Lecture 4: warp shuffles, and reduction / scan operations

Prof. Mike Giles

mike.giles@maths.ox.ac.uk

Oxford University Mathematical Institute

Warp shuffles

Warp shuffles are a faster mechanism for moving data between threads in the same warp.

There are 4 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)

Warp shuffles

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

- `mask` controls which threads are involved — usually set to `-1` or `0xffffffff`, equivalent to all 1's
- `var` is a local register variable (int, unsigned int, long long, unsigned long long, float or double)
- `delta` is the offset within the warp – 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);
```

- defined similarly

Warp shuffles

```
T __shfl_xor_sync(unsigned mask, T var, int laneMask);
```

- an XOR (exclusive or) operation is performed 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")

- a "butterfly" type of addressing, very useful for reduction operations and FFTs

```
T __shfl_sync(unsigned mask, T var, int srcLane);
```

- copies data from `srcLane`

Warp shuffles

Very important

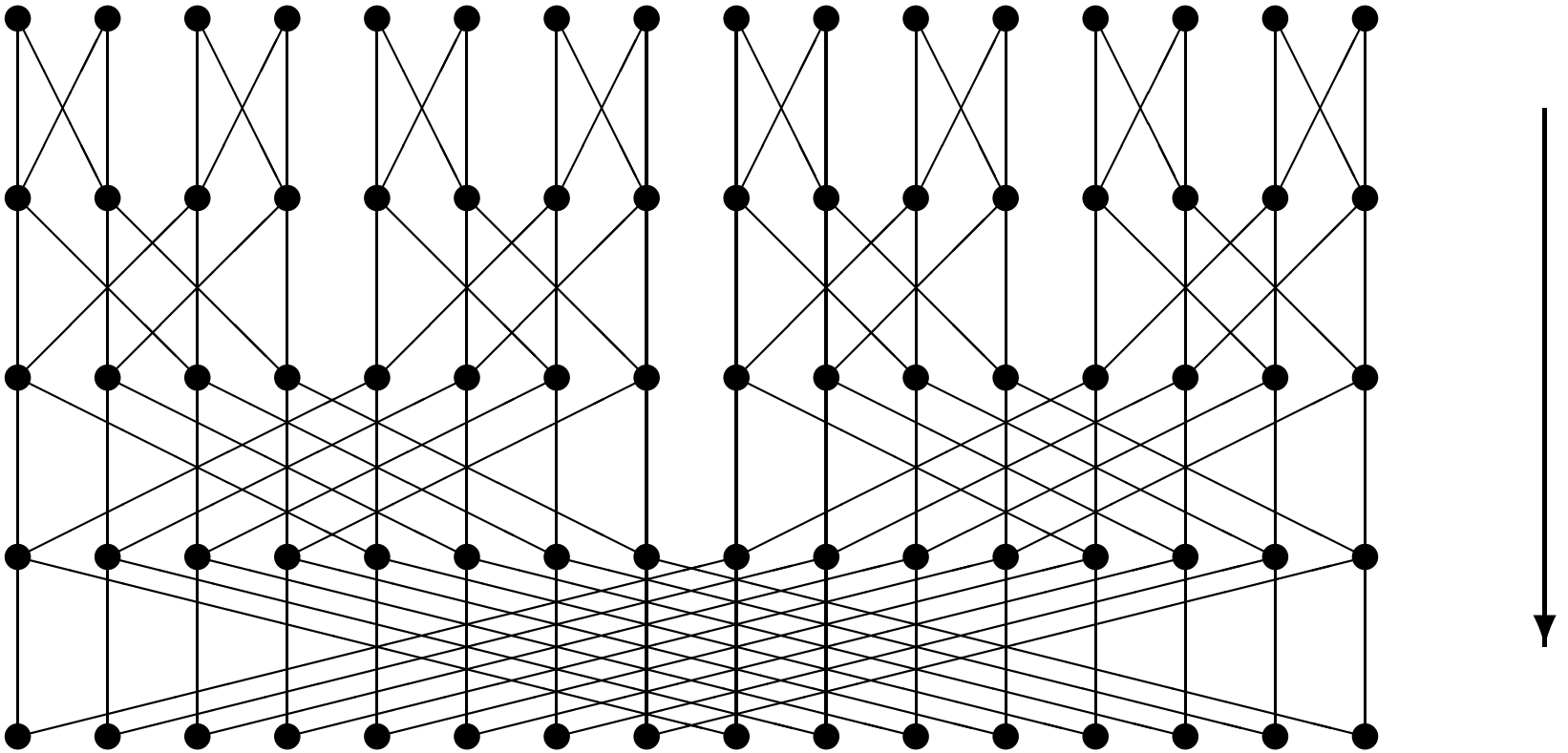
> Threads may only read data from another thread which is actively participating in the shuffle command. If the target thread is inactive, the retrieved value is undefined.

This means you must be very careful with conditional code.

Warp shuffles

Two ways to sum all the elements in a warp: method 1

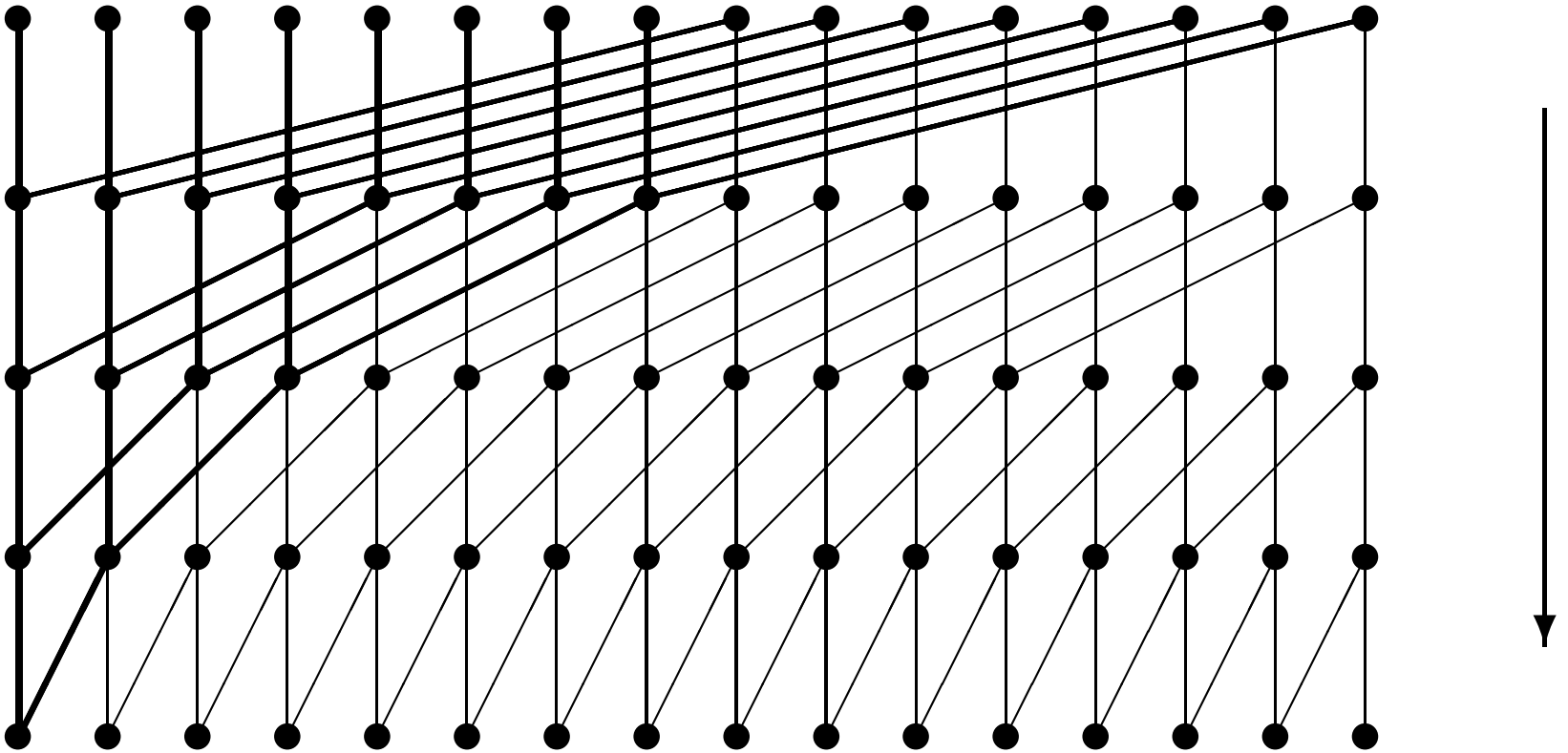
```
for (int i=1; i<32; i*=2)
  value += __shfl_xor_sync(-1, value, i);
```



Warp shuffles

Two ways to sum all the elements in 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:

- 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

Reduction

Other common reduction operations are to compute a minimum or maximum.

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.)

Approach

Will describe things for a summation reduction – the extension to other reductions is obvious

Assuming each thread starts with one value, the approach is 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

I'll look at each of these stages in turn

Local reduction

The first phase is contructing a partial sum of the values within a thread block.

Question 1: where is the parallelism?

“Standard” summation uses an accumulator, adding one value at a time $\Longrightarrow$ sequential

Parallel summation of N values:

- first sum them in pairs to get $N/2$ values
- repeat the procedure until we have only one value

Local reduction

Question 2: any problems with warp 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

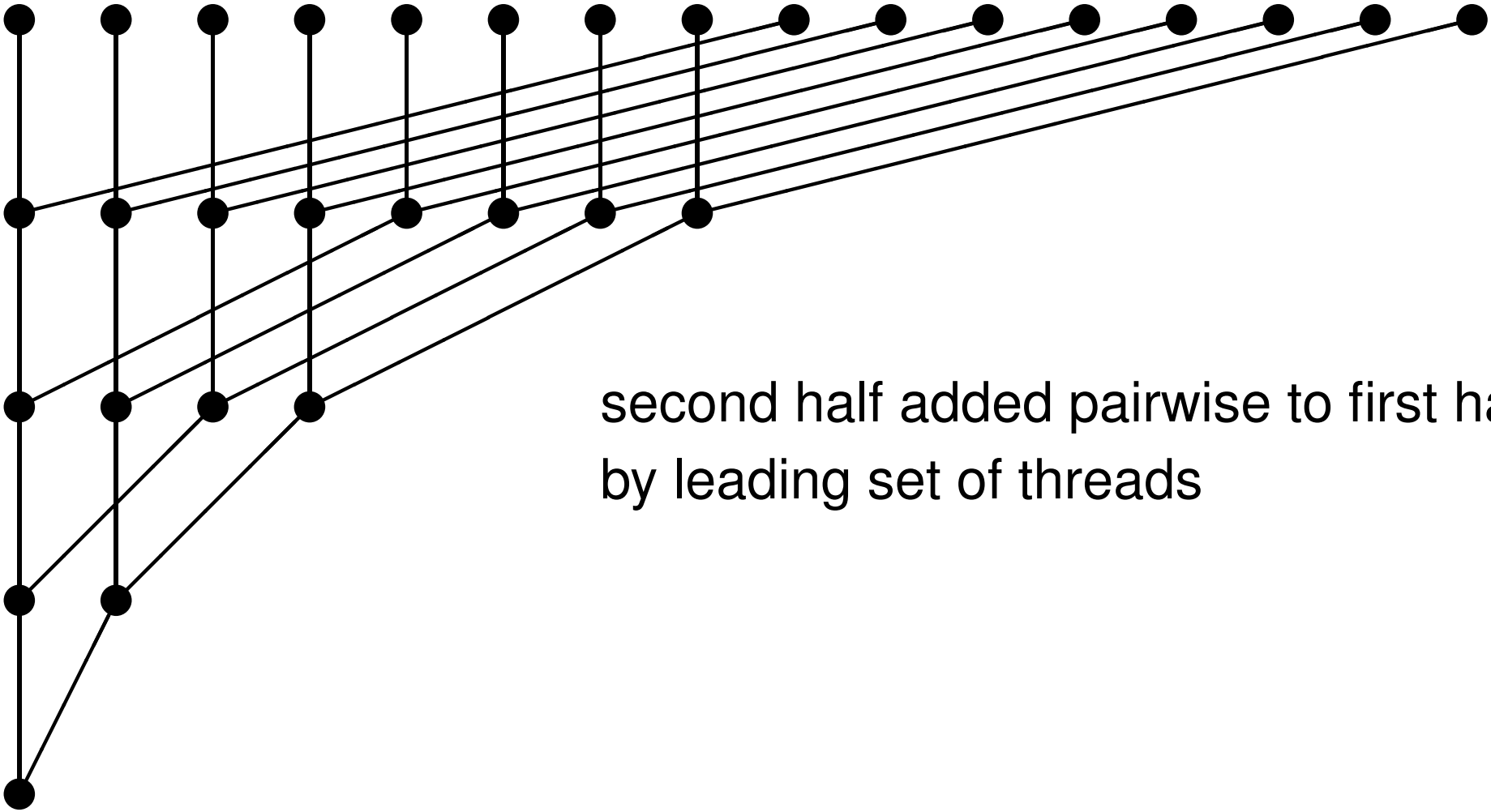
Question 3: where should data be held?

Threads need to access results produced by other threads:

- global device arrays would be too slow, so use shared memory
- need to think about synchronisation

Local reduction

Pictorial representation of the algorithm:



second half added pairwise to first half
by leading set of threads

Local reduction

```
__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];
}
```

Local reduction

Note:

- use of dynamic shared memory – size has to be declared when the kernel is called
- use of `__syncthreads` to make sure previous operations have completed
- 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 reduction: version 1

This version of the local reduction puts the partial sum for each block in a different entry in a global array

These partial sums can be transferred back to the host for the final summation – practical 4

Global reduction: version 2

Alternatively, can use the atomic add discussed in the previous lecture, and replace

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

by

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

Global reduction: version 2

More general reduction operations could can use the atomic lock mechanism, also discussed in the previous lecture:

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

by

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

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

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

Scan operation

Given an input vector $u_i,\ i = 0, \ldots, I-1$, the objective of a scan operation is to compute

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

Why is this important?

- a key part of many sorting routines
- arises also in particle filter methods in statistics
- related to solving long recurrence equations:

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

- a good example that looks impossible to parallelise

Scan operation

Before explaining the algorithm, here's the "punch line":

- some parallel algorithms are tricky – don't expect them all to be obvious
- check NVIDIA's sample codes, check the literature using Google – don't put lots of effort into re-inventing the wheel
- the relevant literature may be more than 30 years old – back to the glory days of CRAY vector computing and Thinking Machines' massively-parallel CM5

Scan operation

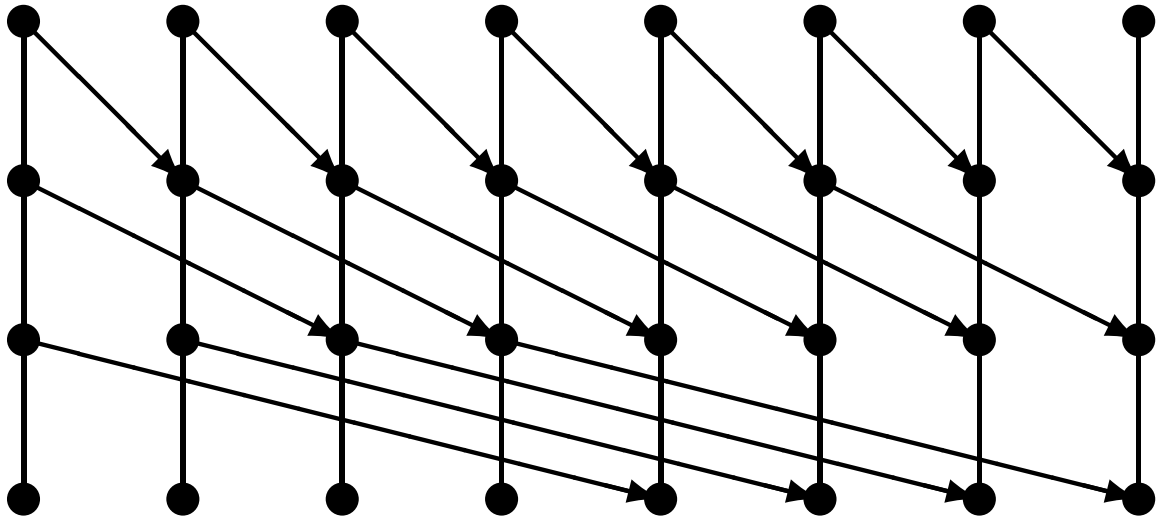
Similar to the global reduction, the top-level strategy is

- perform local scan within each block
- add on sum of all preceding blocks

Will describe two approaches to the local scan, both similar to the local reduction

- first approach:
  - very simple using shared memory, but $O(N \log N)$ operations
- second approach:
  - more efficient using warp shuffles and a recursive structure, with $O(N)$ operations

Local scan: version 1



- after n passes, each sum has local plus preceding $2^n - 1$ values
- $\log_2 N$ passes, and $O(N)$ operations per pass
  $\Longrightarrow$ $O(N \log N)$ operations in total

Local scan: version 1

```
__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: version 1

Notes:

- increment is set to zero if no element to the left
- both `__syncthreads();` are needed

Confession: my most common CUDA programming error is failing to use a `__syncthreads();` when needed

Local scan: version 2

The second version starts by using warp shuffles to perform a scan within each warp, and store 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: version 2

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: version 2

Finally, we add the sum of previous warps:

```
  __syncthreads();

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

  ...
}
```

Global scan: version 1

To complete the global scan there are two options

First alternative:

- use one kernel to do 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: version 2

Second alternative – do it all in one kernel call

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

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.

Solution: use atomic increments to create an in-order block ID

Global scan: version 2

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: version 2

In the second 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: version 2

```
// 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 operation

Conclusion: this is all quite tricky!

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

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

Recurrence equation

Given s_n, u_n, want to compute v_n defined by

$$v_n = s_n \; v_{n-1} + u_n$$

(Often have

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

with $0 < \lambda_n < 1$ so this computes a running weighted average, but that's not important here.)

Again looks naturally sequential, but in fact it can be handled in the same way as the scan.

Recurrence equation

Starting from

$$\begin{aligned} v_n &= s_n\ v_{n-1} + u_n \\ v_{n-1} &= s_{n-1}\ v_{n-2} + u_{n-1} \end{aligned}$$

then substituting the second equation into the first gives

$$v_n = (s_n s_{n-1})\ v_{n-2} + (s_n u_{n-1} + u_n)$$

so $\quad (s_{n-1}, u_{n-1}),\ (s_n, u_n) \longrightarrow (s_n s_{n-1},\ s_n u_{n-1} + u_n)$

Repeat at each level of the scan, eventually getting

$$v_n = s'_n v_{-1} + u'_n$$

where v_{-1} represents the last element of the previous block.

Recurrence equation

When combining the results from different blocks we have the same choices as before:

- store s', u' back to device memory, combine results for different blocks on the CPU, then for each block we have v_{-1} and can complete the computation of v_n
- use atomic trick to launch blocks in order, and then after completing first phase get v_{-1} from previous block to complete the computation.

Similarly, the calculation within a block can be performed using shuffles in a two-stage process:

1. use shuffles to compute solution within each warp
2. use shared memory and shuffles to combine results from different warps and update solution from first stage

Key reading

CUDA Programming Guide:

- Appendix B.22: warp shuffle instructions
- Appendix B.21: new warp reduction instruction – this is only for integers currently, and I have not experimented with it