MP5.1

1. (2pt) How many bytes are read from global memory by the reduction kernel? How many bytes
are written to global memory? The length of input array is N, the block size is BLOCK_SIZE, each block reads 2 × BLOCK_SIZE float elements from global memory(possibly except the last block). Please give answers separately for reads and writes and explain your answer for full credit. (Assume arbitrary input array length which may or may not fit in one block)

Solution:
Total bytes read from global memory: 4 × N. (**1pt** for answer and explanation)

Every block (possibly except the last block) will always read 2 × BLOCK_SIZE number of float element from global memory, then calculate the sum of these 2×BLOCK_SIZE number of elements, and save that block sum (of float type) to global memory, and the last block may read more than 0 but maybe less than or equal to 2 × BLOCK_SIZE number of float element from global memory, then calculate the sum of these elements, and save block sum (of float type) to global memory. Every input element will be read from memory exactly once. And the total number of global memory read is N. Each float has 4 bytes, therefore the total bytes read from global memory is 4 × N.

Total bytes written to global memory: 4 × ceil(N/((float)(2 × BLOCK_SIZE))).(**1pt** for answer and explanation)

Every block including the last block will calculate one float type sum of all input elements of the current block, and write that block sum to global memory. The total number of global memory writes is equal to the total number of blocks, which is ceil(N/((float)(BLOCK_SIZE × 2))). Each float has 4 bytes, therefore the total bytes written to global memory is 4 × ceil(N/((float)(2 × BLOCK_SIZE))).

2. (2pt) How many times does a single thread block synchronize in the reduction kernel? Assume
each thread reads 2 floating point elements. In other words, each thread block reads 2 × BLOCK_SIZE floating point elements from global memory. BLOCK_SIZE
is a power of 2. Give the answer based on BLOCK_SIZE (=blockDim.x) and explain your answer for full credit.

Solution:
We accept three answers: log(2, BLOCK_SIZE), log(2, BLOCK_SIZE) + 1, log(2, BLOCK_SIZE) + 2. (1pt for answer)
(1pt for explanation) The __syncthreads() in the for loop is used to make sure loading data are all complete for the block before doing computation. Each level of addition shrinks the remaining

number of elements to half of the number before this level of addition, and before doing addition in each level we need to do block synchronization. The number of input elements is 2 × BLOCK_SIZE including ghost elements. So there will be log(2,BLOCK_SIZE) number of levels of additions to reduce them to one block sum.
Some students have one more __syncthreads() right before writing back results to global memory to make sure that all the computations in the thread block are finished before writing result back; and some students have one more __syncthreads() just before we start reduction calculation iterations. These two __sync-threads() are not always strictly needed in the reduction kernel. But we didn't ask for the minimum number of __syncthreads() needed. Based on students' implementations we accept the three answers above.

3. (1pt) Is it possible to get different results from running the serial version (CPU) versus the parallel version (GPU) of reduction? Explain your answer for full credit. (Hint: the type of our input array is floating-point)

Solution:
Yes. The serial version and the parallel version of reductions have different orders of summing up elements from the input array, they may have different rounding errors, which may lead to slightly different calculation results.

We don't accept explanations that simply say "GPU reduction is more accurate than CPU reduction" or "CPU reduction is more accurate than GPU reduction".

4. (2pt) There are two styles of indexing for reduction. For the simple reduction kernel with simple
thread index to data mapping, assume each thread block reads 2 × BLOCK_SIZE floating point elements from global memory. For the improved reduction kernel, where we always compact the partial sums into the first locations in the partialSum[] array, each thread block reads 2 × BLOCK_SIZE elements from global memory. If the block size is 1024 and warp size is 32, for each reduction style, what is the iteration when a thread block contains at least one warp with control divergence? Assume we count the first iteration as Iteration 1. Explain your answer for full credit.

Solution:
Iteration 2 for the simple reduction kernel. **(1pt for answer and explanation)**
Iteration 7 for the improved reduction kernel. **(1pt for answer and explanation)**

One of the drawbacks of using a simple reduction kernel is that in the first several iterations we have warp divergence in all warps except iteration 1.
By using an improved reduction kernel, we can eliminate warp divergence in the first several iterations. There are 1024/32=32 threads being active in the 6th iteration, which occupies exactly one warp. There are 1024/64=16 threads being active in the 7th iteration, which takes half a warp.

5. (1pt) For the simple reduction kernel (with simple thread index to data mapping, each thread block reads 2 × BLOCK_SIZE elements from global memory), if the input size is 1984, the block size is 1024 and warp size is 32, how many warps in a block will have divergence during the first iteration?
A. 32
B. 16
C. 1
D. 0
E. None of the above.
Solution: (D) **(1pt, explanation is not required)**
For boundary condition, the thread block will load 2048-64 elements. The entire last warp with 32 threads will be out of boundary. Therefore, the boundary condition doesn't cause any warp to have control divergence.
For reduction iteration, all threads are actively doing addition in the first iteration since each thread loads two elements. No warp will have divergence in the first iteration.

6. (2pt) How many floating point operations are being performed in your reduction kernel if the length of the input array is N? Explain your answer for full credit. Assume the input array is small but not too small which can just fit in one block(the input array won't fit in one block if we reduce BLOCK_SIZE to its half), N is not necessarily a power of 2, while BLOCK_SIZE is a power of 2.
A. 2^(ceil(log(2,N)))
B. 2^(ceil(log(2,N))) - 1
C. N-1
D. 2N
E. None of the above.

Solution: (B) **(2pt, explanation is required)**
Even if we perform the addition of N input elements in the order of reduction instead of serial order, we still perform floating point additions for (2 × BLOCK_SIZE − 1) times where 2 × BLOCK_SIZE is (input size + number of ghost elements), and ghost elements whose value are 0 also participate in float additions. Since the input array is small but not too small which can just fit in one block, we know 2×BLOCK_SIZE is equal to 2^(ceil(log(2,N))), which is the smallest 2's integer power that is larger than or equal to N. And we perform (2*BLOCK_SIZE-1) times of float additions, so we perform (2^(ceil(log(2,N))) - 1) number of float additions.