Development Flow

The program works in the following way:

1. Each thread block computer TILEDIM_M * TILEDIM_N sized tile of C
2. We use shared memory blocks of TILEDIM_M*TILEDIM_K for A and TILEDIM_K*TILEDIM_N for B
3. Each thread would compute TILESCALE_M*TILESCALE_N values of C
4. Correspondingly, each thread loads TILESTEP_M*TILESTEP_AK and TILESTEP_BK*TILESTEP_N in shared memory
5. During loading, our function checks if we are not going out of bounds to deal with edge cases. getVal returns 0 if indices are out of bounds and setVal does nothing in the same case..
6. Each thread loads/writes are spaced with bx along x axis and by along y axis. Due to block size, this helps coalesce loads/stores for a warp

Optimizations

1. We first updated our setGrid function to make the gridsize dependent on our matrix dimension which improved the performance and was able to reach above 100 GFLOPS for n=256
2. We then added static shared memory and loaded one value per thread again. This gave us improvement and put us around 300 GFLOPS in n=256. For higher values also, it was around 300 GFLOPS.
3. We then shift to using dynamic shared memory and try to compute more values per thread, which after some optimization in config, we were able to reach our target performance. We wrote a python script to automatically write code based on our configuration, which made testing and optimization easier.

Ideas Tried

1. We tried using more threads/block and found out that they don't correspond to performance 1-1. This was due to their compute intensity being low due to higher number of threads and lower ILP, We found the sweet spot to be 256 threads.
2. The smaller matrix sizes performed well in different configurations compared to the higher sized matrices, this was because the general config used a very small number of thread blocks compared to SM's underutilizing the GPU.

Result

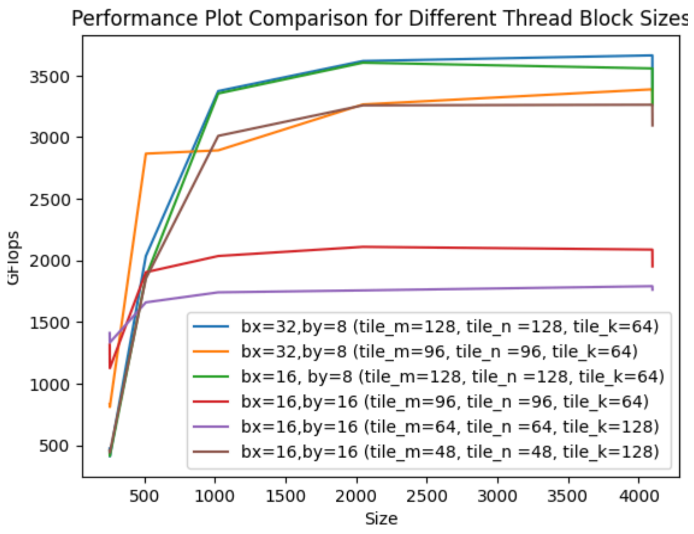
The following configurations are dependent on values of thread block sizes (bx,by) and tile sizes (tiledim_m, tiledim_n, tiledim_k) :-

- tilestep_n = tiledim_n/bx

- tilestep_m = tiledim_m/by
- tilestep_ak = tiledim_k/bx
- tilestep_bk = tiledim_k/by

For every configuration, we have written a code generator script that parses the my_types.h file and generates code for mmpy_kernel.cu, where loop unrolling for load/stores and computations is done as per that specific configuration.

Thread Block Size	**N=256**	**N=257**	**N=512**	**N=1024**	**N=2048**	**N=4096**	**N=4097**
bx=32,by=8 (tiledim_m=128, iledim_n =128, tiledim_k=64)	474.5	410.3	2035.9	**3374.4**	**3618.4**	**3664.1**	**3379.3**
bx=32,by=8 (tiledim_m=96, tiledim_n =96, tiledim_k=64)	835.5	811.0	**2866.7**	2893.4	3265.9	3388.7	3281.2
bx=16, by=8 (tiledim_m=128, tiledim_n =128, tiledim_k=64)	454.2	413.7	1867.0	3355.0	3604.7	3558.4	3262.5
bx=16,by=16 (tiledim_m=96, tiledim_n =96, tiledim_k=64)	840.8	805.0	2782.8	2976.0	3203.0	3343.9	3256.3
bx=16,by=16 (tiledim_m=64, tiledim_n =64, tiledim_k=128)	1317.6	1125.9	1904.6	2036.0	2110.4	2088.1	1951.9
bx=16,by=16 (tiledim_m=48, tiledim_n =48, tiledim_k=128)	**1411.3**	**1333.7**	1660.0	1741.2	1757.1	1791.1	1762.8
bx=32,by=16 (tiledim_m=128, tiledim_n =128, tiledim_k=64)	456.8	441.8	1852.7	3011.0	3258.8	3263.4	3095.7



Difference between the results for n=256 and n=257 as well as n=4096 and n=4097

For n=257 , a dip in performance is observed as compared to n=256. Similarly, for n=4097 , a dip in performance is observed as compared to n=4096. Since, for n = 4097 and n =257, the size of the grid increases and extra threads blocks are allocated. This means the number of threads also increases. But the number of useful points being computed per block and per thread are less. Hence, instruction throughput decreases and a dip in performance is observed. As per the best performing config for n=4096, the grid size increases from 1024 to 1089. Since every thread block computes 16384 points and the number of useful points being computed per thread and per block are less, there is a decrease in performance.

Choice of optimal thread block sizes for each N(matrix size - 256, 512, 1024, 2048, 4096).

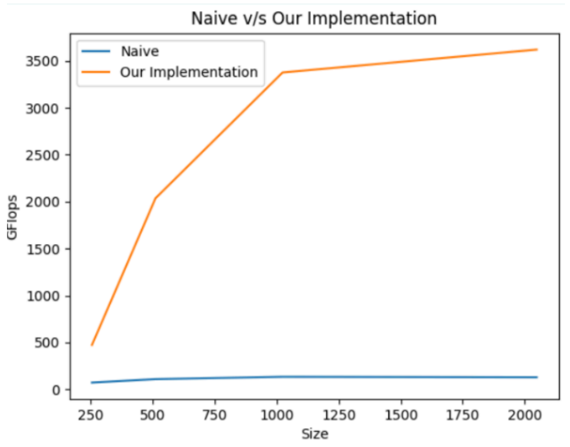
For n= 256, the optimal choice for thread block sizes and tiles sizes is bx=16,by=16, tiledim_m=48, tiledim_n =48 and tiledim_k=128. According to this configuration, more thread blocks are being used (i.e 36) and hence more SMs would be utilized. Other configs lead to less threads block being used, leaving SMs underutilized and hence leading to lower performance than the best configuration for n=256. For n=512, the optimal choice for thread block sizes and tiles sizes is bx=32,by=8, tiledim_m=96, tiledim_n=96 and tiledim_k=64. According to this configuration, we are using 36 SMs, causing better utilization of SMs and leading to higher performance. When comparing with the best config for n =256, the grid size becomes 64 for n=512. This leads to some thread blocks being stalled, causing a dip in performance. For n=1024, 2048 and 4096, the optimal choice for thread block sizes and tiles sizes is bx=32,by=8 tiledim_m=128, tiledim_n =128 and tiledim_k=64. In this configuration, every thread block and every thread per block computes more points, leading to higher performance. Upon running Nvidia Nsight for this config, we see the LSU pipeline is well utilized and compute & memory are well balanced. When comparing the best config for n =256 and n=512, we are computing a lesser total number of points per thread and per thread block, leading to lower performance.

Peak GF achieved and the corresponding thread block size for each matrix size

N	Peak GF	Thread Block Size
256	1411.3	bx=16,by=16 (tile_m=48, tile_n =48, tile_k=128)
512	2866.7	bx=32,by=8 (tile_m=96, tile_n =96, tile_k=64)
1024	3374.4	bx=32,by=8 (tile_m=128, tile_n =128, tile_k=64)
2048	3618.4	bx=32,by=8 (tile_m=128, tile_n =128, tile_k=64)
4096	3664.1	bx=32,by=8 (tile_m=128, tile_n =128, tile_k=64)

Compare best result with the naive implementation.

(GFLops)	**N=256**	**N=512**	**N=1024**	**N=2048**
NAIVE	73.7	110.2	135.3	130.1
OUR RESULT	474.5	2035.9	3374.4	3618.4

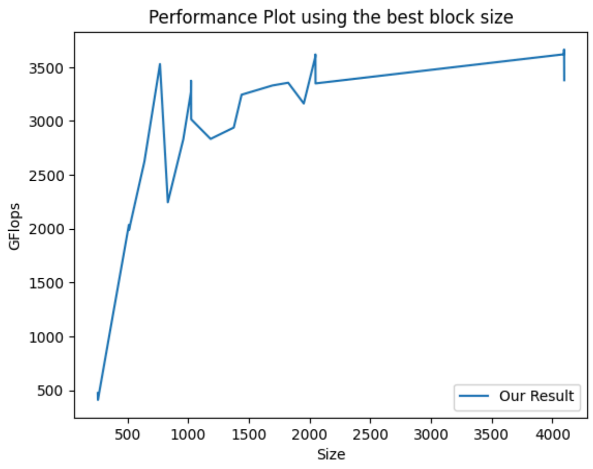


Our kernel shows a good supelinear performance improvement with larger matrix sizes upto a certain size by leveraging shared memory and performing warp-level optimizations. Whereas the naive implementation shows sublinear improvement in performance. The naive implementation exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of this device, hence implying all compute pipelines are under-utilized.

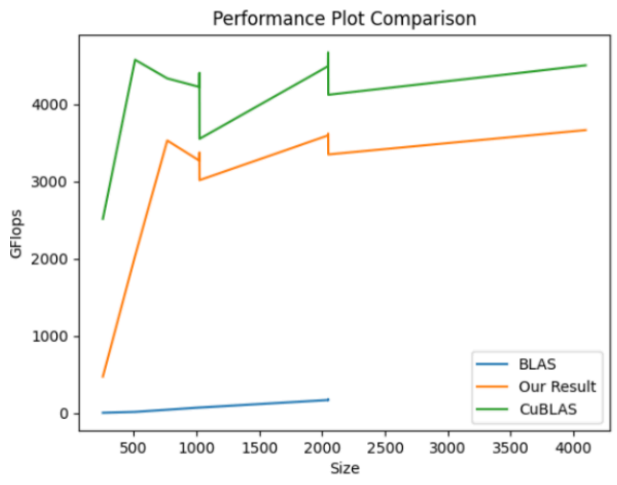
	N=256		**N=512**		**N=1024**		**N=2048**	
	Compute(%)	*SM Busy(%)*	*Compute(%)*	*SM Busy(%)*	*Compute(%)*	*SM Busy(%)*	*Compute(%)*	*SM Busy(%)*
NAIVE	10.21	9.06	13.76	10.22	17.07	10.89	17.71	10.11
OUR RESULT	8.68	67.92	35.00	68.47	71.02	68.32	81.55	68.46

Analysis

Comparing our results to the multi-core BLAS and cuBlas results in the table below.



N	BLAS (GFlops)	CuBLAS	Your Result (GFlops)
256	5.84	2515.3	474.5
257	-	-	410.3
512	17.4	4573.6	2035.9
513	-	-	1988.1
640			2623.6
768	45.3	4333.1	3529.2
833			2245.7
960			2831.0
1023	73.7	4222.5	3269.4
1024	73.6	4404.9	3374.4
1025	73.5	3551.0	3016.4
1185			2833.9
1376			2939.7
1440	-	-	3244.3
1695	-	-	3330.4
1824	-	-	3355.7
1952			3162.8
2047	171	4490.5	3594.8
2048	182	4669.8	3618.4
2049	175	4120.7	3349.4
4095	-	-	3620.1
4096	-	4501.6	3664.1
4097	-	-	3379.3



Compared to BLAS which shows sublinear improvement in performance, our curve shows a supelinear improvement in performance until a certain point. Our kernel shows better scalability with larger problem sizes due to its ability to efficiently utilize thousands of parallel threads. Our kernel shows a good performance improvement with larger matrix sizes by leveraging shared memory and performing warp-level optimizations. In our curve, the performance improvement is seen but peak performance is not achieved until a certain size, this could be because the performance is bounded by memory bandwidth. We achieve peak performance, upon reaching that size. After that point, the performance improvement above peak performance is not achieved, indicating it is compute bound. This indicates that with larger n, the computation starts to utilize memory bandwidth more efficiently and at smaller n, the overhead of memory access is relatively higher compared to the computational workload.

Compared to cuBLAS, the shape of our curve is almost similar, so we can theorize that our solution and cuBLAS have the same thread geometry. Despite similarity in shape of the curves, our solution performs somewhat worse than cuBLAS. The reason for it could be adding another warp level layer for matrix product which we have not currently implemented in our solution, which may lead to better instruction level parallelism.

Explanation regarding unusual dips, peaks or irregularities in performance with varying n.

As the value of n increases until a certain size, the performance improvement is seen but peak performance is not achieved indicating it is bounded by memory bandwidth. At n =768, we achieve near peak performance. After this point as the value of n increases, performance improvement above peak performance is not achieved, thus indicating performance is compute bound . The peak performance is achieved at values where n equal to powers of 2 greater than 768. This is because the thread and block organization becomes more efficient, aligning with hardware configurations and optimizing memory access patterns within shared memory. Dips in performance are seen at n =1025, 2049 and 4097, indicating that the instruction throughput decreases because extra blocks are being allocated due to an increase in grid size. This is due to the decrease in the number of useful points being computed per block and per thread, causing dip in performance.

Q5.a)

Zhe Jia, Marco Maggioni, Jeffrey Smith, Daniele Paolo Scarpazza , "Dissecting the

NVidia Turing T4 GPU via Microbenchmarking((https://arxiv.org/pdf/1903.07486.pdf) specifics that this GPU has a maximum memory bandwidth of 320 GB/sec and an actual bandwidth of 220 GiB/sec. Using the 320 GiB/sec figure, plot a roofline model (log-log) or (lin-lin) for the GPU and plot your achieved n=2048 number on this plot.

Assume that the T4 GPU has 40 SMs and each SM has 64 SP FP cores that can do one FPMAD/cycle. Assume the GPU runs at 1.5GHz and each core can do 2 ops (1 multiply and 1 add per cycle).

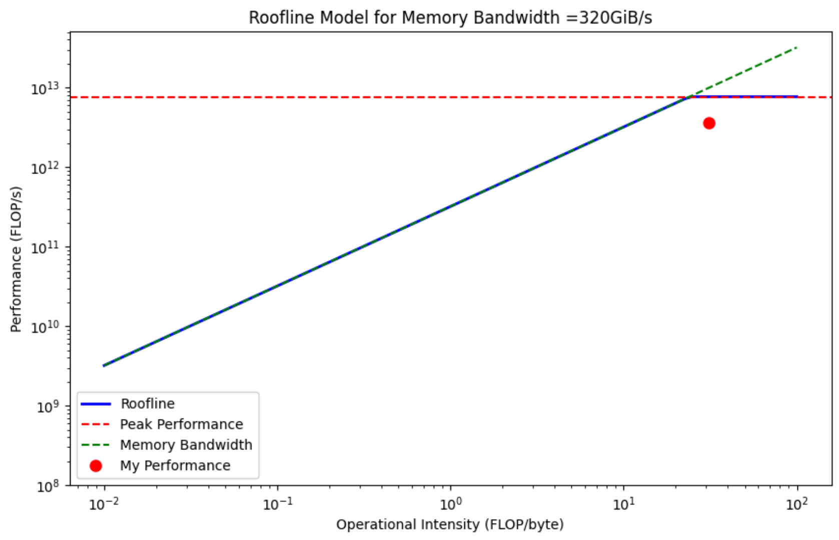
Answer - We can calculate the peak GFLOPS = 40 SMs * 64 SP cores * 2 ops/cycle * 1.5 GHz = 7680 Gflops

Calculate the peak performance of the roofline plot and explain how you arrive at the peak. Q5.b) Estimate the value of q in ops/word. Consider that the actual BW is less than 320GB/sec - Jia, etal say it is 220 GiB/sec. , Using this smaller BW, plot this roofline and calculate the new "q" value. How has the value of q been affected by the change in BW?

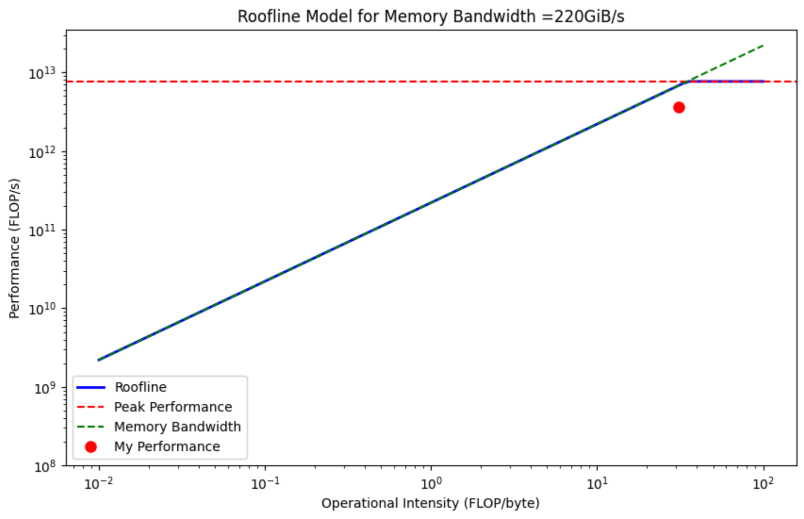
Answer - Q value is computational intensity. We assume that shared memory is fast memory and all the load/stores from global would be slow and affect load stores. So we have 32 loads each for A and B respectively, and then we do 64*2*TILEDIM_K = 8192 computations per thread. Loading is only done once at the end after amortization for n=2048 case. We are doing 2 loads per each kk block amortized. So our q value would be 8192/(66 * 4 bytes) = 31.03.

Estimating word size to be 2 bytes , q value would be 62.06

Estimating word size to be 4 bytes, q value would be 31.03 (Using this for graphs)



Q value is not being affected by change in peak bandwidth , but the change affects if the computation is memory bound or compute bound. If the BW was 320GB/sec, we would be memory bound, because the roofline would happen after q is 24. If it was 220 GB/sec, it would happen at q = 34.9 , which makes our process somewhat memory bound.



Potential Future work

- Adding another warp level layer for matrix product which may lead to better instruction level parallelism.
- Upon running Nvidia Nsight, we found that Block Limit for cuBLAS is 4 whereas for our assignment Block Limit is limited to 1. Hence, cuBLAS implementation schedules more blocks per SM, leading to higher performance. We can try to reduce the shared memory per block making it such that we can schedule more blocks per SM and test if that results in improved performance.

References

1. Jia, Maggioni, Smith, Scarpazza, "Dissecting the NVidia Turing T4 GPU via Microbenchmarking" : https://arxiv.org/abs/1903.07486
2. https://docs.nvidia.com/cuda/index.html
3. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
4. https://www.nvidia.com/content/gtc-2010/pdfs/2238_gtc2010.pdf