Section (1) - 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).

The program aims to solve the problem where we have two N by N matrices A and B, and we want to compute C = A * B efficiently by using a highly optimized kernel.

In the kernel, we start by allocating shared memory, defining two fixed-size arrays, As and Bs, for each thread block to store small tiles of matrices A and B, respectively. The dimensions of submatrix As are TILEDIM_MxTILEDIM_N, and those of submatrix Bs are TILEDIM_NxTILEDIM_N, where TILEDIM_M and TILEDIM_N are predefined tile dimensions. Consequently, each thread block calculates a TILEDIM_MxTILEDIM_N submatrix of matrix C by iterating through the matrices A and B. Essentially, a TILEDIM_M row tile of A is multiplied by a TILEDIM_N column tile of B, producing the TILEDIM_MxTILEDIM_N submatrix Cij of C.

Each thread in a thread block loads 32 elements from matrix A into As and 32 elements from matrix B into Bs. With each thread responsible for loading multiple elements, a thread block can fully load As and Bs into shared memory, accessible by all threads within the same block. The outer product of matrices is then performed to compute the partial sum of each element in the final submatrix C.

After completing the iterations, all elements in Cij are accurately calculated, and the results are stored back into matrix C. Once all thread blocks complete their computations, the final matrix C is returned by the kernel.

Our pseudocode is as follows:
Matrix C is mapped to different tiles of size TILE_DIM_M by TILE_DIM_N. Each thread block is mapped to each tile. Each thread block is responsible for calculating its tiles. If N does not divide evenly into a natural tile size, then allocate an extra tile in both the x and y directions.

```
Matmul{
    For every tile across the columns of A and down the rows of B:
        Each thread loads TILE_DIM_M /blockDim.y * TILE_DIM_N/blockDim.x data
            from A and B into shared memory As, Bs

        If N does not divide evenly, then load 0 if the indices are out of bounds, load A
        and B otherwise

        Syncthreads
```

```
            For  TILE_DIM_M /blockDim.y * TILE_DIM_N/blockDim.x:
                    Do multiply-add across the columns of As and rows of Bs and store the
                            results into register
            Syncthreads


    If indices within the bounds of N:
            Store the elements in the register into C.
}
```

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

Starting from the baseline kernel, we did the following:

1. Used shared memory
2. Handled cases where the thread blocking factor does not divide *n* evenly
3. Implemented the first version of 2D tiling
4. Improved 2D tiling by adding #pragma unroll and by reducing the number of unnecessary boundary checks

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.

(All performance in this section is computed for N=256)

1. The shared memory worked well because it significantly reduced the latency compared to global memory accesses, allowing for faster data retrieval and improved parallelism through cooperative thread operations. It boosted our performance from the naive kernel (**504 GFLOPs**) to **580 GFLOPs**.
2. The 2D tiling also worked well. Instead of loading a single element into shared memory, each thread loads multiple elements and computes multiple elements of matrix C. Consequently, fewer threads are required to perform the same computations as before, reducing the time needed for thread synchronization. With each thread handling more work, instruction-level parallelism is increased. Additionally, each thread loads the necessary data from shared memory into registers, improving data

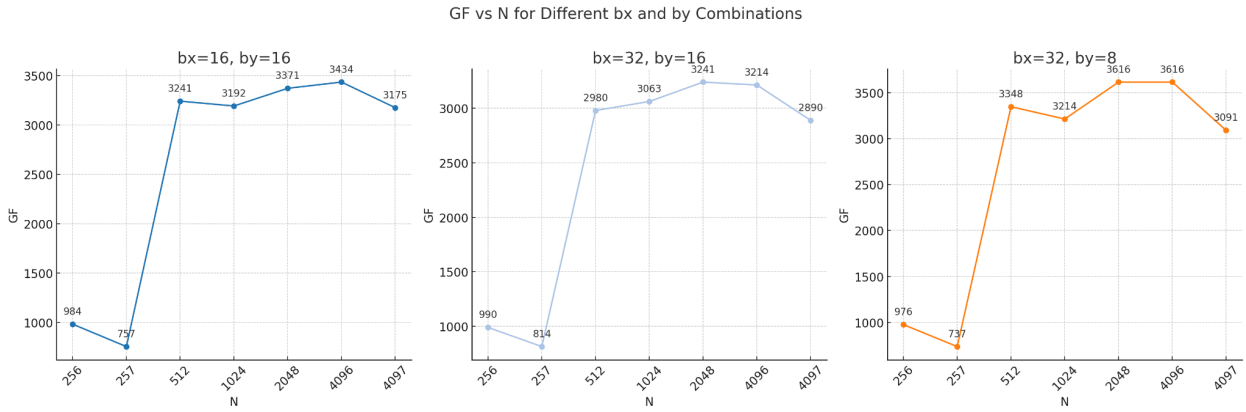
reuse efficiency. By implementing 2D tiling, we improved performance from **580 GFLOPs** to **946 GFLOPs.**

3. By adding the #pragma unroll, we instruct the compiler to unroll loops, effectively expanding the loop body multiple times to reduce loop control overhead and increase instruction-level parallelism. This boosts performance by minimizing the number of loop control instructions and allowing the GPU to execute more operations concurrently, thus improving throughput and reducing latency. The performance improved from **946 GFLOPs** to **999 GFLOPs.**

Section (2) - Result

Your implementation will be graded on its performance at the following matrix sizes: n=256, n=512, n=1024, n=2048, n=4096

Q2.a) For the problem sizes n= 256, 257, 512, 1024, 2048, 4096, and 4097, 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.



The constraint on thread block sizes is that they must be multiples of the tile sizes. We did not address cases where thread block sizes are not divisible by tile sizes, because we calculate the number of outputs each thread computes based on tile sizes divided by block sizes.

Q2.b) How do the results for n=256 and n=257 differ? How about n=4096 and n=4097? Please provide reasons for your observations.

The performance of N=257 and N=4097 drops significantly from N=256 and N=4096 respectively. It is because we need to pad 0s to the shared memory As and Bs so that the result can be calculated correctly. It adds extra blocks, which cause extra work of memory accessing and addition/multiplication, hence lowering the performance.

Q2.c) 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 in performance than others?

If the block size is too small, occupancy is low, leading to significant performance loss due to insufficient thread-level parallelism. Conversely, if the block size is too large, even though occupancy is high and the kernel is highly parallelized at the thread level, a lot of time is spent waiting for thread synchronization. Smaller block sizes allow more outputs per thread, which indicates higher instruction-level parallelism. Balancing instruction-level parallelism and thread-level parallelism, a 32*8 block size tends to be optimal.

For matrices of size 256, the optimal block size is 32*16 to better utilize register memory. Since 256 is small, the grid dimension is small, meaning there are only a few blocks, and synchronization duration does not increase significantly. For larger matrices, the increase in synchronization time outweighs the performance gain from a larger block size, making 32*8 the optimal block size for larger matrix computations.

Q2.d) 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	990	bx * by = 32 * 16
512	3348	bx * by = 32 * 8
1024	3214	bx * by = 32 * 8
2048	3616	bx * by = 32 * 8
4096	3616	bx * by = 32 * 8

Section (3)

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

N	naive	Our Result	Our Result/naive
256	504	990	1.96
512	530	3348	6.31
1024	468	3214	6.86
2048	423	3616	8.54

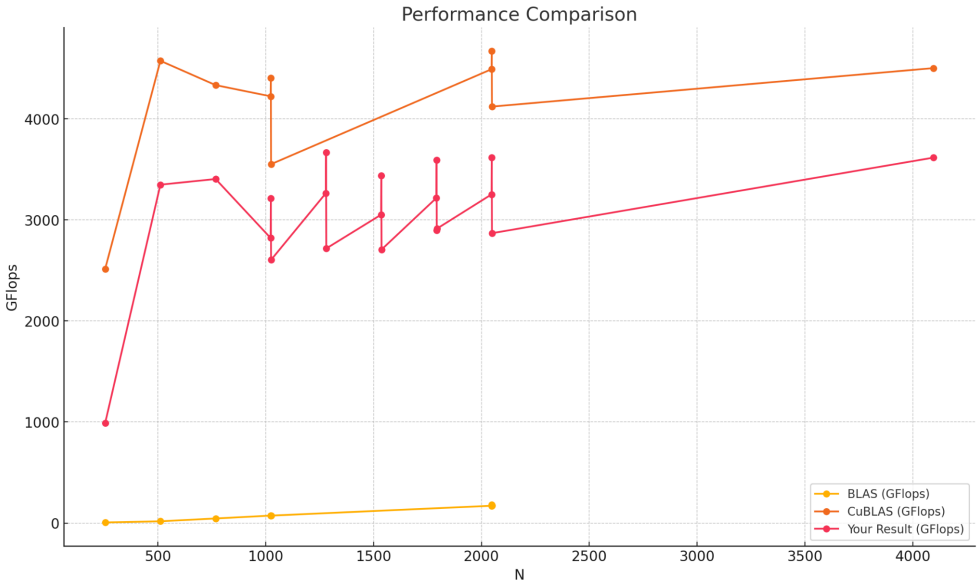
Section (4) - Analysis

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	Our Result (GFlops)

256	5.84	2515.3	990
512	17.4	4573.6	3348
768	45.3	4333.1	3404
1023	73.7	4222.5	2820
1024	73.6	4404.9	3214
1025	73.5	3551.0	2604
2047	171	4490.5	3252
2048	182	4669.8	3616
2049	175	4120.7	2868
4096		4501.6	3616
1279			3264
1280			3668
1281			2717
1535			3051
1536			3439
1537			2706
1791			3216
1792			3592
1793			2896
1794			2913

Plot:



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.

From the plot, we can observe that the shape of our curve is similar to the BLAS values. Both performance curves stabilize for matrix sizes larger than 1024. Notably, BLAS and our implementation exhibit similar irregularities around every matrix size that are multiples of our tile size. This is due to the use of tiling: when the matrix size is not a multiple of the tile size, computational resources are wasted due to zero padding.

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

As mentioned in the previous section, our implementation exhibits similar irregularities around every matrix size that are multiples of our tile size. It is because we need to pad 0s to the shared memory As and Bs so that the result can be calculated correctly. It adds extra blocks, which cause extra work of memory accessing and addition/multiplication, hence lowering the performance. For values like 257 and 4097, the work added is higher in

proportion to the original work compared with values like 255 and 4095, which is also reflected in the table.

Section (5)

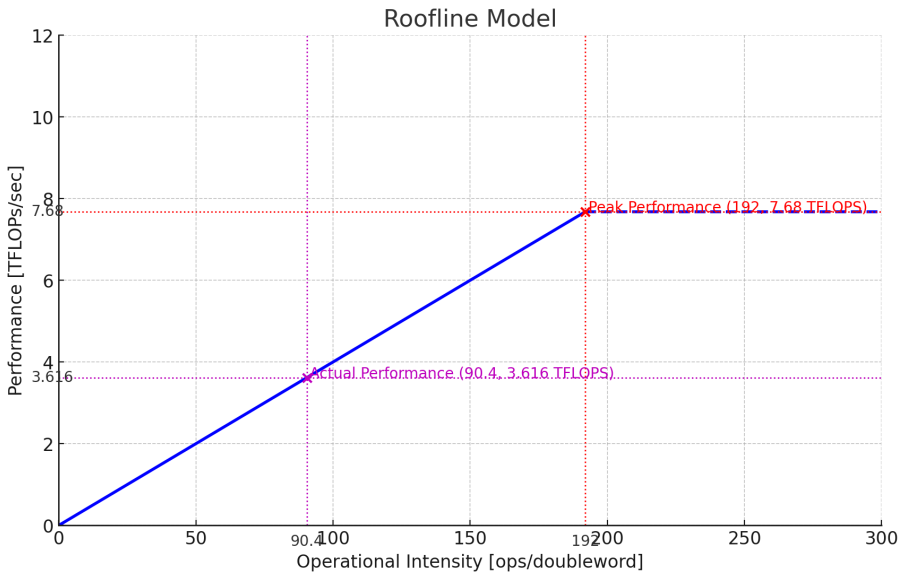
Q5.a)

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 figureplot 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.

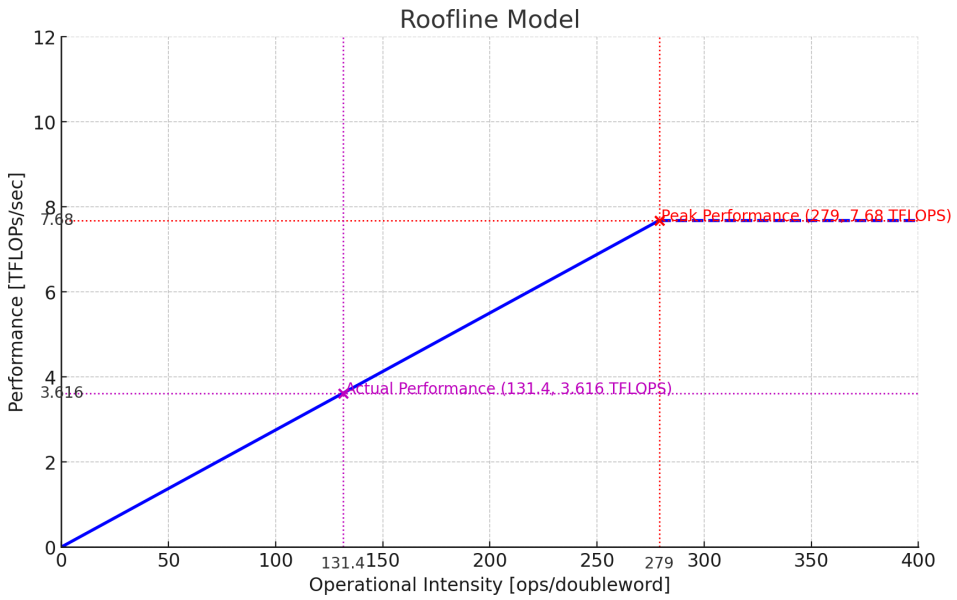
There are 40 SMs, and each SM has 64 SP cores, so there are 40*64=2560 SP cores. Each core can do 2 ops per cycle, so it has 2560*2=5120 FLOPs/cycle. Hence, the peak performance is 5120*1.5*10^9 = 7.68*10^12 = 7.68 TF/sec.

The bandwidth is 320 GiB/sec = 40 G doubles/sec. The peak performance has q value = 7.68 TF/sec / 40G doubles/sec = 192 ops/word. Our performance reaches 3616 GFLOP = 3.616 TF/sec, which has q value = 3.616 TF/sec / 40G doubles/sec = 90.4 ops/word. hence the roofline model will look like this:



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?

The peak performance is still 5120*1.5*10^9 = 7.68*10^12 = 7.68 TF/sec. The bandwidth is 220 GiB/sec = 27.5 G doubles/sec. The new q value = 7.68 TF/sec / 27.5G doubles/sec = 279 ops/word, which is higher than the previous model. Our performance reaches 3616 GFLOP = 3.616 TF/sec, which has q value = 3.616 TF/sec / 27.5G doubles/sec = 131.4 ops/word. hence the new roofline model will look like this:



Section(6) - Potential Future Work

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

If we had more time, we would address edge cases where block sizes are not divisible by tile sizes, allowing for more flexible block sizes. This would enable us to better utilize register memory and make better use of available resources.

Section(7) - Extra credits

EXTRA CREDIT points for higher T4 Performance:

n=512	3348**>=1500**	**2**
n=1024	3214**>=3100**	**2**
n=2048	3616**>=3500**	**2**

Section (8) - References (cite all references used)

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

[2] Cuda C++ programming guide -

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[3] Jia, Maggioni, Smith, Scarpazza, "Dissecting the NVidia Turing T4 GPU via Microbenchmarking": https://arxiv.org/abs/1903.07486