COM2039: Parallel Computing

Lab Class 5

Fixing the Race Condition in the Scan Kernel

Contents

1 Introduction 1

2 Reduce .... **Error! Bookmark not defined.**

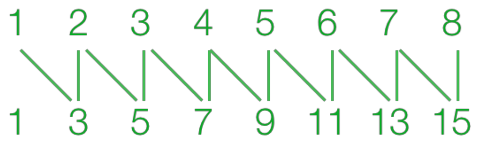
2.1 Solution .... **Error! Bookmark not defined.**

3 Scan .... **Error! Bookmark not defined.**

3.1 Partial Solution .... **Error! Bookmark not defined.**

1 Introduction

As mentioned in the solution sheet, the Scan kernel I provided you there has a potential race condition in it. Let's just see how this is arising first, before we look at an updated version of the kernel that eliminates this race condition.
Take a look at this illustration of the first step in a Hillis and Steels scan:



Each thread, apart from thread #0, is going to add to elements together. For example, thread #2 is going to add 2+3 to get 5. However, it is just possible that before it reads to value "2", thread #1 will have already updated it to "3". So, thread #2 will get the answer "6" instead of the answer "5".
The problem is that in the Kernel I gave you, we have a single array in shared memory and there is no synchronisation of the additions across threads within one iteration of the for loop.
The way to fix this is to use a "double buffered" solution. That is, on one iteration we read from one buffer (array in shared memory) and write into a second buffer. We can then syncthreads and progress to a second iteration of the for loop. But this time we read from the second buffer and write into the first. That way, we can be sure that we retain coherence of the data as we run through the steps of the Hillis and Steele scan.

2 Doubled Buffered Hillis and Steele Kernel

This should be a straight swap. But note that:

1. It does handle input array sizes that are not exact multiples of the block size. So, make sure that you calculate the grid size correctly (you need to round up to the nearest integral value of array_size/block_size, not down).
2. It uses the input array size, N. So replace the declaration on N in the main() method with #define N= 25600 (or whatever value you want to use) in the header of your .cu file.

The Kernel definition is on the next page. I have inserted comments to (hopefully) explain what is going on.

```
__global__ void scanKernel(int n, float *idata) {

  int thIdx = threadIdx.x + blockIdx.x * blockDim.x;
  int tid = threadIdx.x;

  // We need to create two buffers to avoid the race condition - read from one and write into the other
  __shared__ float temp[BLOCK_SIZE];
  __shared__ float temp2[BLOCK_SIZE];

  // declare a boolean to identify which of the two buffers we are currently reading from
  bool tempSelector = true;

  // each thread reads one data item into the first buffer in shared memory
  temp[tid] = idata[thIdx];
  __syncthreads();

  for (int offset = 1; offset < n; offset *=2) {
    // Let's also make sure that threads are only working on array elements that have data
    if (tid >= offset && thIdx < N)
    {
      if (tempSelector) // for odd loop numbers, read from first buffer into second
      {
        temp2[tid] = temp[tid] + temp[tid - offset];
      }
      else // for even loop numbers, read from second buffer into first
      {
        temp[tid] = temp2[tid] + temp2[tid - offset];
      }
    }

    // We also need to make sure all the unmodified values are copied between the two buffers
    if (tid < offset)
    {
      if (tempSelector) temp2[tid] = temp[tid];
      else temp[tid] = temp2[tid];
    }

    // and update the condition
    tempSelector = !tempSelector;
    // then make sure all threads have finished before going round the loop again
    __syncthreads();
  }
  // we need to make sure we output the value from the correct buffer
  if (tempSelector)
  {
    idata[thIdx] = temp[tid];
  }
```

```
   else
   {
      idata[thIdx] = temp2[tid];
   }
}
```