- Wait until another global counter my_block_count2 shows that preceding block has computed the sum of the blocks so far.
- Get the sum of blocks so far, increment the sum with the local partial sum, then increment my_block_count2.
- Add previous sum to local scan values and store the results.

Global scan – second algorithm

```
// get global sum, and increment for next block

if (tid == 0) {
  // do-nothing atomic forces a load each time
  do {} while( atomicAdd(&my_block_count2,0)
                                < my_blockId );

  temp = sum;             // copy into register
  sum  = temp + local; // increment and put back
  __threadfence();    // wait for write completion

  atomicAdd(&my_block_count2,1);
                     // faster than plain addition
}
```

Scan operations – conclusions

Conclusion: this is all quite tricky!

Advice: best to first see if you can get working code from someone else (e.g. investigate Thrust library).

Don’t re-invent the wheel unless you really think you can do it better.

What have we learnt?

In this lecture we have learnt about different types of warp shuffle instructions and why they are useful.

We have used warp shuffles to construct different memory access patterns.

We have looked at the reduction algorithm within a warp, then thread block and finally a more generic global reduction algorithm.

Finally we considered the scan algorithm and it's implementation on a GPU.