Matrix Multiplication on GPU

Aayush Gupta and Sahil Bhalla

November 10, 2022

Abstract

Goal of this assignment is to optimize the task of matrix multiplication on GPU architectures. We explored the tiling algorithm, use of shared memory, thread level and instruction level parallelism to get optimal performance.

1 Development Flow (Q1)

1.1 How the program works (Q1.a)

The program uses the tiling algorithm that utilizes shared memory to improve locality in matrix multiply. Using shared memory has benefits because once a tile is loaded any thread can access values from this tile directly from shared memory instead of slower global memory. This is faster and more efficient than naive method where a row of input matrix is loaded multiple times by different threads. Moreover, tiling needs lesser memory operations than the naive method which results in higher performance.

In tiling, matrices A and B are divided into tiles, which are then loaded into shared memory collectively by multiple threads. Each thread multiplies a row in the tile of A with a column in tile of B to get a partial matrix multiplication result. Each thread repeats this for a row of tiles in A and a column of tiles in B and stores the final result in matrix C.

Using many threads in parallel for computation exhibits thread level parallelism. In addition to using multiple threads, the program computes multiple outputs per thread instead of a single output. Doing this improves resource utilization and keeps the GPU busy. Computing multiple outputs per thread exhibits instruction level parallelism and is based on the concept that instead of waiting for an instruction to finish, functional units in a GPU can work on other independent instructions and increase the device throughput.

For cases where N(matrix size) is not divisible by tile dimensions, the program has some tiles which are partially loaded with useful values but the remaining portion has values not needed for matrix multiplication. Results computed by these extra values are conditionally ignored. The program does this by checking the indices while storing elements in matrix C and ignores the values that fail the index check. This is how the program handles edge cases where N does not divide evenly into a natural block size in the program.

Apart from square tiles, rectangular tiles were also used for optimization. With rectangular tiles, the program gets another parameter to fine tune the performance. A few more optimizations were attempted to improve the performance of matrix multiplication on a GPU. They will be described in the next sub section.

1.2 Development Process (Q1.b)

1.2.1 Tiling

First step was implementing tiling. To do so, the program first initializes two pointers in shared memory - one to indicate start of tile of A (referred to as As) and another to indicate start of tile of B (referred to as Bs). Then x and y values of block index and thread index are determined for the current thread. Block indices specify the tile on which the current thread has to work. Thread indices specify the location of the element in tile of C that the thread has to compute. For every tile of C, there is a row of tiles in A and a column of tiles in B that take part in computing the tile of C. The program iterates over these tiles and in every iteration a tile of A and B is used to compute a partial result.

In every iteration, the initial step is to load a tile of matrix A (As) and B (Bs) into shared memory which is done collectively by all the threads. The current thread uses its thread indices to decide which element to load, loads it into shared memory and waits for other thread to finish loading. Once the tiles have been loaded, the current thread multiplies a row in tile As (given by the y dimension of thread index) with a column in tile Bs (given by the x dimension of thread index), stores the result in a float variable and waits for other threads to do the same.

After all iterations, all the threads store their final results in C at the location determined by a combination of block index and thread index. This gives us a tile of values in the output matrix C.

1.2.2 Instruction level parallelism

Tiling performs better than the naive implementation, but even in tiling, one thread computes only one output. So, the next step was to attempt instruction level parallelism by making every thread compute multiple outputs.

This was done by making every thread compute 4 outputs in both dimensions (Value '4' is used here for illustration. The program uses a parameter TILESCALE instead of '4'). Since every thread computes more than 1 output now, lesser threads are required to achieve same performance as before.

Moreover, the values that each thread computes are not adjacent to each other and are instead spread uniformly over the tile. This way, when memory requests from multiple threads fall into the same memory segment, they will be combined into a single burst and number of bank accesses will be reduced. This is known as memory coalescing and it improves performance by satisfying multiple memory requests in one attempt.

1.2.3 Edge cases, Rectangular tiling and other optimizations

If N is divisible by tile dimensions, there is a whole number of tiles fully filled with useful values and everything works normally. But when N is not divisible by tile dimensions, some extra tiles are required to consume remaining values in the input matrices. But such tiles are only partially filled with useful values. To handle this case, the program stores only those results that lie inside the index bounds of matrix C and all other values are discarded.

Once the program was working for edge cases, we experimented with rectangular tiles. Using square tiles, only half of shared memory (32KB) could be utilized. By switching to rectangular tiles, the program was able to use all the shared memory (64KB) and trying different tile sizes led to optimal dimensions.

In addition, the program uses register keyword for storing partial results of C into registers. Loop unrolling and restrict keyword are also used inside the kernel to get some performance improvements.

1.3 What worked well, what didn't (Q1.c)

Tiling improved performance significantly over naive implementation. One reason behind this is the way input matrices are loaded from memory. In naive implementation, two threads that compute adjacent elements in matrix C will load the same row of matrix A twice into memory. Whereas in tiling, a tile of A is loaded into faster shared memory once and different threads can access required values directly from shared memory thereby reducing the number of memory operations and memory latency.

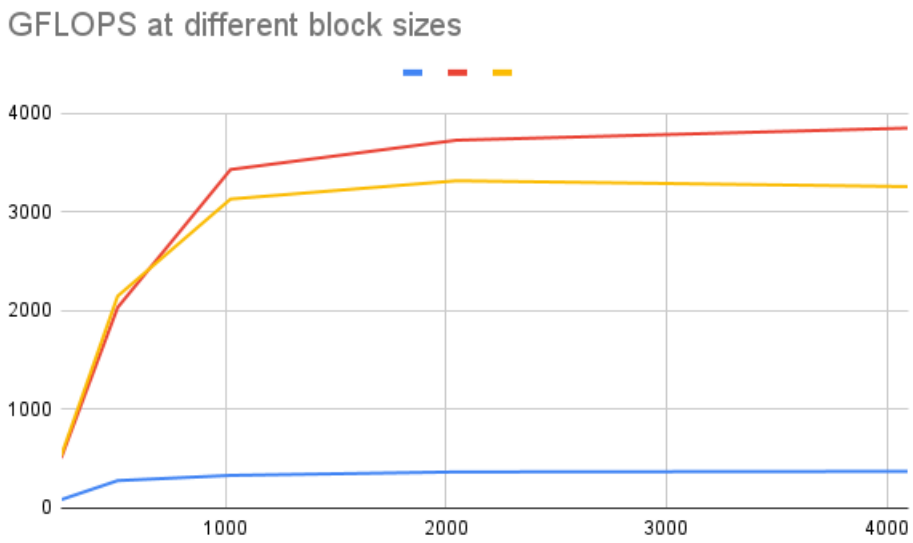
Making threads output multiple values instead of one also improved performance. This is because by computing multiple outputs per thread, cores can hide latency and increase throughput by executing other independent instructions while waiting for a particular instruction to finish.

For handling edge cases, one way was to add conditional if statements while loading tiles of matrices A and B into memory (this did **not** end up in final implementation) to prevent using extra values in tiles that do not correspond to any value in the input matrix. But this idea lowered the performance significantly, probably because introducing conditionals interferes with parallelism. A conditional might cause two threads to take different branches and what otherwise could have been executed in one cycle gets executed in multiple cycles. This causes a dip in performance.

Introducing rectangular tiles also gave a boost to performance. With square tiles, the program could use only half of shared memory (with 64 X 64 tile size each for A and B). But with rectangular tiles, it was able to fully utilize the shared memory (with 128 X 64 tile size each for A and B). With more values in shared memory as compared to previous cases, load instructions were executed faster and this explains the performance improvement.

2 Results (Q2)

2.1 Performance Plot (Q2.a)



Where blue is 8x8, red is 16x16, and orange is 32x32.

In each of these block sizes the tile dimensions are exactly the same (128 x 64), in order to use all of the shared memory. This means in the smaller block sizes each thread is doing comparatively more work, so TILESCALE increases as the block size decreases.

There are limitations on thread block sizes higher than 32 x 32, as the maximum warps in an SM for the T4 gpu is 32, and a 32 x 32 block perfectly uses all of them. Any larger block size would not be able to fit in an SM, so would not be possible.

2.2 Optimal Thread Block Geometry (Q2.b)

For N = 256 and 512, the 32x32 block size performs best. For $N \geq 1024$, 16x16 works best. At the lower values of N 32x32 works best because it results in a higher occupancy (100% as measured by NInsight), and still includes enough ILP (Instruction Level Parallelism) to hide latency and fill the pipeline. At these Ns, it works better than the smaller block sizes because they have a lower occupancy (according to NInsight), and therefore less threads to throw at the problem.

At higher Ns, where $N \geq 1024$, 16x16 works best because the memory latency increases. I believe that this is because the L2 cache for the GPU is size 4096KiB, which can at maximum fit a 1024x1024 matrix. This means that at values of $N \geq 1024$, the entire matrix is no longer within the L2 cache, so there are more cache misses which causes increased memory latency. This means more ILP is needed to hide the latency and since increased ILP occurs at smaller block sizes, 16x16 is faster than 32x32. 8x8 is still slower since the

occupancy is extremely low for that one, and there is not enough parallelism to have fast performance.

2.3 Peak GF (Q2.c)

N	Peak GF	Thread Block Size
256	531.1	32x32
512	2148.5	32x32
1024	3431.1	16x16
2048	3728.8	16x16
4096	3851.3	16x16

Table 1: Peak GF and thread block size for different matrix sizes

3 Comparison with Naive Implementation (Q3)

N	Our implementation (gflops)	Naive Implementation (gflops)
256	531.1	72.545636
512	2148.5	108.841584
1024	3431.1	132.654689
2048	3728.8	125.668684

Table 2: Quantitative comparison with naive implementation

One can see that at all listed values of N, our implementation has at least a 20x times speedup over the naive implementation, in terms of gflops.

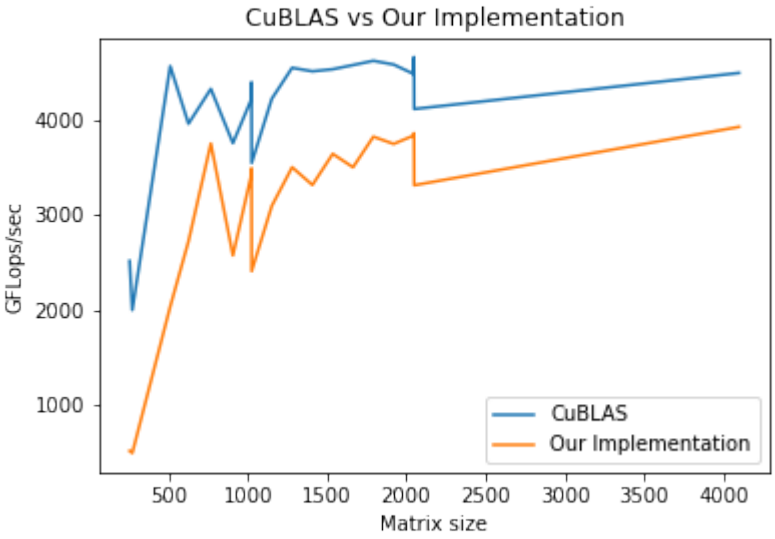
We can see that at lower values of N, this speed up is lower than at higher values, and this is because the tiled approach introduces some minimal overhead that becomes less significant at higher values of N.

4 Analysis (Q4)

4.1 Comparison with BLAS and CuBLAS (Q4.a)

N	BLAS	CuBLAS	Our Result
256	5.84	2515.3	508.4
273		1996.1	484.7
512	17.4	4573.6	2032.9
627		3965.4	2722.3
768	45.3	4333.1	3755.7
907		3760.4	2573.1
1023	73.7	4222.5	3425.6
1024	73.6	4404.9	3495.2
1025	73.5	3551.0	2407.2
1152		4229.0	3096.2
1280		4556.0	3505.1
1408		4519.3	3317.9
1536		4540.9	3646.7
1664		4585.2	3505.1
1792		4630.2	3828.1
1920		4590.5	3751.8
2047	171	4490.5	3840.9
2048	182	4669.8	3864.6
2049	175	4120.7	3314.9
4096	-	4501.6	3931.7

Table 3: BLAS vs CuBLAS vs Our Result



4.2 Comparison of curve shape with CuBLAS (Q4.b)

Shape of curve of the current implementation is similar to shape of CuBLAS values. Dips and peaks in both the curves occur at similar matrix sizes. Since CuBLAS also uses thread level and instruction level parallelism, the matrix sizes for which both kinds of parallelism peak or go down will be same for both the implementations. That might explain the similarity between shapes of the two curves in the plot.

One difference between CuBLAS and our implementation is that we have a sharp constant decline in performance starting around N=800 to N = 0, while CuBLAS peaks a few more times in the range 0-800. This is because our grid size is based only on the ratio of N to our tile dimensions. Since our tile dimensions are constant, at around N=800 and below, the area of our grid drops below 40, so there are less than 40 total blocks. Since there are 40 SMs in the Turing GPU, this means we are underutilized the GPU, resulting in this drop in performance. CuBLAS on the other hand might be reducing the tile dimensions at this point, so that the grid area remains at or above 40.

4.3 Explain irregularities in performance (Q4.c)

In general, performance increases with matrix size with minor ups and downs but there are sudden significant dips at certain sizes like at N=907, 1025 and 2049. One reason behind this might be the fact that for these matrix sizes, tile dimension does not divide N and in addition, a large portion of the extra tiles (required for accommodating leftover values in input matrices) remains unused.

For instance, when N=1025 and tile size for matrix A is 128 X 64, there are 17 tiles in a row. The first 16 tiles are fully filled with useful values from matrix A but the 17^{th} tile has only $1st$ column filled with useful values. Remaining 63 columns of the tile remain unused. This might explain the sudden dips in performance at certain matrix sizes.

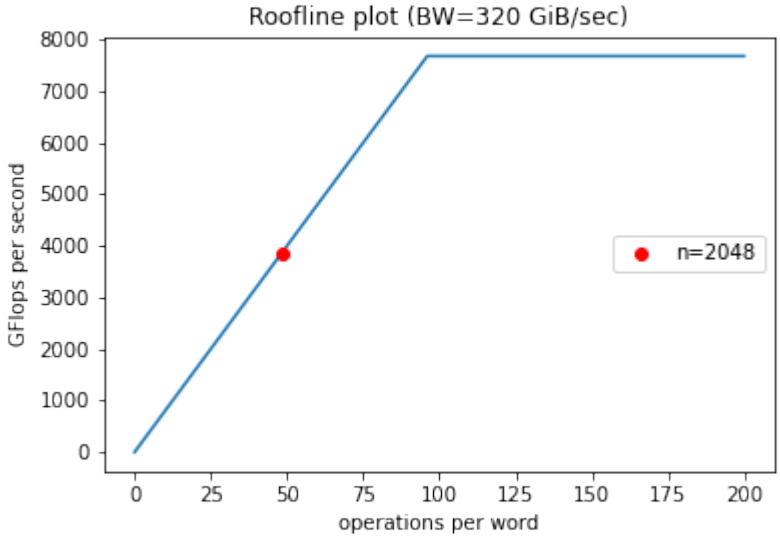
5 Roofline Plot (Q5)

At peak performance, every core is active and executes instructions in every cycle.

=> Peak performance = number of SMs X number of SP FP cores in an SM X operations per core X core frequency

=> Peak performance = 40 X 64 X 2 X 1.5 X 10^9 flops/sec = 7680 GFlops/sec

To get peak performance, q has to be increased. This can be done by increasing the number of arithmetic operations for every memory operation or by decreasing the number of memory operations for every arithmetic operation.

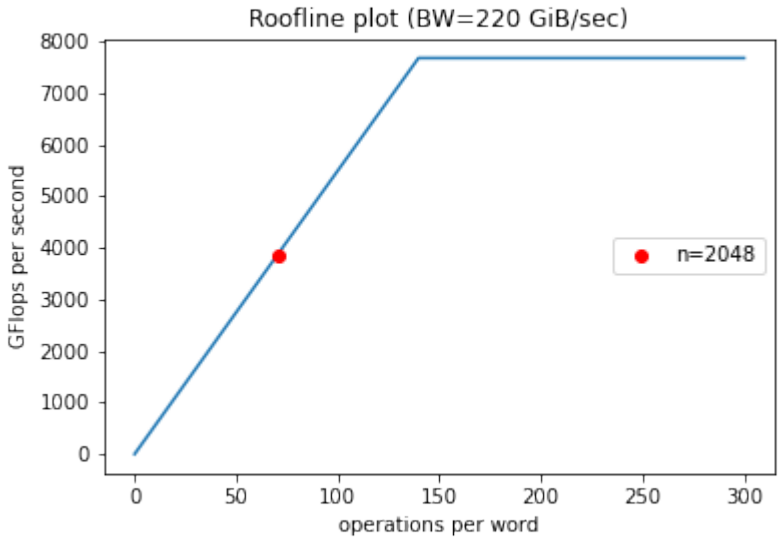


For n=2048 and BW = 320 GiB/sec :-

$$\text{q} = \frac{3864.6 GFlops/sec}{320 GiB/sec} 4B/word = 48.3\ ops/word$$

For peak performance and BW = 320 GiB/sec :-

$$\text{q} = \frac{7680 GFlops/sec}{320 GiB/sec} 4B/word = 96\ ops/word$$



For n=2048 and BW = 220 GiB/sec :-

$$\text{q} = \frac{3864.6 GFlops/sec}{220 GiB/sec} 4B/word = 70.3\ ops/word$$

For peak performance and BW = 220 GiB/sec :-

$$q = \frac{7680 GFlops/sec}{220 GiB/sec} 4B/word = 140\ ops/word$$

From the values computed above, it can be concluded that q increases on decreasing the bandwidth.

6 Future Work (Q6)

1. In the current implementation tiles in matrices A and B are of same size. One idea is to use totally different tile sizes for matrices A and B.
2. Reduce the number of calls to syncthreads() by creating a new buffer in shared memory.
3. Use of shuffle instruction to share data between threads.
4. Implementing a matrix multiplication kernel that can multiply two non-square matrices.

7 References (Q7)

1. Andrew Kerr, Duane Merrill, Julien Demouth and John Tran , CUTLASS: Fast Linear Algebra in Cuda C++, December 2017
2. Nvidia Cutlass github - `https://github.com/NVIDIA/cutlass/blob/master/media/docs/efficient_gemm.md`
3. Volkov, Demmel, Benchmarking GPUs to Tune Dense Linear Algebra, SC2008
4. Volkov, Better Performance at lower Occupancy, GTC2010
5. Cuda C++ programming guide - `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`
6. Cuda documentation - `https://docs.nvidia.com/cuda/index.html`
7. Jia, Maggioni, Smith, Scarpazza, "Dissecting the NVidia Turing T4 GPU via Microbenchmarking" : `https://arxiv.org/abs/1903.07486`