Theodore Jagodits / CS 677 / HW #2 Report /

"I pledge my honor that I have abided by the Stevens Honor system"

1.4

a)

I start off with 256 threads, which equals to log2(256) = 8 thread synchronizations within my thread block. This is because I add the elements from global into the shared memory in the first computation, so I have only 256 elements to do the algorithm on, which halves every time until the stride >= 1. Every time it halves is when I must sync the threads.

b)

The minimum amount of operations needed is just 1. Since each thread loads from global and stores into shared, which would count as one operation. Following that, the maximum amount would be log2 of 512 which is 9. The thread 0 has to compute for every iteration which halves every time.

The average is (256+128+64+32+16+8+4+2+1) /256 = 1.99 operations per thread. Each for loop of the operation loses half of threads that actually do work, so halve the number of threads and increment the number of operations every turn.

4.

a)

There is a total of 256 * 1024 threads at the start. As there are vector n blocks of element n threads. = 262,144

b)

Each thread does 2 two loads and then do one store each. They access vectors A and B on line 21 and then store once on line 30.

c)

In each block, we access and store per thread accumResult, so that is already 256. Then we access in the for loop, which goes to zero and halves from 256/2 = 128. This brings the total of accesses to 256 + 128 … until the stride is 0 then we stop. Then each thread does the global access it puts the result of that accumResult[0] into the global memory. In the for loop itself there are 3 memory accesses. So, the total per block is 256 for each thread plus 256 at the end. Then (128 + 64 + 32 + 16 + 8 + 4 + 2 + 1)*3 = 765 access per block per for loop. Total = (256 + 256 + 765) = 1277.

d)

It doesn't look like there are any shared memory conflicts. In line 21, the thread accesses by its own id so there shouldn't be any problems there since its unique. In line 27 the thread accesses the by own id and adds together the result so no problems there. The last line 30 would have issues with global write because every thread in the block would be pushing the first value in shared memory to global. But that isn't a problem with shared.

e)

The for loop on line 23 will not have any divergence itself since each thread will be executing that part. However, the if statement inside will have divergence whether the stride is part the thread that is being added. On the first iteration the stride is 128 so only 128 threads are active and the other 128 threads in the block are inactive which equals to 4 warps. This repeats itself until the stride is 0 so log2(256) = 8. If the for loop runs 8 times, then we would have 128 then 64 …then 1 active each step. We would have block_size – stride inactive each step. Total = (256-128) + (256-64) + (256-32) + (256-16) + (256-8) + (256-4) + (256-2) + 255 = 1791 threads inactive in total which is equal to the equation nlog2(n)-n-1.

f)

On line 31, each thread does a global memory access that does by block ID. You could significantly reduce this access by putting an if statement at the end before line 31 that says

```
If(threadID < 1)
        d_C[blockId] = accumResult[0];
```

In addition, we could get rid of another access in by getting rid of vector bases in lines 17 and 18. Instead of setting a memory address for the vector base of the blockID and calling from there. We can just access by calling d_B[blockID*blocksize + threadID]. This reduces the bandwith of calling by mathematically calling the specific part of the address, rather than calculating and accessing the vector base and then accessing in the vector base by the thread ID.

I can eliminate the bandwith needed in access d_A and d_B arrays by 2 per thread which would increase the access and also eliminate 255 global accesses in line 30.