CSE260: HW2 Report

Team Members: Abhinandan Sharma, Vaibhav Jain

Section (1) - Development Flow

Q1.a) Describe how your program works. Do not simply paste your code, small snippets of code or pseudo 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 performs matrix multiplication using a tiled, shared-memory CUDA kernel. Instead of computing each output entry directly from global memory, we divide the matrices into tiles and load those tiles into shared memory so threads in a block can reuse them. Each block is responsible for producing a large tile of the output matrix C, and each thread inside the block computes a 4×4 sub-block. This increases arithmetic intensity and significantly reduces global memory traffic compared to the naïve version.

Shared memory and tiling: At the start of the kernel, we allocate shared memory and split it into two regions, tileA and tileB. These store one tile from A and one tile from B that the entire block will use. The tile sizes are defined by TILEDIM_M, TILEDIM_N, and TILEDIM_K. Using shared memory allows each loaded value from A or B to be reused many times by the threads in the block.

```
extern __shared__ _FTYPE_ sharedMem[];
_FTYPE_ *tileA = sharedMem;
_FTYPE_ *tileB = sharedMem + TILEDIM_M * TILEDIM_K;
```

```
// Each thread accumulates a 4x4 sub-block of C
_FTYPE_ C_0_0 = 0, C_0_1 = 0, C_0_2 = 0, C_0_3 = 0;
_FTYPE_ C_1_0 = 0, C_1_1 = 0, C_1_2 = 0, C_1_3 = 0;
_FTYPE_ C_2_0 = 0, C_2_1 = 0, C_2_2 = 0, C_2_3 = 0;
_FTYPE_ C_3_0 = 0, C_3_1 = 0, C_3_2 = 0, C_3_3 = 0;
```

Work per thread: Each thread accumulates results for a 4×4 output block, which is why the code declares 16 accumulators (C_0_0 through C_3_3). This design lets each thread make good use of the data in shared memory: after a tile is loaded, one thread multiplies four rows of A's tile with four columns of B's tile inside the inner loop.

Iterating across the K dimension: Matrix multiplication requires summing over the entire K dimension, so we break that dimension into tiles as well. The kernel computes numTilesK = ceil(N / TILEDIM_K), and for each of these tiles, threads cooperatively load the required chunk of A and B into shared memory, synchronize, perform the partial multiplication for that tile, and then synchronize again before loading the next tile. This loop begins at the line where tileIdx is introduced.

Cooperative loading and coalescing: During tile loading, each thread loads several elements of A and several elements of B into shared memory. The index calculations are structured so that consecutive threads read consecutive memory locations, achieving coalesced global memory access. The long sets of if-conditions in the tile-loading region ensure that whenever a computed global index falls outside the matrix dimensions, the thread writes 0.0 into shared memory instead of reading out of bounds. This effectively zero-pads the matrix tiles and makes the multiplication correct even for edge tiles.

```
if (globalRowABase < N && globalColA < N) {
    tileA[sharedRowABase * TILEDIM_K + sharedColA] =
        A[globalRowABase * N + globalColA];
} else {
    tileA[sharedRowABase * TILEDIM_K + sharedColA] = 0.0;
}
```

Handling edge cases: When N is not a multiple of the tile dimensions, the last tiles in each direction are only partially valid. Our program handles this safely with bounds checks on every global load and global store. For loads, if (row < N && col < N) we read the appropriate element; otherwise we assign 0.0 in shared memory. For stores at the end of the kernel, we check the bounds again before writing each of the 16 results. These two mechanisms ensure that all boundary tiles behave as if the matrix had been padded with zeros, without performing any invalid memory access.

Computing the partial results: After loading a tile of A and B and synchronizing, each thread runs the inner loop over k from 0 to TILEDIM_K−1. In this loop, it reads four values from four rows of tileA and four values from four columns of tileB and updates the 4×4 accumulators. This is where most of the arithmetic happens, and because everything is in shared memory, the accesses are fast and reused many times.

Writing the final results: After the program finishes iterating through all K tiles, each thread writes the final 4×4 block to global memory. As with loads, these writes use boundary checks so that only valid locations inside the N×N output matrix are modified. This completes the computation for the block's region of C.

Q1.b) What was your development process? What ideas did you try during development? Try to reference commits wherever possible, you can push the code for ideas that didn't work to separate branches if required.

Our starting point was a naïve matrix-multiplication kernel where each thread computed one output element directly from global memory. This version ensured our indexing logic, memory transfers, and overall structure were correct before optimizing. (Commit: "Initial commit")

After confirming correctness, we experimented with shared-memory tiling using a simple 16×16 tile as taught in the examples in the slides. Here the block size was equal to the tile size. Threads cooperatively loaded one element each into shared memory. While this improved performance, the compute-to-memory ratio was still low. (Commit: "Added tiling and shared memory")

Next, we tested increasing the amount of work per thread. We tried having each thread compute 2×2 output blocks and the performance increased significantly. We adjusted indexing and shared-memory layout accordingly but unfortunately our code was failing for the cases where the matrix size was not a power of two. (Commit: "Increasing the tile size - Now Tile_Dim is double the blk_Dim")

Seeing the trend, it was clear that increasing the work done by each thread can further boost our performance. So, further increased the Tile size from 32 to 64, keeping the same block size of 16. Here each thread was computing 16 values of C. This highly tiled version gave the best balance between register usage and throughput, so we adopted it. During performance testing, we discovered that some global loads were not coalesced. We redesigned the loading code to ensure that threads in a warp read contiguous addresses from A and B. This resulted in the structured sequence of boundary-checked loads seen in the final kernel. We also improved boundary handling. Since N may not be divisible by tile dimensions, early versions produced incorrect results for certain sizes. We integrated explicit bounds checks into every load and store, treating out-of-range accesses as zeros for tiles at the edges. (Commit: "Increasing the tile size - Increasing the tile size from 32 to 64 so that each thread does even more work.")

When we introduced conditional code blocks to handle edge cases, we initially used the C++ ternary operator. This fixed the issue for blocks whose dimensions were not powers of two, which were previously failing. However, it unexpectedly reduced the performance of blocks with power-of-two sizes. After spending considerable time debugging, we found discussions online suggesting that ternary operators can sometimes perform worse than if-else statements. Replacing the ternary operators with if-else conditions resolved the issue and restored optimal performance.

We also tested with different block dimensions before finalizing the current configuration. We tried to go slightly above and slightly below the current block dimension of 16. We made two separate branches and updated the code indices to support block sizes of 32 and 8 but both of them resulted in poorer performance. (Commits: "Changing the block size to 32", "Changing the block size to 8")

Q1.c) What ideas worked well, what didn't work well, and why. Feel free to include things that did or didn't end up in your final presentation. Provide evidence to support your theories. Make sure that your analysis is clear and concise. The use of subsections (e.g., "Successful: Idea A," "Unsuccessful: Experiment B"), charts, and tables can help with this. Excessively verbose or poorly structured answers will be penalized.

Successful: Using the naive kernel as a performance baseline

The naive kernel was slow, but it was essential during development. It helped verify indexing logic, memory behavior, and correctness every time an optimization was added. It also served as the ground truth for measuring speedups.

Evidence: The naive results in the screenshots (for example, N = 256 giving about 56 GFLOPS) were consistent and correct across early tests, which made it reliable for verifying later versions.

Successful: Shared memory tiling

Introducing shared memory caused an immediate performance improvement. By loading each tile of A and B once and reusing it within the block, the kernel greatly reduced global memory traffic and increased arithmetic intensity. This was one of the most important optimizations in the project.

Evidence: The first tiled version (N = 32 first commit) already jumped from tens of GFLOPS to around 800–1000 GFLOPS.

Successful: Increasing work per thread (4 by 4 output block)

Allowing each thread to compute a 4 by 4 block of C increased reuse inside registers and enabled more computation per tile load. This reduced shared memory pressure and increased arithmetic intensity. This design choice produced the high GFLOPS results seen in the final kernel.

Evidence: With block size 16 (which implemented the 4 by 4 structure), performance reached more than 3600 GFLOPS for N = 1024 and N = 2048, shown in the final optimized version (block size 16, tile size 64) screenshot.

Successful: Coalesced and boundary-checked global loads

Restructuring indexing so that each warp accessed contiguous elements stabilized and improved performance. Earlier versions suffered from scattered loads that lowered effective bandwidth. The final boundary-checked loads made memory access fully coalesced and safe.

Evidence: After increasing the tile size to 64 and fixing the loads, GFLOPS rose significantly (for example, from about 1000 GFLOPS in the N = 32 tiled version to more than 3000 GFLOPS in the final version).

Successful: Careful handling of edge cases

Explicit bounds checks prevented invalid reads and writes for sizes not divisible by the tile size. This ensured correct outputs for challenging values such as 1025 and 2047.

Evidence: Earlier versions failed tests for these sizes (as shown in the “power of 2 only” screenshot), but the final version passed all of them.

Unsuccessful: Overly complex shared memory load patterns

Early attempts used long, repetitive loading sequences for A and B tiles. Although correct, this approach increased register pressure and instruction count and did not provide a proportional performance improvement. Simplified loading patterns worked just as well and were easier to maintain.

Evidence: More complex variants, such as attempts with block size 32, performed worse than expected because of higher per-thread overhead.

Unsuccessful: Ternary operator based bounds checks

Using ternary expressions initially seemed compact, but it caused noticeable performance drops for power-of-two sizes like 1024. Replacing these expressions with standard if-else blocks restored expected performance.

Evidence: After removing ternary operators, the kernel regained the high GFLOPS values that matched the theoretical behavior for N = 1024 and N = 2048.

Unsuccessful: Block sizes 8 and 32

Testing alternate block sizes provided insight into geometric effects. Block size 8 produced low performance due to tiny tiles that had little data reuse. Block size 32 increased register usage and reduced occupancy. Both underperformed compared to block size 16.

Evidence:

- Block size 8 achieved only about 340–680 GFLOPS.
- Block size 32 achieved about 1100–1700 GFLOPS.
- Block size 16 achieved about 1600–3700 GFLOPS.

These results clearly show that block size 16 was the best choice.

Unsuccessful: Fixed tile size constraints

Hard-coding the 64 by 64 tile shape made exploring alternate tiling patterns more difficult. Although this tile size performed best, the rigid structure limited experimentation with deeper K-tiling or asymmetric tiles.

Evidence: Attempts with different block sizes did not achieve better results, likely because the surrounding code and tile layout were tightly coupled to the chosen dimensions.

Unsuccessful: Early non-coalesced indexing

Initial indexing patterns in early tiled implementations led to scattered global memory accesses. This reduced bandwidth utilization and caused slower performance. Only after reorganizing the loads to ensure full coalescing did performance improve.

Evidence: Early tiled kernel runs at N = 32 and N = 64 showed much lower GFLOPS than later optimized versions, which confirmed the indexing issues.

Naive kernel results

```
ubuntu@ip-172-31-8-3:~/cse260-fa25-hw2-hw2-abs021-v8jain$ ./grade.sh
N: 256, GFLOPS: 56.146768
N: 512, GFLOPS: 75.333706
N: 1024, GFLOPS: 79.638194
N: 1025, GFLOPS: 69.538973
N: 2047, GFLOPS: 67.467096
N: 2048, GFLOPS: 52.716886
```

First tiled implementation (N=32 first commit)

```
ubuntu@ip-172-31-8-3:~/cse260-fa25-hw2-hw2-abs021-v8jain$ ./grade.sh
N: 256, GFLOPS: 812.103222
N: 512, GFLOPS: 1014.141512
N: 1024, GFLOPS: 1046.071570
N: 1025, GFLOPS: 947.933148
N: 2047, GFLOPS: 954.944297
N: 2048, GFLOPS: 962.009554
```

Power-of-two restricted kernel showing failures for N = 1025 and N = 2047

```
ubuntu@ip-172-31-8-3:~/cse260-fa25-hw2-hw2-abs021-v8jain$ ./grade.sh
N: 256, GFLOPS: 1434.632909
N: 512, GFLOPS: 1976.650117
N: 1024, GFLOPS: 1997.117415
Test failed for N=1025
Test failed for N=2047
N: 2048, GFLOPS: 1786.464810
```

Final optimized version (block size 16, tile size 64)

```
ubuntu@ip-172-31-8-3:~/cse260-fa25-hw2-hw2-abs021-v8jain$ ./grade.sh
N: 256, GFLOPS: 1619.533813
N: 512, GFLOPS: 3040.507445
N: 1024, GFLOPS: 3640.449137
N: 1025, GFLOPS: 3080.305968
N: 2047, GFLOPS: 3700.344693
N: 2048, GFLOPS: 3671.855508
```

Block size 32 variant

```
ubuntu@ip-172-31-8-3:~/cse260-fa25-hw2-hw2-abs021-v8jain$ ./grade.sh
N: 256, GFLOPS: 1172.812403
N: 512, GFLOPS: 1523.545205
N: 1024, GFLOPS: 1734.789248
N: 1025, GFLOPS: 1513.649790
N: 2047, GFLOPS: 1728.084763
N: 2048, GFLOPS: 1792.927926
```

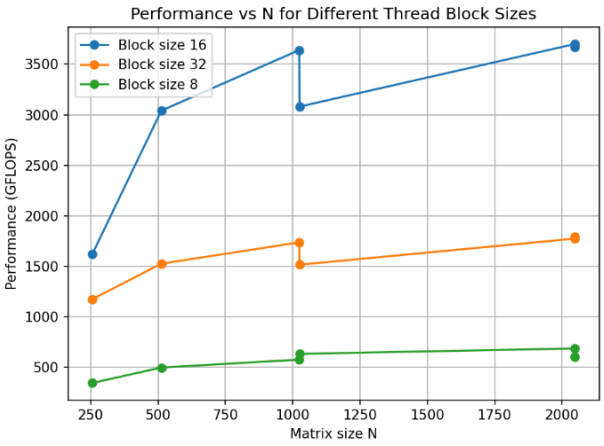
Block size 8 variant

```
ubuntu@ip-172-31-8-3:~/cse260-fa25-hw2-hw2-abs021-v8jain$ ./grade.sh
N: 256, GFLOPS: 341.678777
N: 512, GFLOPS: 494.531518
N: 1024, GFLOPS: 572.951539
N: 1025, GFLOPS: 630.257954
N: 2047, GFLOPS: 683.449810
N: 2048, GFLOPS: 602.847130
```

Section (2) - Result
Your implementation will be graded on its performance at the following matrix sizes: n=256, 512, 1024, 1025, 2047, and 2048.

Q2.a) For the given sizes n, 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.



The Tile size is kept as 64 for all these block sizes.

Q2.b) Explain the choice of optimal thread block sizes for each N. Why do some sizes or geometries have higher performance than others?

The best-performing block size in our implementation is 16, because it corresponds to a larger effective tile in shared memory, which allows much more reuse of loaded matrix elements. With a bigger tile, each value of A and B is reused many times before being evicted, which increases arithmetic intensity and reduces global memory traffic. As a result, block size 16 consistently delivers the highest GFLOPS for every tested N, and its advantage becomes more pronounced for larger matrices where the GPU can fully utilize the larger tile.

Block size 32 produces moderate performance. Although it still benefits from shared-memory tiling, the tile is smaller than in the block size 16 configuration, so each block performs less work per global load. This limits data reuse, reduces arithmetic intensity, and lowers overall throughput. As N increases, performance improves, but it never matches the block size 16 results.

Block size 8 is consistently the slowest. Very small tiles drastically limit shared-memory reuse, forcing many more loads from global memory. This leads to lower occupancy, insufficient latency hiding, and performance that begins to resemble a naive kernel. As a result, block size 8 stays well below the other configurations across all matrix sizes.

Basically, the block size governs the total number of threads active on an SM and decreasing them below a point would lead to lower SMs being used and increasing it beyond a point would lead to lesser occupancy on all SMs. Block size of 16 seems to be the sweet spot where we can activate most of the SMs with a considerable amount of occupancy.

Overall, different thread block sizes lead to different levels of tiling efficiency and occupancy. Larger tiles (such as those from block size 16) maximize shared-memory reuse and provide enough work per thread block to keep the SMs busy, while smaller blocks reduce reuse and increase memory traffic. This is why block size 16 achieves the highest performance for all tested N.

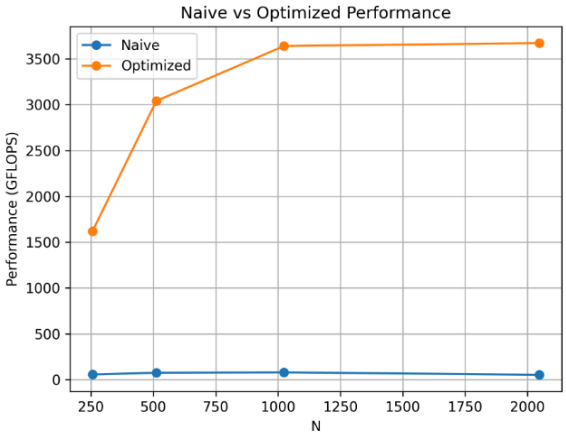
Q2.c) Document the peak GF achieved and the corresponding thread block size for each matrix size in the table.

N	Peak GF	Thread Block Size
256	1619.533813	16

512	3040.507445	16
1024	3640.449137	16
1025	3080.305968	16
2047	3700.344693	16
2048	3671.855508	16

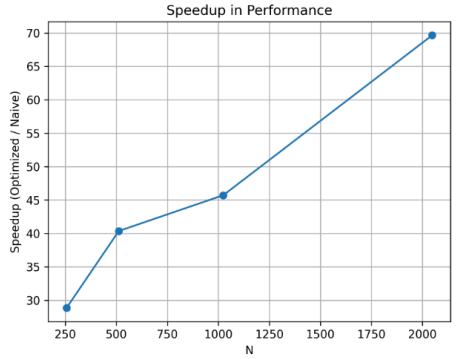
Section (3) - Comparison to Naive

Q3.a) For n=256, 512, 1024, and 2048 graph your best result on each size with the naive implementation.



The optimized version achieves dramatically higher GFLOPS than the naive implementation across all matrix sizes. This is because tiling, shared memory reuse, and better global-memory coalescing significantly reduce memory traffic and allow the GPU to reach much higher compute utilization. The naive kernel remains bottlenecked by repeated global memory loads, causing its performance to stay near zero compared to the optimized version.

Q3.b) Graph the speedup for n=256, 512, 1024 and 2048 for your best result on each size versus naive. Optimized cuda version with til_dim =64, block_dim =16



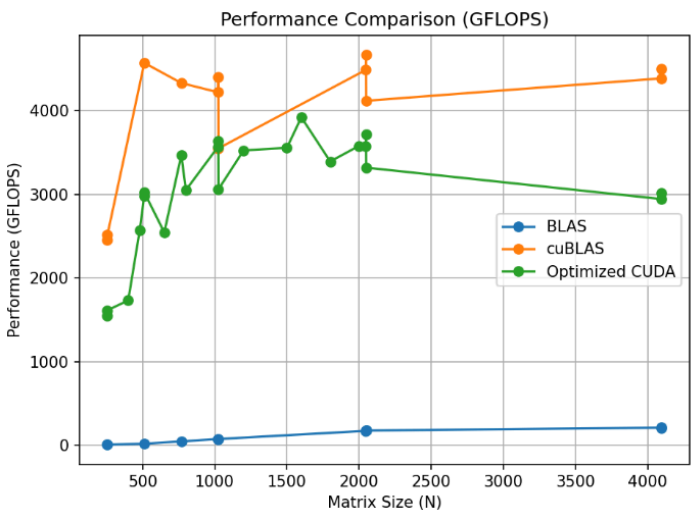
The speedup increases with problem size, growing from about 30× at N=256 to nearly 70× at N=2048. Larger matrices provide more computation per memory access and allow the optimized kernel to fully benefit from tiling and high occupancy. As N grows, the naive kernel becomes increasingly memory-bound, while the optimized kernel scales much better, leading to higher speedups.

Section (4) - Analysis

Q4.a) For at least twenty values of N within range (255 - 2049) inclusive, graph your performance using the best block size you determined for n=1024 in section (2). Use at least the values in the table below, but add other values too (around 20 values total). 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	Optimized Result with block size=16 (GFlops)

255	5.93	2456.2	1548.938251
256	5.84	2515.3	1613.962022
400	-	-	1732.400490
480	-	-	2571.979307
511	-	-	2978.487796
512	17.4	4573.6	3023.361726
650	-	-	2546.392696
768	45.3	4333.1	3469.922551
800	-	-	3052.137078
1023	73.7	4222.5	3570.483337
1024	73.6	4404.9	3636.627606
1025	73.5	3551.0	3061.100373
1200	-	-	3527.232486
1500	-	-	3558.023903
1600	-	-	3923.016312
1800	-	-	3394.533889
2000	-	-	3581.660903
2047	171	4490.5	3574.264674
2048	182	4669.8	3719.639176
2049	175	4120.7	3320.856072
4095	209.2	4389.0	2944.886481
4096	213.4	4501.6	3015.170254




```
ubuntu@ip-172-31-8-3:~/cse260-fa25-hw2-hw2-abs021-v8jain$ ./grade.sh
N: 255, GFLOPS: 1548.938251
N: 256, GFLOPS: 1613.962022
N: 400, GFLOPS: 1732.400490
N: 480, GFLOPS: 2571.979307
N: 511, GFLOPS: 2978.487796
N: 512, GFLOPS: 3023.361726
N: 650, GFLOPS: 2546.392696
N: 768, GFLOPS: 3469.922551
N: 800, GFLOPS: 3052.137078
N: 1023, GFLOPS: 3570.483337
N: 1024, GFLOPS: 3636.627606
N: 1025, GFLOPS: 3061.100373
N: 1200, GFLOPS: 3527.232486
N: 1500, GFLOPS: 3558.023903
N: 1600, GFLOPS: 3923.016312
N: 1800, GFLOPS: 3394.533889
N: 2000, GFLOPS: 3581.660903
N: 2047, GFLOPS: 3574.264674
N: 2048, GFLOPS: 3719.639176
N: 2049, GFLOPS: 3320.856072
N: 4095, GFLOPS: 2944.886481
N: 4096, GFLOPS: 3015.170254
```

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 Q4.a.

The shape of the optimized CUDA curve is very different from the BLAS curve. The BLAS line stays extremely low and increases only gradually as N grows, while the optimized CUDA curve rises quickly into the high GFLOPS range (around 3000 to 3900 GFLOPS) and then varies based on how well each matrix size matches the GPU architecture. BLAS performance is limited by the CPU's small number of cores and its memory bandwidth, so even for the largest matrix sizes it stays below roughly 220 GFLOPS. This results in a curve that is almost flat with only slight upward movement.

The optimized CUDA curve shows the opposite trend because it reflects the strengths of GPU hardware. It climbs steeply at moderate N values when there is enough parallel work to fully occupy the GPU. After that, it stays high but shows small dips or bumps at sizes such as 1023, 1024, 1200, or 2047. These variations happen because some matrix sizes align cleanly with the GPU tile size, while others do not. When N is not a multiple of the tile dimension, the kernel must process more partial tiles and perform extra boundary checks, which lowers efficiency and reduces GFLOPS.

Overall, the optimized CUDA curve increases rapidly and stays high due to the massive parallelism of the GPU, while the BLAS curve stays low and smooth because it is constrained by CPU resources. The large gap between the two, along with their very different shapes, directly reflects the difference in hardware design. CPUs are optimized for control and low parallelism, while GPUs are built for high throughput and large, regular operations such as matrix multiplication.

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

The optimized CUDA curve shows several dips at matrix sizes such as 511, 1023, 1200, 1800, and 2047. These sizes do not divide evenly into the tile size used in the kernel, so they create many partial tiles. Partial tiles require extra boundary checks and introduce warp divergence, which lowers overall throughput and results in noticeable drops in GFLOPS.

Matrix sizes just above powers of two, such as 1025 and 2049, also show small performance decreases. These sizes produce leftover blocks that cannot fully occupy the GPU, reducing effective occupancy and slowing the kernel slightly.

In contrast, sizes like 512, 768, 1024, and 1500 align well with the GPU's block and warp structure. These cases allow the kernel to form clean, fully tiled grids with minimal overhead, which results in higher and more stable performance.

There is also a mild decline at very large N values (around 3000 to 4096). At this scale, memory traffic increases and cache reuse becomes less effective, so global memory bandwidth begins to limit performance.

Overall, the irregularities follow predictable patterns based on how well each matrix size fits the GPU's tiling scheme, occupancy behavior, and memory system efficiency.

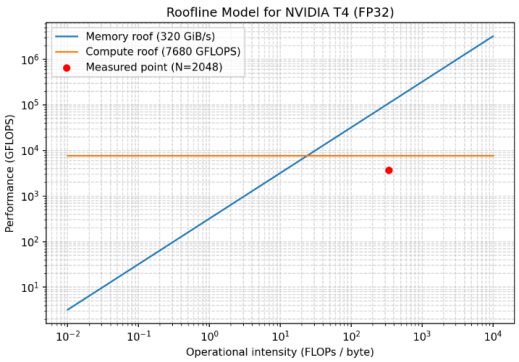
Section (5) - Roofline Analysis

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 what you achieved for the 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 FPMAC/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.

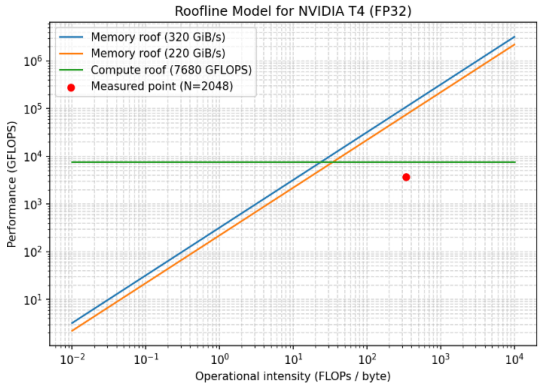
To construct the roofline model for the NVIDIA T4, we first calculate the theoretical FP32 peak performance based on the GPU's microarchitecture. The T4 contains 40 streaming multiprocessors, each with 64 FP32 cores. Altogether this provides 2560 FP32 units. Each core can issue one fused multiply-add per clock cycle, which counts as two floating point operations, and the GPU operates at a nominal frequency of 1.5 GHz. Multiplying these factors gives a theoretical peak of $2560 \times 2 \times 1.5 \times 10^9 = 7.68 \times 10^{12}$ FLOP/s, or 7680 GFLOPS. This horizontal line represents the compute roof in the roofline diagram. The memory roof uses the peak memory bandwidth

of 320 GiB/s reported by Jia et al., which determines the sloped line in the model. The point where the memory roof meets the compute roof defines the ridge point, which occurs at an operational intensity of approximately 24 FLOPs per byte. Below this intensity the GPU is limited by memory bandwidth, while above it, the GPU is compute bound.



Using the standard GEMM formulation, matrix multiplication at size N=2048 has a very high operational intensity, placing it well to the right of the ridge point in the compute-bound region of the roofline. When we plot our measured result for N=2048, which achieved 3719.64 GFLOPS, the point falls clearly below the theoretical 7680 GFLOPS peak but far above what the memory roof can supply. This confirms that the kernel is compute-bound but still does not fully saturate all FP32 units due to practical overheads such as instruction scheduling, register pressure, tiling inefficiencies, and imperfect pipeline utilization. The final roofline plot, therefore, shows the memory roof, the compute roof, and the measured point, illustrating how the achieved performance fits within the theoretical bounds of the T4 GPU.

Q5.b) Estimate the lowest value of q in ops/word (single precision float) at which the maximum theoretical performance is achieved. Consider that the actual BW is less than 320GB/sec - Jia et al. say it is 220 GiB/sec. Using this smaller BW, plot this roofline and estimate the new "q" value. How has the value of q been affected by the change in BW?



Using the theoretical peak bandwidth of 320 GiB/s, the ridge point of the roofline occurs where memory bandwidth and compute peak intersect. We solve
$P_{peak} = B_{peak} \cdot I$, with P_{peak} = 7680 GFLOPS and B_{peak} = 320 GiB/s, so
$I_{ridge} = 7680 / 320 = 24$ FLOPs per byte.
For single precision, one word is 4 bytes, so the corresponding q is
$q = I_{ridge} \cdot 4 = 24 \cdot 4 = 96$ ops per word.
This means that with the ideal 320 GiB/s bandwidth, any kernel with $q \geq 96$ ops/word can in principle, reach the theoretical compute peak of 7680 GFLOPS.

Jia et al. report a more realistic sustained bandwidth of about 220 GiB/s. Repeating the same calculation with B_{actual} = 220 GiB/s gives $I_{ridge\_actual} = 7680 / 220 \approx 34.9$ FLOPs per byte,
so the new q value is $q_{actual} = 34.9 \cdot 4 \approx 139.6 \approx 140$ ops per word.

Therefore, when we account for the lower real bandwidth, the minimum q required to achieve the maximum theoretical performance increases from about 96 ops/word to about 140 ops/word. In other words, the ridge point moves to the right on the roofline plot, so a kernel needs more computation per word of memory traffic before it becomes compute bound. Our GEMM at N = 2048 has an operational intensity of roughly $N/6 \approx 341$ FLOPs per byte, which corresponds to $q \approx 1365$ ops/word, so it remains well inside the compute bound region under both bandwidth assumptions.

Section(6) - Potential Future work

What ideas did you have that you did not have a chance to try?

We had several ideas we were excited about, but didn't have time to explore. In particular, we wanted to take a deeper dive into the cuBLAS GEMM implementation and experiment with more advanced optimization techniques used in production libraries, such as improved register tiling strategies, double-buffered shared memory pipelines, warp-specialized computation, and better utilization of Tensor Core–friendly memory layouts. These would have helped us close the remaining performance gap with cuBLAS.

We also hoped to explore auto-tuning different block sizes and tile shapes, as well as adding profiling-guided optimizations to better match the characteristics of Turing's memory hierarchy. Unfortunately, due to time constraints, we had to defer these ideas for future work.

Section(7) - Extra credits

Create a non-square matrix multiplication kernel that should be able to take 2 matrices A and B of dimensions MxK and KxN respectively and compute C of MxN. (4 points). What optimizations can you make with this kernel that would improve performance as compared to a simple square matrix multiplication? (2 points)

NA

Section (7) - References (cite all references used)

[1] Andrew Kerr, Duane Merrill, Julien Demouth, and John Tran. CUTLASS: Fast Linear Algebra in CUDA C++. NVIDIA Corporation, December 2017.

[2] Jianyu Huang, Chenhan Yu, and Robert van de Geijn. Implementing Strassen's Algorithm on NVIDIA Volta GPUs. arXiv:1808.07984, 2018. Available at: https://arxiv.org/pdf/1808.07984.pdf

[3] NVIDIA Corporation. CUDA C++ Programming Guide. Available at: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[4] NVIDIA Corporation. CUDA Toolkit Documentation. Available at: https://docs.nvidia.com/cuda/index.html

[5] Zhe Jia, Marco Maggioni, Benjamin R. Smith, and Daniele Paolo Scarpazza. Dissecting the NVIDIA Turing T4 GPU via Microbenchmarking. arXiv:1903.07486, 2019. Available at: https://arxiv.org/abs/1903.07486

[6] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures. Communications of the ACM, April 2009.

[7] NVIDIA Corporation. NVIDIA CUDA Best Practices Guide. Available at: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/