CSE 260: PA2 REPORT
Optimization of matrix multiplication for GPU

- By Abijith Yayavaram and Haiyu Zhang

Github repo link: https://github.com/cse260-fa22/pa2-ayayavaram-haz063

(We forgot to put it in the checkpoint)

Q.1.a)

We have used 2D tiling in our implementation where the matrices are divided into tiles of equal size. The dimensions of these tiles are defined in the 'mytypes.h' file in the form of TILEDIM_M, TILEDIM_N, and TILEDIM_K.

We chose a Tile dimension of 128*128*64 so that we could use the entire 64 KB of shared memory available in an SM. This is so that we can fit a tile each from A and B completely into the shared memory.

In the main code, we then define our stride lengths (number of elements to be interleaved) in both dimensions. Note that the strides should be the same as the thread block size. We also define mult_x, mult_y and mult_k which are the number of elements being computed by each thread in the respective directions.

The non-native implementation part of our code can be divided into 3 parts –

1. Taking care of edge cases and loading the tiles into shared memory.
2. Doing the actual multiplication
3. Storing back the results

All the above three parts of the code are parameterized for easy modification and debugging.

Taking care of edge cases –

1. When we load the tiles of A and B into As and Bs (shared memory), we look for whether the coordinates of A being loaded are outside the NxN region and if they are, we force that value to 0 and store it in the shared memory.

Note: We tried other approaches to padding such as using a different size of As and Bs for the edge cases, but that had huge negative impact on performance. We think this was because when we try to use variables in the for-loop condition, the compiler is not able to unroll the loop efficiently or perform other optimization techniques as opposed to when the loop condition had a constant value.

2. After padding, when we enter the for loop for the actual multiplication, we use a variable t_k which tells the program to whether use TILEDIM_K (as usual) or use a smaller value N-kk (for the edge cases). Essentially, we tell the program to ignore the padded values whenever we are near the edges of the tile. We do not use the padded values of 0 during multiplication because we found that there was a slight decrease in the performance when we did.
3. Similarly, while storing the values back into C, we only store the values which are within the matrix bounds.

Multiplication –

The multiplication is done for one tile at a time using outer product, and the elements of A and B used for the computation are interleaved by stride_x and stride_y elements.

Storing back the results –

Similar to how we loaded A and B into the shared memory and performed the multiplication, we store the results back into C in an interleaved fashion. This is done so that the threads use different banks of shared memory and global memory coalesces.

We use __syncthreads() at the end of loading the shared memory values and after multiplication because there is a true dependency when we must load all As and Bs before starting computation and an anti-dependency when we need to finish all computation before starting the next As and Bs.

Q.1.b and Q.1.c)

During the development of our code, we followed a step-by-step approach which helped us in understanding the basics of what we were supposed to do, before optimizing the code for edge cases, faster testing, etc.

We first started by modifying the starter code to use shared memory and reduce global memory accesses. At this point, each thread was still working on one element of C at a time but each thread was also loading an element of A and B into the shared memory. We found that this improved the performance of the code.

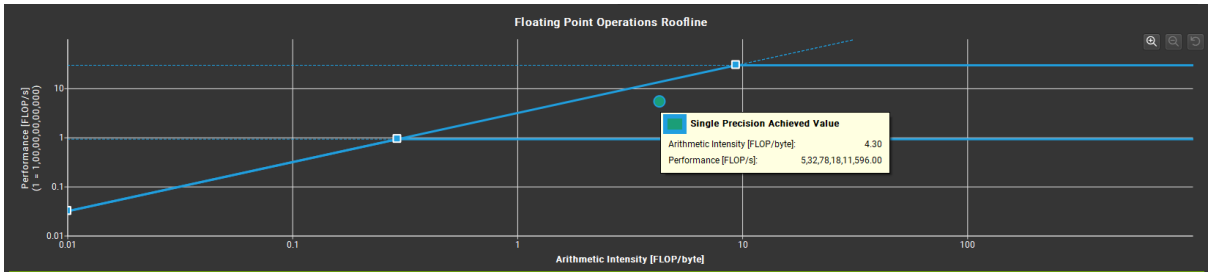
Our next step was to use 2D tiling along with shared memory. This enabled us to make each thread compute 4 values (2x2) of C. Now, each thread was also loading 4 values into As and Bs each. By doing this, we were increasing the ratio of multiplications/shared memory accesses as compared to the previous implementation; thus, improving the performance by almost 10 times.

After seeing the improvement in performance, we realised that we need increase our ILP while simultaneously play around with the Tile and block dimensions. So, we decided to parameterize the whole code and use loops so that we could just plug in our numbers once and measure the performance quickly. We also implemented padding during this step to take care of edge cases.

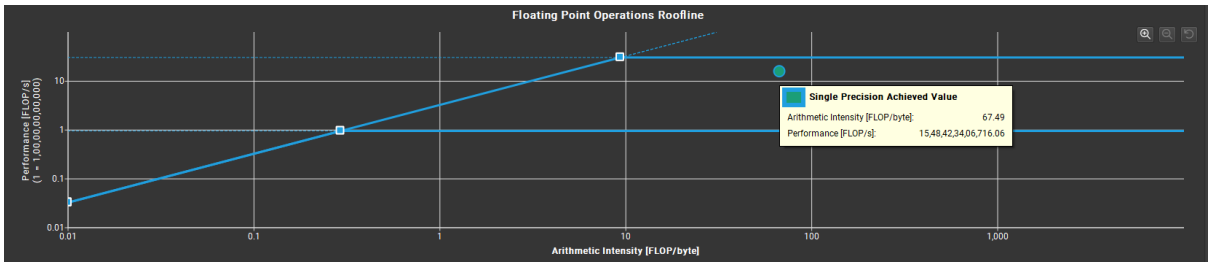
Testing out various tile dimensions -

While we began our experiments with equal tile dimensions in all directions, we quickly realised that we must choose a rectangular tile to optimize the utilization of shared memory. We first chose a tile size of 256*256*32 (in m, n and k directions) but found that a tile size of 128*128*64 works better.

Below are the roofline plots generated by NVIDIA Nsight for the 2 configurations (both are using a block size of 16x16).



(256x256x32)



(128x128x64)

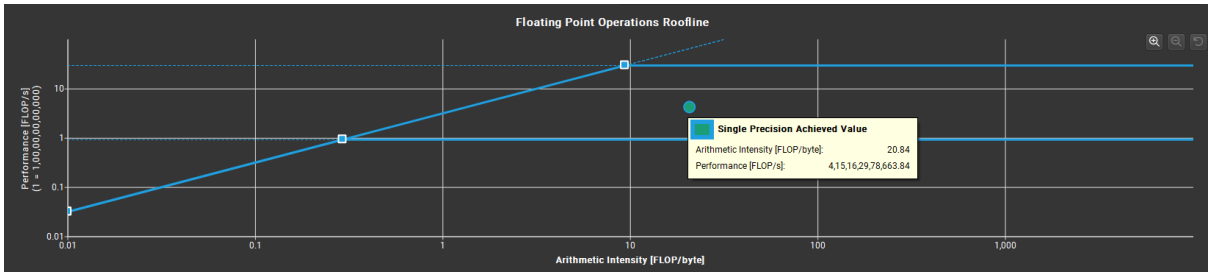
Through the roofline plots, we saw that we achieve a higher arithmetic intensity, memory bandwidth (higher slope) and performance in the 128x128x64 configuration. We also saw that the SM frequency was higher for 128x128x64 (598.05 MHz) compared to 256x256x32 (586.05 MHz).

<u>Testing out various block dimensions –</u>

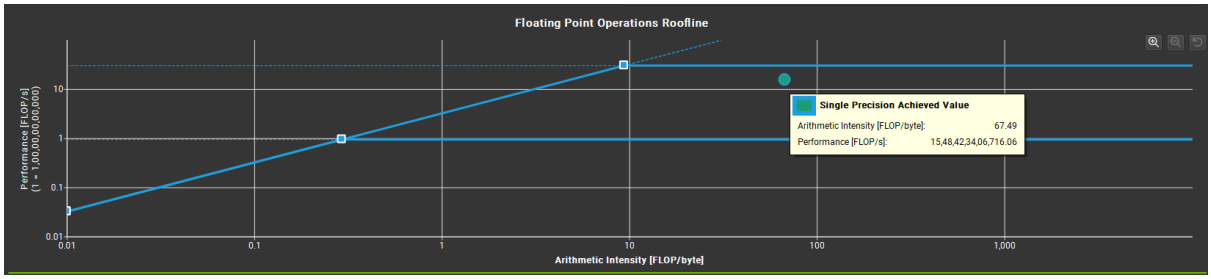
After finalizing the tile dimensions, we tried various block dimensions. We tried many configurations, but most of them did not have any unique results. So, we will talk about just three block sizes and compare their performances and resource utilization with the help of Nsight.

Thread block size	**8x8**	**16x16**	**32x32**
Achieved occupancy [%]	6.25	25	99.96
Compute (SM) Throughput [%]	28.38	56.99	83.03
Memory Throughput [%]	28.38	56.99	83.03
Registers used	255	96	64

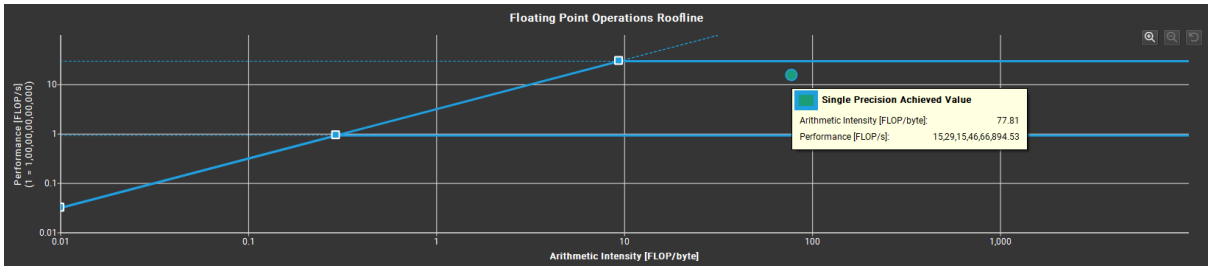
Roofline plots:



(8x8)



(16x16)



(32x32)

The peak performance for all three configurations is approximately the same. Through the roofline plots we see that 16x16 block size has the highest performance, followed by 32x32 in close second while 8x8 is a distant third.

We found the comparison of 16x16 and 32x32 configurations interesting because one would expect 32x32 block size to have a higher performance because it has a higher occupancy, compute throughput per SM and even memory throughput per SM as compared to 16x16. However, we find that is not the case in practice. This falls in line to Volkov's paper which was discussed in class.

This paper says that a higher occupancy does not always mean that we get a higher performance. Even with a lower occupancy and lesser utilization of the resources, if each thread has more arithmetic intensity, we can expect a similar or even higher peak performance. And this is exactly what we are observing in the case of 16x16 block size. Each thread is computing more multiplications as the tile size stays constant in both the block size configurations.

We might also infer from these results that using more registers while using the same amount of the shared memory is more important than trying to maximize the number of threads per block, as we see that the 16x16 block size uses 96 registers while the 32x32 block size uses only 64.

Q.2.a)

All the following configurations are using tile dimensions of 128x128x64

8x8 block

Matrix size (N)	Performance (GFlops)	Average SM frequency over 5 seconds (MHz)	Max SM frequency over 5 seconds (MHz)
256	125.60	1574.57	1590
512	477.20	1566.55	1590
1024	913.98	1522.37	1590

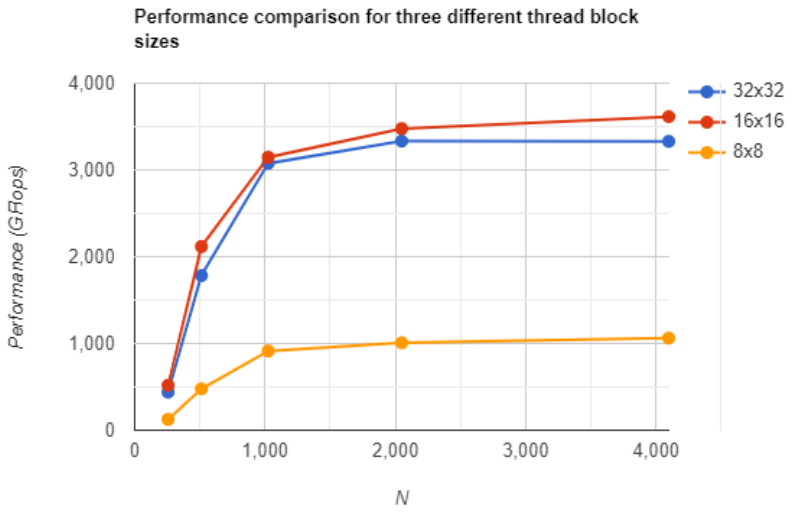
2048	1007.99	1489.21	1590
4096	1062.20	1422.79	1590

16x16 block

Matrix size (N)	Performance (GFlops)	Average SM frequency over 5 seconds (MHz)	Max SM frequency over 5 seconds (MHz)
256	518.74	1575.00	1590
512	2117.04	1572.16	1590
1024	3146.79	1397.37	1590
2048	3509.12	1348.42	1590
4096	3614.54	1221.25	1590

32x32 block

Matrix size (N)	Performance (GFlops)	Average SM frequency over 5 seconds (MHz)	Max SM frequency over 5 seconds (MHz)
256	439.19	1590.00	1590
512	1783.04	1560.00	1590
1024	3076.20	1327.37	1590
2048	3333.27	1240.86	1590
4096	3329.41	1158.50	1590



A few things to note:

1. The tile dimensions used for all 3 block sizes are: TILEDIM_M = TILEDIM_N = 128 and TILEDIM_K = 64. These dimensions were used to maximize the utilization of the 64KB shared memory.
2. All the three thread block sizes perfectly divide the tile dimensions.
   8x8 block size – There are 256 blocks in each tile
   16x16 block size – There are 64 blocks in each tile

32x32 block size – There are 16 blocks in each tile

3. All the above calculations are obtained after running the program for 5 seconds.

Limitations in choosing thread block size –

1. Block size should be able to perfectly divide a tile into blocks of equal sizes (power of 2).
2. Block size cannot exceed 32x32. We think it is because we cannot exceed 1024 threads per block in our Turing GPU.

Q.2.b)

In our results, we find that a block size of 16 x 16 gives us the highest performance for all the given values of N (256, 512, 1024, 2048, 4096). We find that this performance is closely followed by the 32x32 block size.

Some block sizes, thread sizes and geometries are better than others because they manage to utilize the resources of a GPU in a better manner. Some of the most important resources of an SM that limit occupancy and arithmetic intensity are shared memory, registers, and number of warps/blocks per SM.

In our assignment, we chose a Tile dimension of 128*128*64 so that we could use the entire 64 KB of shared memory available in an SM. We tried other shapes of a tile of the same size (for e.g. 256*256*32) and found that 128*128*64 gave the best performance.

Q.2.c)

N	**Peak GF**	**Thread block size**
256	518.74	16 x 16
512	2117.04	16 x 16
1024	3146.79	16 x 16
2048	3509.12	16 x 16
4096	3614.54	16 x 16

Q.3.a)

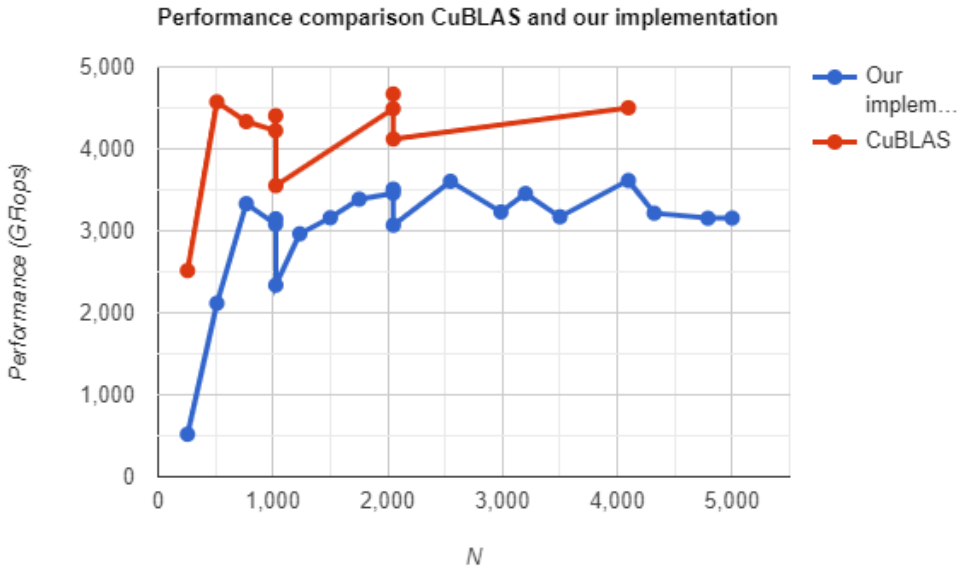
N	**Naïve Peak GF**	**Best results Peak GF (over 5 seconds)**
256	72.54	518.74
512	108.77	2117.04
1024	132.61	3146.79
2048	125.59	3509.12
4096	135.76	3614.54

Our best result's peak performance is considerably better than the naïve implementation for all values of N. The difference in performance for both the implementations grows as we increase the value of N from around 9x for N=256 to almost 27x times the performance obtained in the naïve implementation for N=4096.

Q.4.a)

Using the 16x16 block -

N	BLAS (GFlops)	CuBLAS (GFlops)	Our Result (GFlops)
256	5.84	2515.3	518.74
512	17.4	4573.6	2117.04
768	45.3	4333.1	3331.43
1023	73.7	4222.5	3079.93
1024	73.6	4404.9	3147.05
1025	73.5	3551.0	2337.46
1234			2963.73
1500			3160.00
1751			3386.90
2047	171	4490.5	3459.35
2048	182	4669.8	3509.12
2049	175	4120.7	3068.86
2546			3604.06
2986			3230.53
3200			3455.25
3500			3171.64
4096		4501.6	3614.54
4321			3215.57
4789			3153.67
5000			3155.75



Q.4.b and c)

From the graph plotted in the previous question, we can see that the shapes of the curves for CuBLAS and our implementation are almost identical. The only difference, other than the obvious overall downward shift of the curve, is that for our implementation the performance for N=768 was better than that for N=512, unlike CuBLAS - where the curve slightly dips from 512 to 768.

We theorize that the shapes are same because like our implementation, CuBLAS might also be zero padding the edge tiles for the values of N which are not exactly divisible by the tile dimensions. This padding causes some overhead which we can see in those dips at values of N which are not divisible by a power of 2.

The shapes might also be similar because both the implementations are bound by the limits of the hardware, like maximum number of threads per block, maximum registers available to a thread, size of the shared memory in an SM etc.

We also think that CuBLAS is able to achieve a much higher performance for even smaller matrices because it is using a different kernel with tile and block sizes more suitable for those values of N.

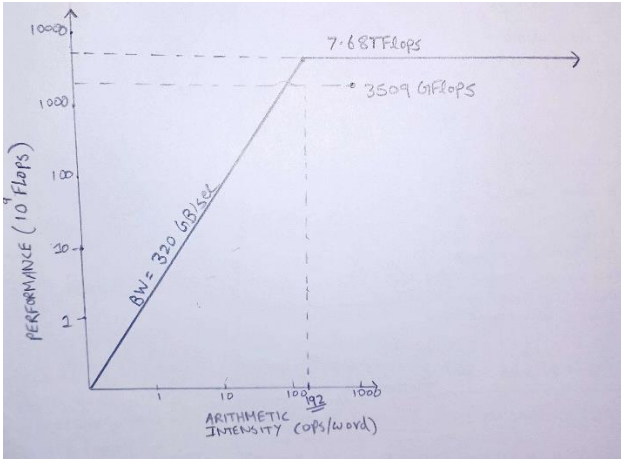
The irregularities in performance across the various values of N can be explained by the padding required for values of N which are not powers of 2, as mentioned above. Two clear examples are the cases of N = 1023, 1024, 1025 and N = 2047, 2048, 2049. We can see that the performance peaks locally at 1024 and 2048 which are both powers of 2 and hence require no padding.

Q.5.a)

Calculating peak performance:

- ⇨ 40 SMs * 64 SP FP cores * 1 FPMAD/cycle * 1.5 GHz
- ⇨ 40 SMs * 64 SP FP cores * 2 ops/cycle * 1.5 GHz
- ⇨ **7.68 TFlops**

Assuming the maximum memory bandwidth to be 320 GB/sec, we generate the following roofline plot:



Q.5.b)

q value: 7.68TFlops/(320 GB/sec)

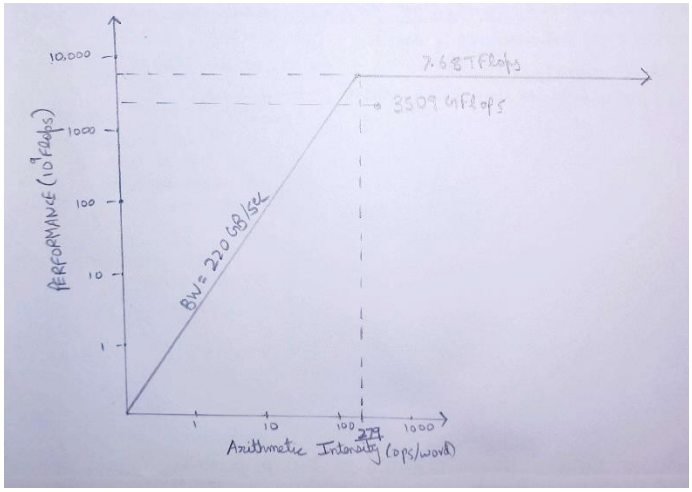
- ⇨ 7.68TFlops/(40 G words/sec) (Assuming 1 word = 8 Bytes)
- ⇨ **192 ops/word**

New q value after changing bandwidth:

- ⇨ 7.68 TFlops/(220 GB/sec)
- ⇨ 7.68 TFlops/(27.5 G words/sec)
- ⇨ **279.27 ops/word**

We can see that q has increased after decreasing the BW from 320 to 220 GB/sec.

Roofline plot with new BW:



Q.6) Some of the ideas that we wanted to try but did not get the chance to do so were –

1. Trying different patterns of interleaving and comparing their performances.
2. Making our code such that it accepts rectangular block dimensions.
3. Trying inner product instead of outer product.
4. Trying out some ideas from CUTLASS.

Q.7) References

1. Cuda C++ programming guide - https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

2. AWS Description of T4 - https://aws.amazon.com/blogs/aws/now-available-ec2-instances-g4-with-nvidia-t4-tensor-core-gpus/

3. Volkov, Better Performance at lower Occupancy, GTC2010 - https://www.nvidia.com/content/GTC-2010/pdfs/2238_GTC2010.pdf

4. NVIDIA Nsight documentation - https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#roofline