CSE 260

PA2 Report

Yijian Liu, Zhongrui Cao

11/10/2022

Development Flow

Q1.a) Describe how your program works (pseudo code is fine, do not include all of your code in the write-up). Small snippets of code are fine as long as they help understanding. Be sure to include a description of how your program deals with edge cases (e.g. when N does not divide evenly into a natural block size in your program).

Our program starts by having all threads load a tile of A and a tile of B to the shared memory cooperatively. We use two separate nested loops to load A and B respectively so that the memory access can coalesce. Each A tile will have size 8Nx4N and each B tile will have size 4nx8n, n being the thread block size. If an A value or B value's x coordinate or y coordinate exceeds the size of the matrix N, 0 will be loaded into the tile as padding.
Then, we do inner product of the two tiles of A and B to calculate part of the 32 C values in each thread. Repeat the loading and calculation N/(4n) times to get the final result of the 32 C values. Transfer the 32 calculated values to the result C matrix in global memory. If there are padded values, we will not assign them to the C matrix if their corresponding x or y coordinate in the C matrix exceeds the size of the matrix N.

Q1.b) What was your development process? What ideas did you try during development?

First, we tried simply using shared memory on the naive kernel, however, interestingly, we found the performance actually decreased. From 500 gFlops to around 300. Manually unrolling the loop in our shared memory implementation gave us a lot more performance, however the gFlop numbers are still comparable to the naive kernel.
From the foundation of shared memory, we implemented 1D tilling. We had a 1x4 tile and we computed 4 results per thread. This version saw some improvement from the naive kernel, at around 800 gFlops for n=1024.
To further optimize our kernel, we implemented 2D tilling. First we implemented a 2x2 simple 2D tile. Although the kernel still calculates 4 output per thread, we saw good improvement towards the performance. The kernel performs at around 1200 gFlops. Then we implemented 4x4 2D tiles. This version shows significant improvement at around 2500 gFlops, since it calculates 16 output per thread.
Since all the 2D tiling we implemented thus far are all hand unrolled, we faced a very long and unmaintainable code. We decided to rewrite everything in for loops for better readability and expandability. We also implemented bound checks for non-divisible sized matrices and smaller matrices.

Finally, we implemented non-square 2D tiling. To achieve that, our implementation utilized the TILE_SCALE variables. We saw another boost in performance at around 3000 gFlops. We further tuned our TILE_SCALE variable to avoid bank conflicts, the final parameter we used is 8x8x4.

Q1.c) What ideas worked well, what didn't work well, and why. Feel free to plot or chart results from experiments that did or did not end up in your final implementation and, as possible, provide evidence to support your theories

During development, we found that 1D tilling worked very well. The performance went from 500 gFlop to around 800 gFlops. We think this is because when we are doing 1D tilling, we are computing 4 outputs per thread, which will achieve better performance by using fewer threads, trading parallelism for locality.
We also found 2D tilling improves our performance very well. 2D tilling uses the shared memory to its full potential, and also provides more thread level parallelism (from 1x4 to 4x4), each thread can output even more. The following is our results.

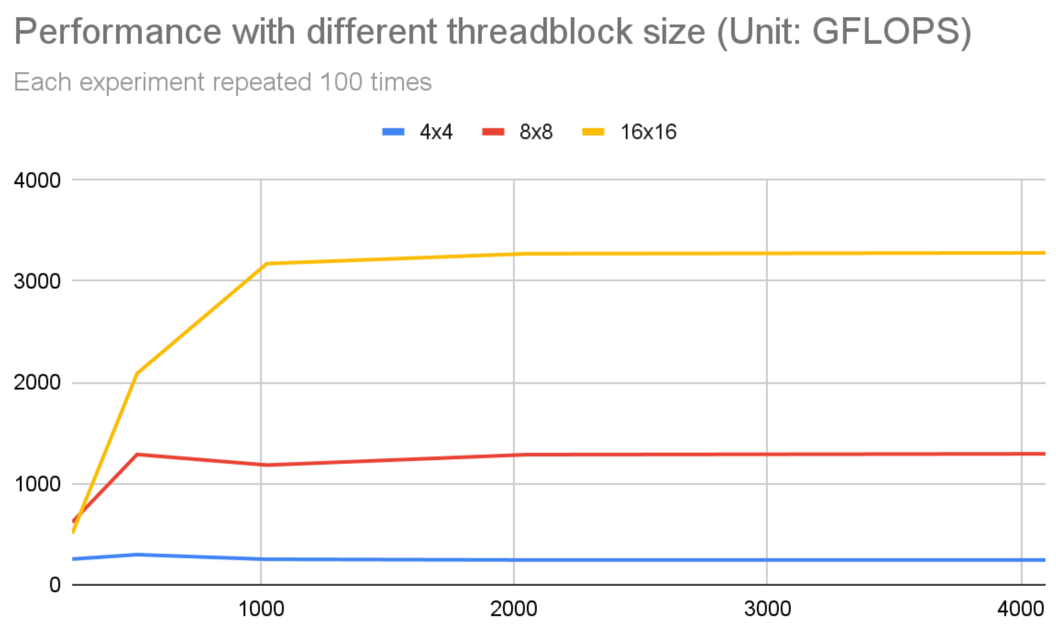
N	1D Tile 1x4	2D Tile 4x4
256	887	1379
512	1080	2371
1024	1092	2649
2048	1037	2693
4096	1021	2418

One thing that did not work is that we tried using TILE_SCALE 7x7x4 for the hope to prevent bank conflicts. We found that using the TILE_SCALE 8x8x4 is better than 7x7x4 (MxNxK) when the matrix size is big, however for smaller matrices 7x7x4 is better. For smaller matrices, we suspect this is due to the bank conflict 8x8x4 will cause. Since there are 32 threads in a warp, there are 32 independent banks of shared memory. 8 and 32 have a common divider, however 7 and 32 are relatively prime. Therefore, 7x7x4 will provide better bank performance. However, this improvement is offset by more thread level parallelism. Here are our results.

N	7x7x4	8x8x4
256(100000r)	725	514
512(43000r)	2508	2123
1024(6000r)	2990	3045
2048(830r)	3286	3339
4096(85r)	3405	3402

Result

Q2.a) For the problem sizes n=256, 512, 1024, 2048 and 4096, plot the performance of your code for a few different (at least 3) different thread block sizes. These thread block sizes may map to different tile sizes. Please mention the relationship between thread block sizes and tile size in your report. If your code has limitations on thread block size, please state the reason for that limitation.



N\Thread Block Size	4x4	8x8	16x16
256	254	618	505
512	298	1288	2088
1024	252	1183	3174
2048	245	1286	3272
4096	245	1294	3280

Given a thread block of size **nxn**, it is mapped to a **8nx4n** tile in A, a **4nx8n** tile in B, and a **4nx4n** tile in C. Because we set non-square tile scale factors, the thread block size is kept square for simplicity. We also find it difficult to move to a bigger thread block because the size of the shared memory limits the tile size and a bigger thread block leads to lower instruction-level parallelism under such limitation.
For example, since we have 64kB shared memory, that means we can load 8192 single precision numbers for both A and B. If we use 16x16 thread block size, 16x8x16x4=8192 values have already fulfilled the shared memory. If we want to use bigger thread blocks, each thread cannot calculate 4x4=16 values of C. So the instruction-level parallelism will have to decrease with a larger thread block.

Q2.b) Your report should explain the choice of optimal thread block sizes for each N(matrix size - 256, 512, 1024, 2048, 4096). Why are some sizes or geometries higher performance than others?
For matrix size 256, the optimal thread block size is 8x8. For larger matrices, the optimal thread block size is 16x16.

For large matrices, it is better to use a large thread block to fully utilize the shared memory and maintain high occupancy to hide latency. For a smaller matrix size of 256, much of the time is spent on accessing and transferring data. Using a smaller thread will separate the calculation to more SMs and parallelize the access to global memory.

Q2.c) Mention the peak GF achieved and the corresponding thread block size for each matrix size analyzed in the previous question in a table like this.

N	Peak GF	Thread Block Size
256(100000r)	624	8x8
512(43000r)	2123	16x16
1024(6000r)	3045	16x16
2048(830r)	3339	16x16
4096(85r)	3402	16x16

Section 3

Q3.a) For n=256, 512, 1024, and 2048 quantitatively compare your best result with the naive implementation

N	ours	naive
256	624	464
512	2123	578
1024	3045	534
2048	3339	418
4096	3402	409

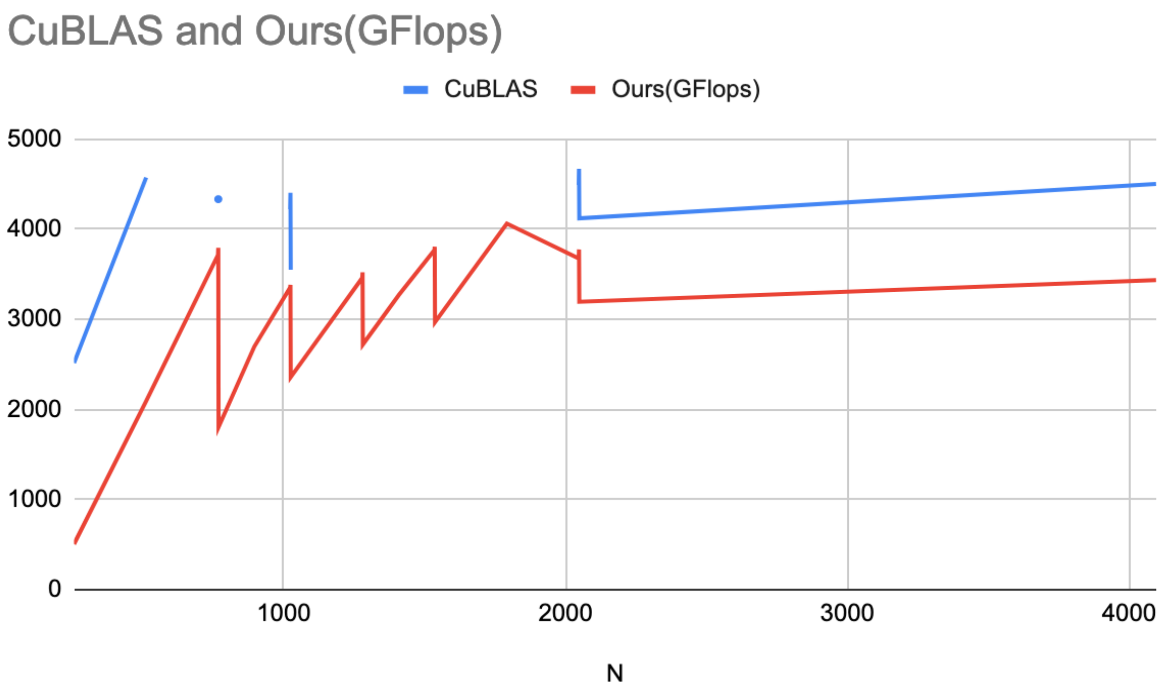
Our implementation's performance gradually increases as the matrix size goes up, however the naive implementation performance starts to decrease as matrix size gets bigger than 512.

Analysis

Q4.a) For at least twenty values of N within range (256 - 2049) inclusive, plot your performance using the best block size you determined for n=1024 in step (2). Use at least the values in the table below, but add other values too(around 20 values). Compare your results to the multi-core BLAS and cuBlas results in the table below. (for the values we gave you in the table. For other values, just report your cuda numbers).

N	BLAS (GFlops)	CuBLAS	Ours(GFlops)
256	5.84	2515.3	506
512	17.4	4573.6	2092
767			3715
768	45.3	4333.1	3793
769			1804
896			2698
1023	73.7	4222.5	3354
1024	73.6	4404.9	3381
1025	73.5	3551.0	2359

1279			3461
1280			3521
1281			2721
1408			3267
1535			3770
1536			3805
1537			2969
1792			4063
2047	171	4490.5	3675
2048	182	4669.8	3775
2049	175	4120.7	3195
4096		4501.6	3434



Q4.b) Explain how the shape of your curve is different or the same to the BLAS values and theorize as to why that might be. You may refer to the plot from Q4a.

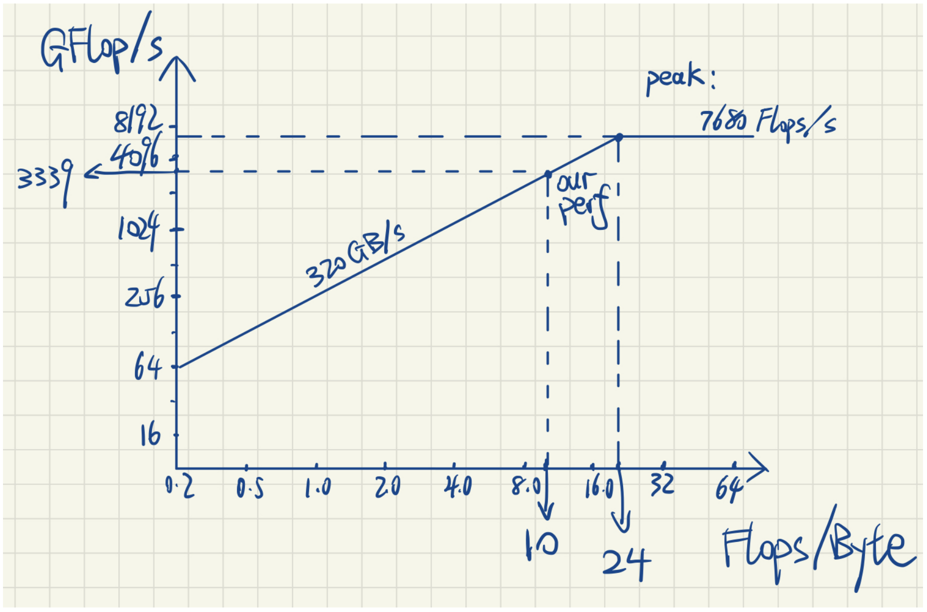
The shape of our curve is very similar to the CuBLAS curve. The two curves all start off with a low performance for very small matrix size and have a sudden increase until size ~800, then its a gradual increase from 800 to 2000. After that the performance nearly plateaued and increased very gradually until 4000. The rate of increase of CuBLAS is higher than our implementation.

Q4.c) For the twenty or so values of performance, identify and explain unusual dips, peaks or irregularities in performance with varying n.

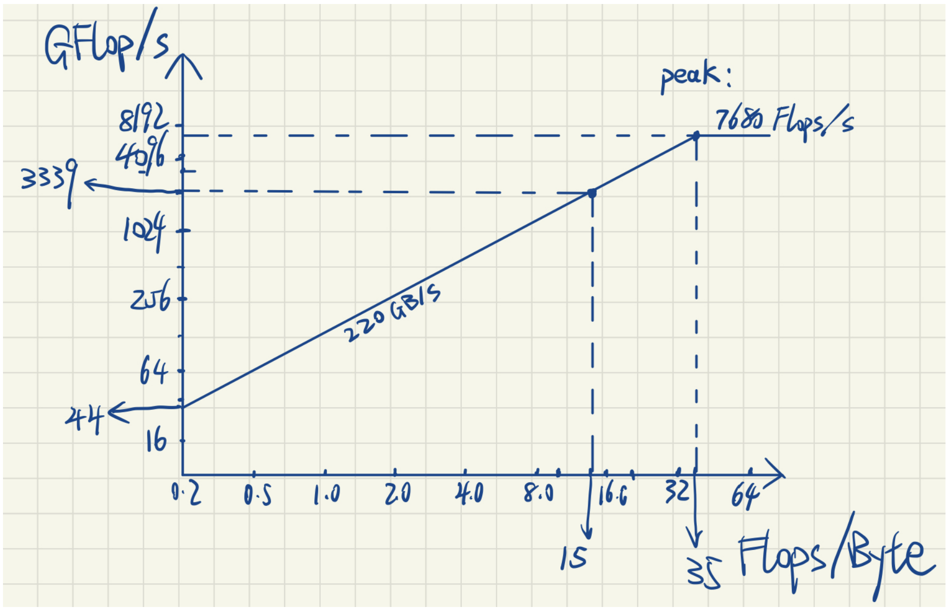
For results between 512 to 2048, there is a zig zag pattern. This is caused by the higher performance by the matrix that have sizes divisible by the tile size, the lower performing ones are the divisible sizes plus or minus one. This is caused by the fewer branches by the divisible matrix sizes, therefore getting better control divergence behaviors. This zig zag becomes less severe as the matrix size gets bigger. We think this is caused by the amount of computation offsetting the effect of control divergence. After 2048 the curve became flat, this is because of a lack of detailed data. However we do expect the curve to get more and more like a straight line after the matrix size gets bigger.

Section 5

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).
Calculate the peak performance of the roofline plot and explain how you arrive at the peak.
Peak Performance: 1.5GHz x 40 x 64 x 2 = 7680 GFLOPS
The q we calculated is at 10 Flops/Byte.



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?



Using a smaller BW, we got the new q to be 15 Flops/Byte, which is 7.5 Ops/Word. This value has increased due to the change in BW. This means that to achieve the same level of performance, more operations are needed to hide the memory access latency bounded by the bandwidth.

Potential Future work

If we had more time we could have implemented warp tiling. Also, we could write a script to do auto parameter tuning. Another thing is we could have considered more about coalescing and bank conflicts.

References

- Andrew Kerr, Duane Merrill, Julien Demouth and John Tran, CUTLASS: Fast Linear Algebra in Cuda C++, December 2017
- Cuda C++ programming guide - https://docs.nvidia.com/cuda/cuda-c-programmingguide/index.html
- Cuda documentation - https://docs.nvidia.com/cuda/index.html
- Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. Commun. ACM 52, 4 (April 2009), 65–76. https://doi.org/10.1145/1498765.1498785