Introduction to CUDA 2

Lecture 8

István Reguly
reguly.istvan@itk.ppke.hu

Recap

- The CUDA execution model has two levels of parallelism:
  - A “block” of threads (up to 1024 per block)
  - A number of blocks

```
my_kernel<<<num_of_block, threads_per_block>>>(….)
```

- Threads in the same block assigned to the same SM, run at the same time, and can share resources
- Threads in different blocks may not run at the same time
  - A thread in block N may have completed execution before a thread in block M starts

Shared memory

- In a kernel, the prefix `__shared__` as in:
```
__shared__ int x_dim;
__shared__ float x[128];
```
  declares data to be shared between all the threads in a thread block – any of them can read or write it
- **Only** between thread in the same block – each block has its own separate chunk of shared memory
- Benefits:
  - Essential for operations requiring communication between threads
  - Useful for data re-use (can be used as a cache!)
  - Reduces use of registers when a variable has same value for all threads

Shared memory

- Not all threads in the block execute simultaneously
  - If thread N need to read the value that thread M wrote, we need to ensure ordering
- Thus, we need synchronisation to ensure correct use of shared memory for communication
- __syncthreads()
- Inserts a barrier; no thread/warp is allowed to proceed beyond this point until the rest have reached it

Shared memory

- So far the examples have shown static shared memory arrays
- Can also create dynamic shared-memory arrays

```
extern __shared__ float *a;
...
kern1<<<blocks,threads,shared_bytes>>>(...)
```

- Such as when block size is not fixed and need a shared memory array with one value per thread

Synchronisation

- Already introduced `__syncthreads()` – this forms a barrier: all threads wait until everyone have reached this point
- When writing conditional code, we must be careful to make sure that all threads do reach the `__syncthreads` call
- Otherwise, we end up with a deadlock...

Shared memory & syncthreads

```
__global__ void kernel(int *a) {
  __shared__ int local[128]; //Assume thread block size of 128
  local[threadIdx.x] = //Always index shared memory with local index
          a[threadIdx.x+blockIdx.x*blockDim.x]; //But global arrays
                                                      //with global index
  __syncthreads(); //make sure all threads finished the load
  if (threadIdx.x < blockDim.x-1)
    //before any thread reads a value written by another thread
    int difference = local[threadIdx.x+1]-local[threadIdx.x];
}
```

Synchronisation

- Extra capabilities:
  - `int __syncthreads_count(predicate)`
    counts how many of the predicates are true
  - `int __syncthreads_and(predicate)`
    returns non-zero if all predicates are true
  - `int __syncthreads_or(predicate)`
    returns non-zero if any predicates are true

Atomic operations

- Occasionally, an application needs threads to update a counter in shared or global memory

```
__shared__ int count;

...

if ( ... ) count++;
```

- There is a problem if two or more threads try to do it at the same time

Atomic operations

time	thread 0	thread 1	thread 2	thread 3
	read	read	read	read
	add	add	add	add
	write	write	write	write

Atomic operations

Atomic memory transaction

thread 0 thread 1 thread 2 thread 3

read/add/write

read/add/write

time

read/add/write

read/add/write

Atomic operations

- Several different atomic operations are supported, mostly integers and floats:
  - Addition (integers and 32/64-bit floats)
  - Minimum/maximum
  - Increment/decrement
  - Exchange/compare-and-swap

Atomic operations

- Atomic add

```
float atomicAdd(float *address, float increment);
```

- Adds increment on top of *address, and returns its old value

Exercise

- See reduction.cu
- Create an array of size 1<<30 (~1 billion), fill it with numbers 1..1<<30
- Calculate the sum of these numbers in different ways
    - Have each thread atomically increment a single value in global memory
    - Have each thread read its value into shared memory, and thread 0 of the thread block sum it up, then atomically increment a single value in global memory
        - Where do we need synchronization?

Warp divergence

- Threads are executed in warps of 32, with all threads in the warp executing the same instruction at the same time
- What happens if different threads in a warp need to do different things?

```
if (x<0.0)
  z = x-2.0;
else
  z = sqrt(x);
```

- Called warp divergence in CUDA

Warp divergence

- GPUs have predicated instructions, which are only carried out if a logical flag is true:

```
p:  a = b + c;  // computed only if p is true
```

- In the previous example, all threads compute the predicate, then two predicated instructions

```
     p = (x<0.0);
 p:  z = x-2.0;      // single instruction
!p:  z = sqrt(x);
```

Warp divergence

- Note that sqrt(x) would normally produce NaN when x<0, but it's not really executed when x<0 so there is no problem

- Execution cost is **sum** of both branches!
  - Potentially a large loss of performance: sqrt is really expensive!

Warp divergence

- Another example:

```
if (n>=0)
   z = x[n];
else
   z = 0;
```

- x only accessed if n>=0
- Don’t have to worry about illegal memory accesses

Warp divergence

- Warp divergence can lead to a big loss of parallel efficiency – one of the first things to look for in a new application
- In the worst case, effectively lose a factor of 32x if one thread needs expensive branch

Reductions

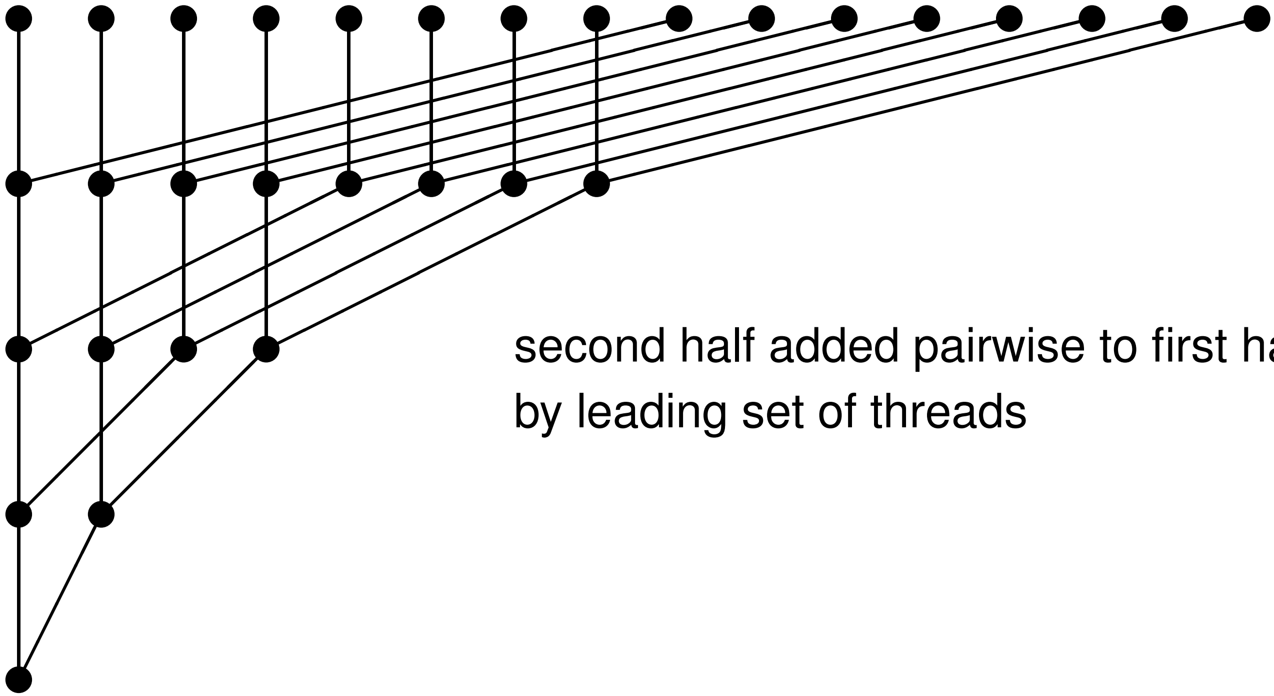
- Common reduction operations are to compute the sum, the minimum, or the maximum
- Key requirements for a reduction operator $\circ$ are:
  - Commutative: $a \circ b = b \circ a$
  - Associative: $a \circ (b \circ c) = (a \circ b) \circ c$
- Together they mean that the elements can be re-arranged and combined in any order

Approach to reduction

- We describe the algorithm for summation reduction, but the generalisation is obvious
- Assuming each thread starts with one value, the approach is to:
  - First add the values within each thread block and form a partial sum
  - Then add together the partial sums from all of the blocks

Shared memory reductions

Pictorial representation of the algorithm:



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

Local reduction

```
__global__ void reduction(float *g_odata, float *g_idata)
{
    // dynamically allocated shared memory
    extern  __shared__  float temp[];

    // global index, and index in block
    int i   = threadIdx.x + blockDim.x*blockIdx.x;
    int tid = threadIdx.x;

    // first, each thread loads data into shared memory
    temp[tid] = g_idata[i];

    // next, we perform binary tree reduction
    for (int d = blockDim.x>>1; d > 0; d >>= 1) {
      __syncthreads();  // ensure previous step completed
      if (tid<d)  temp[tid] += temp[tid+d];
    }

    // finally, first thread puts result into global memory
    if (tid==0) atomicAdd(g_odata, temp[0]);
}
```

Local reduction

- Note:
- Use of dynamic shared memory
- Use of `__syncthreads` to make sure previous operations have completed
- First thread outputs final partial sum into specific place for that block

Exercise

- Extend previous exercise to implement a binary tree reduction in shared memory
- Make it work across multiple blocks
  - Compare with atomic reduction of block sums
- Make it work for any array length

Warp shuffles

- Mechanism for moving data between threads in the same warp, without using shared memory

- Works for 32-bit data

Warp shuffles

- There are four variants
- `__shfl_up`
  copy from a lane with lower ID relative to caller
- `__shfl_down`
  copy from a lane with higher ID relative to caller
- `__shfl_xor`
  copy from a lane based on bitwise XOR of own ID
- `__shfl`
  copy from an indexed lane

Warp shuffles

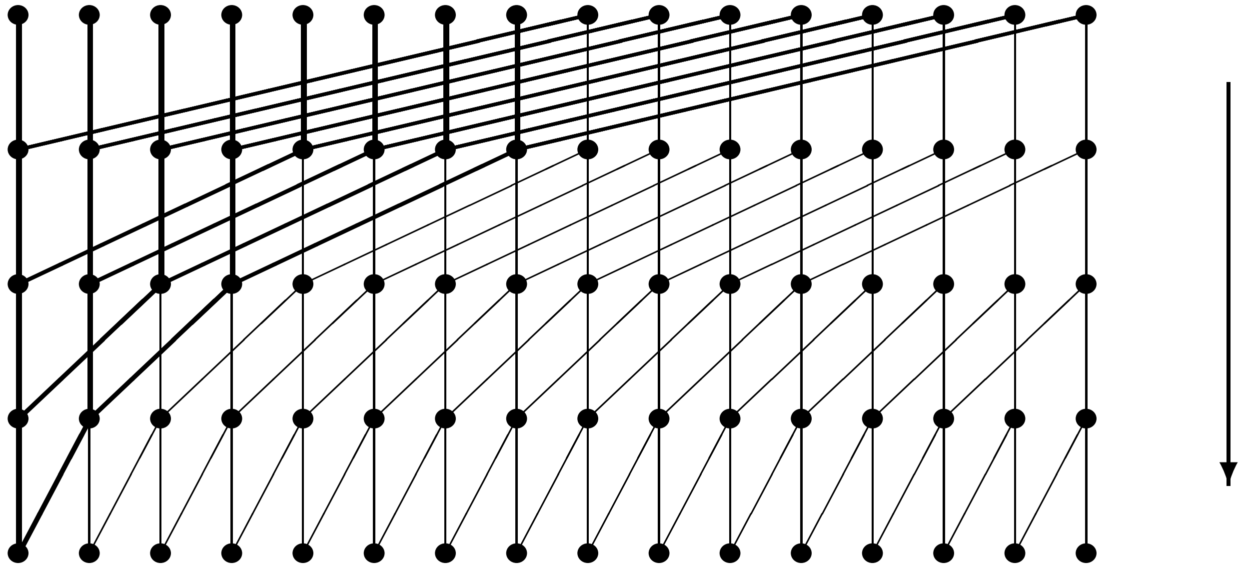
```
int __shfl_up(int var, unsigned int delta)
```

- Var is a local register variable
- Delta is the offset within the warp – if it does not exist, then value is taken from the current thread

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(value, i);
```



Atomic operations

- Compare-and-swap

```
int atomicCAS(int* address,int compare,int val);
```

- If compare equals old value stored at address, then val is stored instead
- In either case, it returns the value of old
- Seems odd, but can be very useful for implementing locks for critical regions!

Global atomic lock

```
// global variable: 0 unlocked, 1 locked
__device__ int lock=0;

__global__ void kernel(...) {
  ...

  if (threadIdx.x==0) {
    // set lock
    do {} while(atomicCAS(&lock,0,1));

    ...

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

Global atomic lock

- Problem: when a thread writes data to device memory, the order of completion is not quite guaranteed, so global writes may bot have completed by the time the lock is unlocked:

```
if (threadIdx.x==0) {
  do {} while(atomicCAS(&lock,0,1));
  ...
  __threadfence(); // wait for writes to finish

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

Threadfence

- __threadfence_block()
  - Wait until all global and shared memory writes are visible to all threads in the block

- __threadfence()
  - Wait until all global memory writes are visible to all threads on the device

Exercise

- Let’s implement reductions in our laplace 2D code

Homework – Due May 12

- Create a CUDA version of the LBM code
  - Use the timed version as a baseline (not the MPI one)
  - Do not parallelize across f (0->9) – in may kernels you can't