W6998-5 HW 4

Daniel Guo

April 17, 2023

Part A

Problem Q1

values per thread	time (s)
500	0.003128
1000	0.006423
2000	0.013014

Problem Q2

values per thread	time (s)
500	0.000601
1000	0.001188
2000	0.002347

Between these two, we see that coalesced memory reads drastically speeds up the performance by about 5x. This has to do with the reduced strain on the global memory bandwidth.

Problem Q3

matrix size	time (s)
256	0.000132
512	0.000807
1024	0.005851

Problem Q4

matrix size	time (s)
256	0.000095
512	0.000396
1024	0.002736

Between these two, we see computing four values also speeds up the performance. Furthermore, the speedup seems more significant for larger matrix sizes. This could make sense, since for larger matrices, the tradeoff of more threads and waiting for them to synchronize is not worth it, and it is more efficient to have fewer threads do more work each.

Problem Q5

Coalescing memory and unrolling loops both seem to always improve performance, so a good rule of thumb is to always do these as much as possible. Also, it seems it is not always the best for each thread to compute one value in the output, so a good rule of thumb is to give every thread "enough" work to do. In our case of matrix multiplication, this meant computing four values in the output instead of one. However, what is "enough" work is likely very problem and system dependent, so it would be good to do some benchmarking ahead of time to find the sweet spot.

Part B

For this section, the K values correspond to the following vector sizes in order to match the 256 block size requirement.

K	vector size
1	1000192
5	5000192
10	10000128
50	50000128
100	100000000

Problem Q1

Host.

K	Host time (s)
1	0.000527
5	0.003429
10	0.008623
50	0.049367
100	0.099582

Problem Q2

1 block with 1 thread.

K	time (s)
1	0.123697
5	0.601515
10	1.203512
50	6.018537
100	12.036772

1 block with 256 threads.

K	time (s)
1	0.004165
5	0.020743
10	0.041488
50	0.090408
100	0.169470

Sufficient blocks with 256 threads so total # threads is equal to vector size.

K	time (s)
1	0.000115
5	0.000520
10	0.001026
50	0.002557
100	0.002215

Problem Q3

1 block with 1 thread.

K	time (s)
1	0.123727
5	0.601493
10	1.203517
50	6.018562
100	12.036735

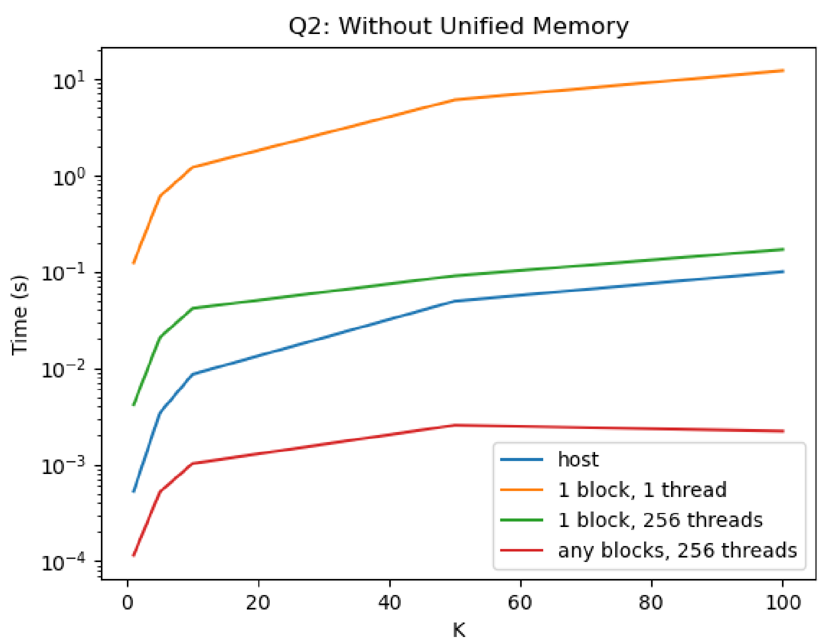
1 block with 256 threads.

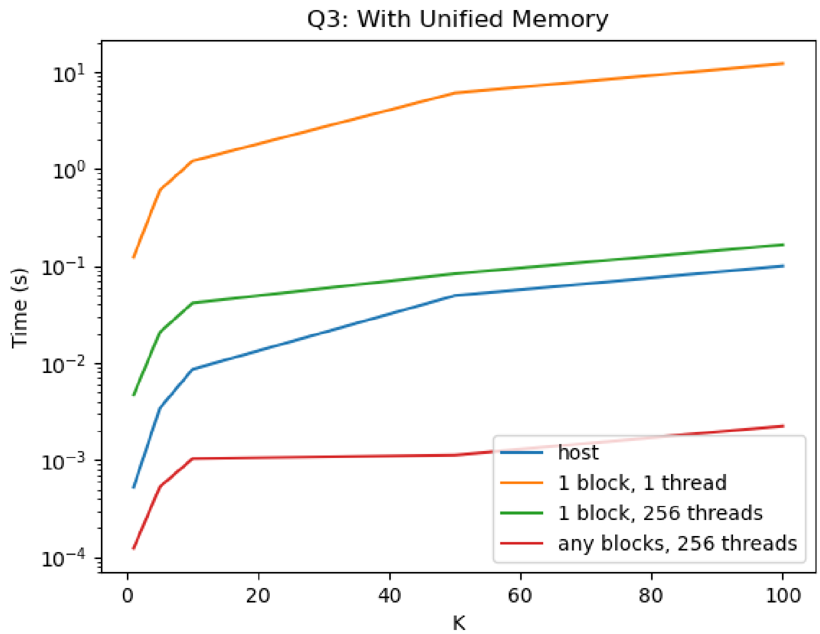
K	time (s)
1	0.004695
5	0.020723
10	0.041441
50	0.082914
100	0.164990

Sufficient blocks with 256 threads so total # threads is equal to vector size.

K	time (s)
1	0.000124
5	0.000531
10	0.001035
50	0.001128
100	0.002238

Problem Q4





It seems that using the unified memory doesn't change the performance by very much. Performance-wise, most of the results are as expected. Using the device with only one thread requires data loading time and computational time, so it should be the slowest. We see even with 256 threads, it is still slower than using host, likely still due to the data loading time. However, as the amount of data increases (and therefore the amount of computation increases), the gap between 256 threads and host is much smaller. Unsurprisingly, using as many block as necessary with 256 threads each so that each result element has its own thread is the fastest. It has the most threads and processing power, so it is expected to be fast.

Part C

The checksum for all three parts is 122756344698240.

Problem C1

For the naive method, my implementation took 0.169075 seconds.

Problem C2

For the tiling method, my implementation took 0.055715 seconds.

Problem C3

For the cudnn method, my implementation took 0.003372 seconds.

Overall, these performances are again as expected. Compared to the naive method, tiling gives approximately a 3x speedup (likely due to fewer global memory accesses), and cudann's method gives approximately a 50x speedup. This is probably because cudann is able to utilize better algorithms since it finds the fastest one given our specs.