Note: Conventions used in this analysis

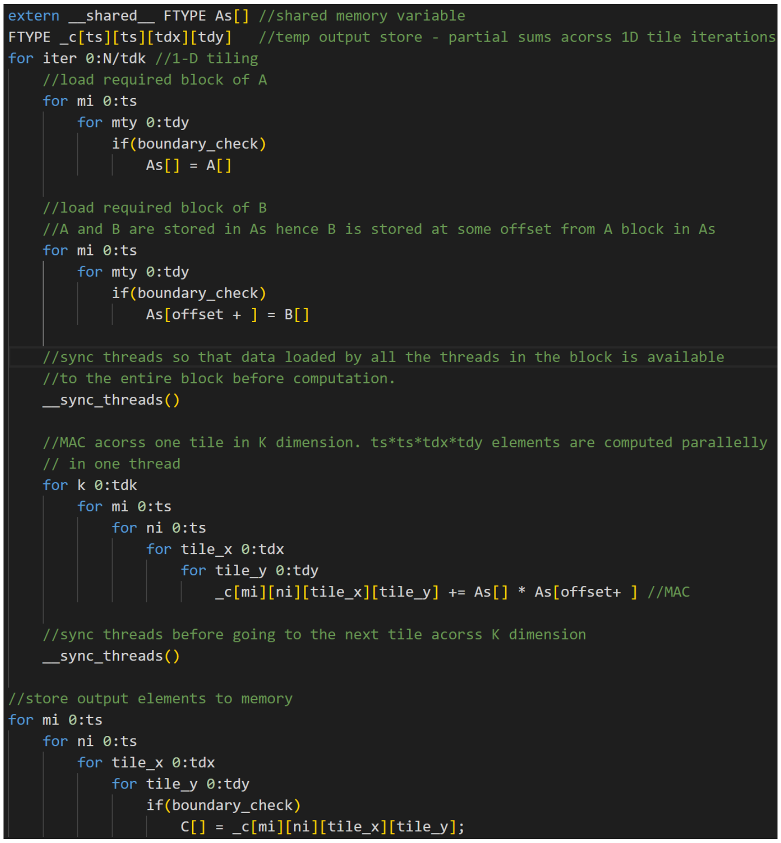
- **ts** - tile scaling
- **bx** - block dimension along x, **by** - block dimension along y
- **tdx** - tile dimension along x, **tdy -** tile dimension along y (tile dimensions say the number of adjacent elements which are computed together in one thread. For example when tdx=2 and tdy=2, C[0,0], C[0,1], C[1,0] and C[1,1] are computed in the first thread (bx 0 by 0 tx 0 ty 0) whereas if tdx=1 and tdy=1, only C[0,0] is compute in the first thread (bx0 by 0 tx 0 ty 0)
- **ILP** - Instruction Level Parallelism, **TLP** - Thread Level Parallelism

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).

In order to improve performance, we used shared memory and data reuse across threads in a block by tiling and also in order to utilize the maximum shared memory per block and maximum number of registers per thread, we computed multiple elements of the output matrix in one thread. We follow the inner product and C is computed completely before writing back to memory (no partial sum is stored back to memory and read later).

The control flow of the program is explained in the following pseudo code



```
extern __shared__ FTYPE As[] //shared memory variable
FTYPE _c[ts][ts][tdx][tdy]   //temp output store - partial sums acorss 1D tile iterations
for iter 0:N/tdk //1-D tiling
    //load required block of A
    for mi 0:ts
        for mty 0:tdy
            if(boundary_check)
                As[] = A[]

    //load required block of B
    //A and B are stored in As hence B is stored at some offset from A block in As
    for mi 0:ts
        for mty 0:tdy
            if(boundary_check)
                As[offset + ] = B[]

    //sync threads so that data loaded by all the threads in the block is available
    //to the entire block before computation.
    __sync_threads()

    //MAC acorss one tile in K dimension. ts*ts*tdx*tdy elements are computed parallelly
    // in one thread
    for k 0:tdk
        for mi 0:ts
            for ni 0:ts
                for tile_x 0:tdx
                    for tile_y 0:tdy
                        _c[mi][ni][tile_x][tile_y] += As[] * As[offset+ ] //MAC

    //sync threads before going to the next tile acorss K dimension
    __sync_threads()

//store output elements to memory
for mi 0:ts
    for ni 0:ts
        for tile_x 0:tdx
            for tile_y 0:tdy
                if(boundary_check)
                    C[] = _c[mi][ni][tile_x][tile_y];
```

Based on ts, tdx, tdy, tdk, bx and by, we have to pad (add zeroes) for certain matrix sizes. When we are loading data (A and B) to share memory, if the row and column dimensions exceeds the input matrix size, we will store zero in the shared memory. Hence this boundary condition is added when loading A and B to shared memory. Once padded (stored as zero in shared memory), we can compute all the elements of C without any errors for a kernel configuration. As we have computed more than required elements of C, we need a similar boundary check to validate the row and column dimensions of C being stored. If the dimensions are within the required matrix size, the computed value is written to memory else it is skipped.

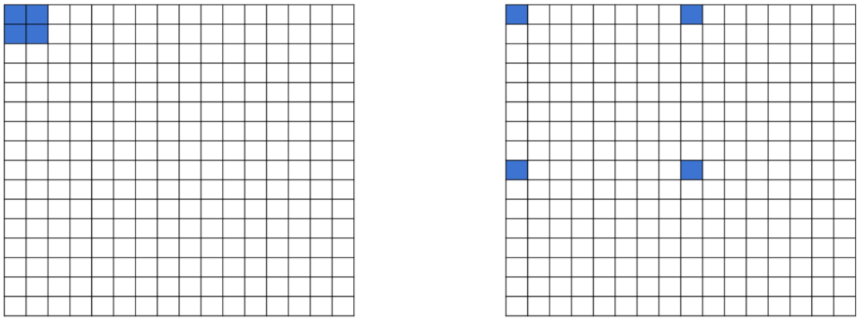
Q1.b) What was your development process? What ideas did you try during development?

As the amount of shared memory is limited, we cannot store a complete row of A and a complete column of B required for computing one element in C. So, first 1D tiling is done across K dimension (across columns of A and rows of B) to efficiently store the inputs in shared memory and compute C. With the tile scaling dimension across K as 16, 16 elements of A and 16 elements of B are stored in the shared memory. As adjacent threads in a block are going to calculate adjacent elements in C, they share elements of A and B. Hence, each thread need not load all 32 (16 elements of A and 16 elements of B). Each thread loads just one element of A and one element of B and stores it in the shared memory. __syncthreads() is used to sync all the threads after loading and as shared memory is per block, the data fetched by each thread is visible to all the threads after syncthreads and can be used for computation. This is repeated for N/16 times to get one element of C. With this 1D tiling across K and using shared memory, the performance of N=2048 is increased from 337 GFlops to 997 GFlops for tile dimension 16 and thread block dimension 16 x 16.

Now tile scaling is done, to increase the number of throughput per thread as we have 64 KB shared memory per block and 255 registers per thread. TIle scaling is calculating more than one element of C which are interleaved. Based on the tile scaling parameter, each thread will fetch required elements of A and B. As tile scaling factor is doubled, the performance also doubled. For tile scaling factor 8, where 64 elements are computed per thread and thread block dimensions 16 x 16, the performance for N=2048 increased to around 3500 GFlops.

We then added boundary conditions to support all the matrix sizes which added an if statement before loading any element from memory and before storing computed output to memory. Though we made the if condition simple enough so that the compiler can add predicates based on thread indices, the performance for N=2048 dropped to around 3300 GFlops.

To improve the performance further, we can increase the number of elements of C computed per thread by increasing tile dimension along x (tdx) and tile dimension along y (tdy) (2D tiling). 2D tiling reuses data as adjacent rows and columns in C use the same rows and columns in A and B respectively. Tile Scaling doesn’t help in data reuse but helps to utilize the shared memory per block and number of registers per thread.



2D tiling (on the left) and tile scaling(on the right)

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

We expected an increase in performance for all the matrix sizes with 2D tiling but smaller matrices did not perform well whereas the performance of higher matrices increased significantly. For 2D tile dimension 4 x 3, tile scaling factor 4 and thread block dimensions 16 x 16, the performance for N=2048 increased to 3834.36 GFlops and for N=4096 the performance increased from 3172 GFlops to 4236 GFlops. Whereas for N=256, the performance dropped from 462.2 GFlops to 206.9 GFlops.

We also tried to use the following __restrict__ , __syncwarp(), manually unrolling the loops, using pragma directives for unrolling in the code but there is no significant performance increase in performance.

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, 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.

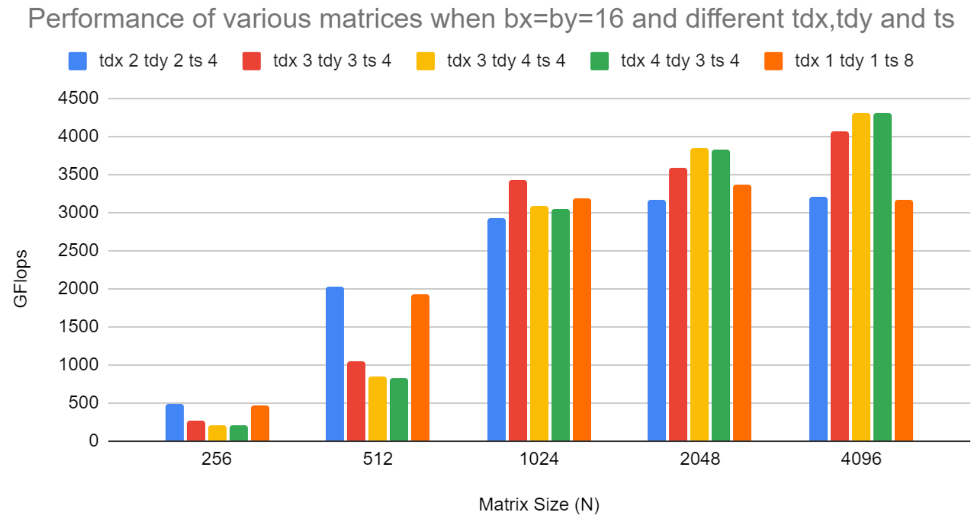
We have plotted the performance for 3 different block sizes - separate plots for each **bx, by** pair.

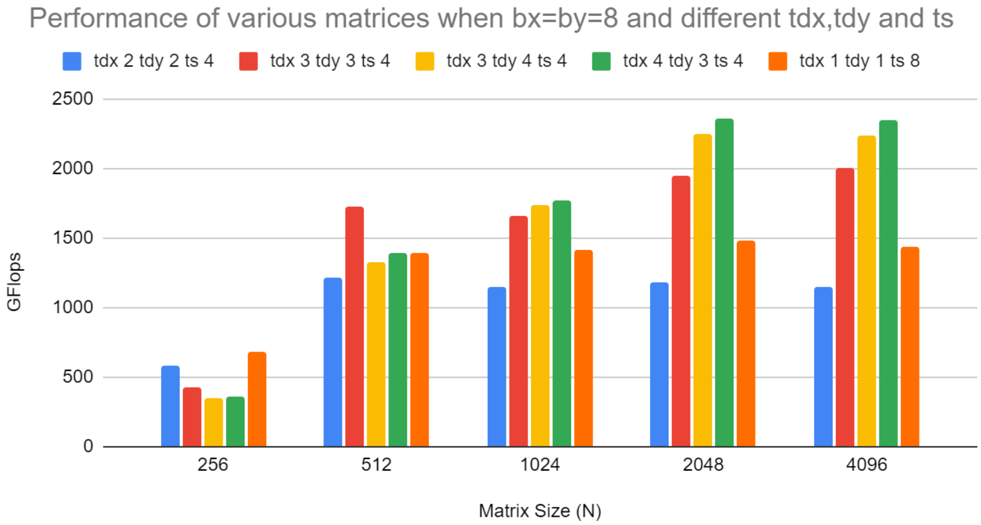
- bx=16 by=16
- bx=8 by=8
- bx=4 by=4

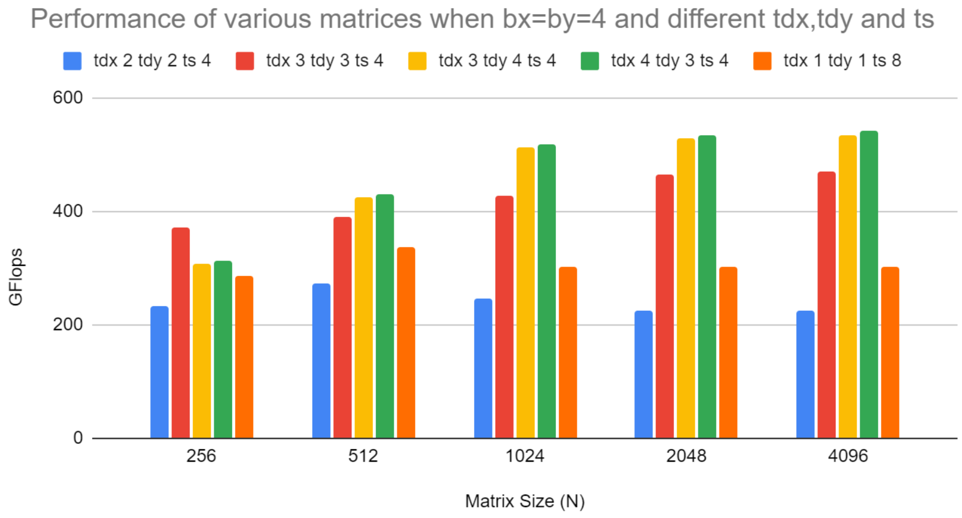
For each of these block sizes we have experimented with different values of **tdx, tdy, ts.**

- It is evident that bx=16, by=16 gives us the best performance.
- For each of these block sizes, the combination with high value of tdx * tdy * ts * ts gives the optimal result, provided there are enough resources available.

- However, it should be noted that incrementing these values after a point is detrimental as each thread is limited by the number of registers **(255).**
- Thus **tdx=4 tdy=3** and **tdx=3, tdy=4** consistently perform better for all block sizes.







If your code has limitations on thread block size, please state the reason for that limitation.

- Maximum possible threads possible per block is 1024 (=32*32) but we couldn't use all the threads per blocks as we have many optimizations such as tile scaling and 2D tiling which would need more shared memory. Hence, we used 16*16 as maximum thread dimension in our code.

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

We see that for sizes for smaller sizes (N~256), a smaller thread block size (8x8) yields good performance. The highest performance is yielded by doing tile scaling(8x8) and no 2D tiling at all. So we are doing ILP rather than TLP. Smaller block sizes ensure less waiting for thread synchronization for a block. (ILP level performance is more dominant)

For sizes(N ~ 512), 16x16 block size yields good performance. This also happens by doing tile scaling(8x8) and no 2d tiling at all. (Balance of ILP and TLP performance)

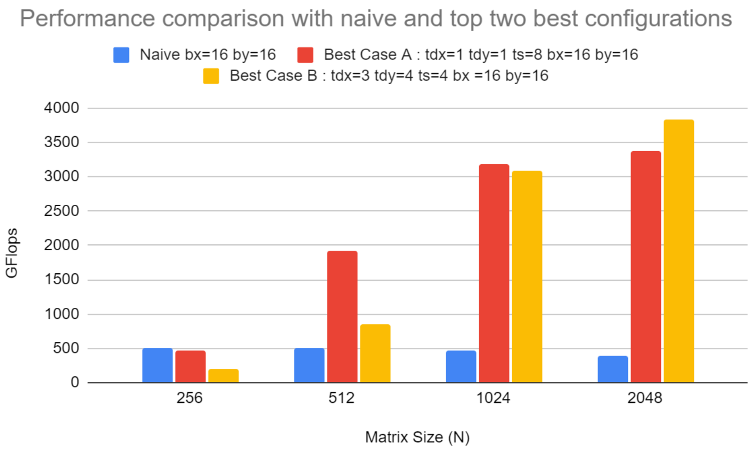
For larger sizes(N>=1024), 16x16 block sizes. This happens by doing tile scaling and 2D tiling (Balance of ILP and TLP performance)

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	684.26	8*8 (tdx=1 tdy=1 ts=8)
512	2032.09	16*16 (tdx=1 tdy=1 ts=8)
1024	3424.79	16*16 (tdx=3 tdy=3 ts=4)
2048	3841.52	16*16 (tdx=4 tdy=3 ts=4)
4096	4503.5	16*16 (tdx=4 tdy=3 ts=4)

Section (3)

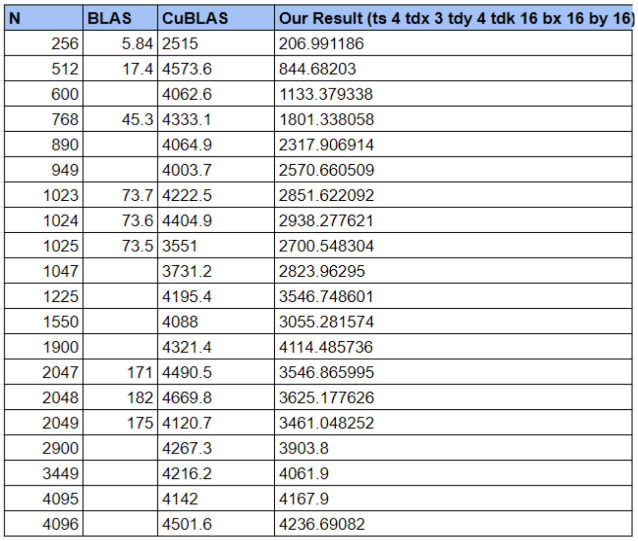
Q3.a) For n=256, 512, 1024, and 2048 quantitatively compare your best result with the naive Implementation.



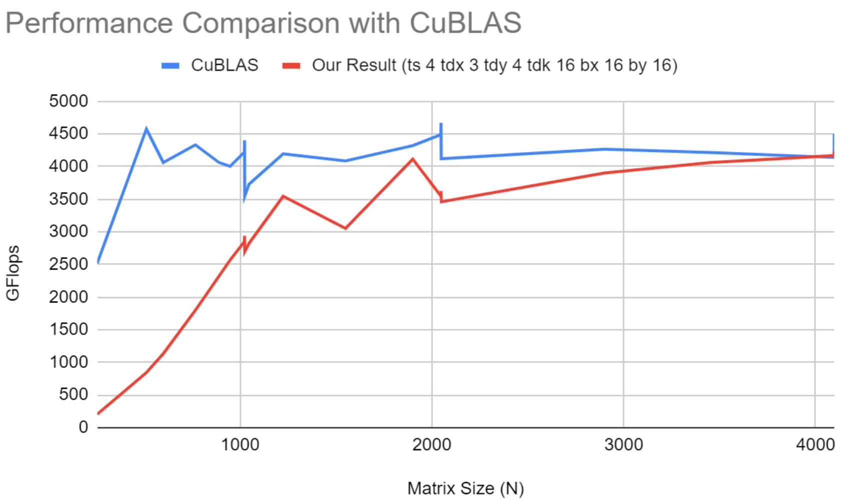
Naive result didn't vary much for various matrix sizes as it was loading each element by element for A and B matrices and one element of C is computed per thread. Whereas in our implementation the performance is increasing as N increases because larger matrices require more computation and the benefits of using shared memory, tiling and tile scaling (balance of ILP and TLP become more dominant.)

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	CuBLAS	Our Result (ts 4 tdx 3 tdy 4 tdk 16 bx 16 by 16)
256	5.84	2515	206.991186
512	17.4	4573.6	844.68203
600		4062.6	1133.379338
768	45.3	4333.1	1801.338058
890		4064.9	2317.906914
949		4003.7	2570.660509
1023	73.7	4222.5	2851.622092
1024	73.6	4404.9	2938.277621
1025	73.5	3551	2700.548304
1047		3731.2	2823.96295
1225		4195.4	3546.748601
1550		4088	3055.281574
1900		4321.4	4114.485736
2047	171	4490.5	3546.865995
2048	182	4669.8	3625.177626
2049	175	4120.7	3461.048252
2900		4267.3	3903.8
3449		4216.2	4061.9
4095		4142	4167.9
4096		4501.6	4236.69082



(As BLAS performance is much lesser than CuBLAS and our implementation and as we don't have few intermediate values we skipped BLAS in the above plot. Within the BLAS performance range, for increasing matrix size, the performance is increasing similar to CuBLAS curve)

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

Shape of our implementation curve is similar to BLAS and CuBLAS where it increases for increasing matrix sizes and there are a few unusual dips. All the three (BLAS, CuBLAS and our implementation) do optimizations to improve memory latency by utilizing the memory hierarchy, data reuse and ILP (more computation per thread). When more computation is required these optimizations offer more increase in performance.

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

Performance is computed by measuring the time for computation and the number of useful computations performed in that time.

The dips in our implementation are particularly for values - 2^p+k i.e. some offsets near the even powers of 2 e.g. 1025, 2049. The reason being these matrices are heavy padded as compared to other matrices that are multiples of appropriate parameters. The padding is done considering tile dimension, thread block dimensions and tile scaling so that they can be appropriately mapped and run in the kernel. As a result more conditionals creep and also there is redundant computation which ultimately gets ignored.

The peaks happen when there is no padding and when there is higher data reuse. Here the data reuse is decided by the tile dimensions and thread block dimensions.

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 in=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.

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?

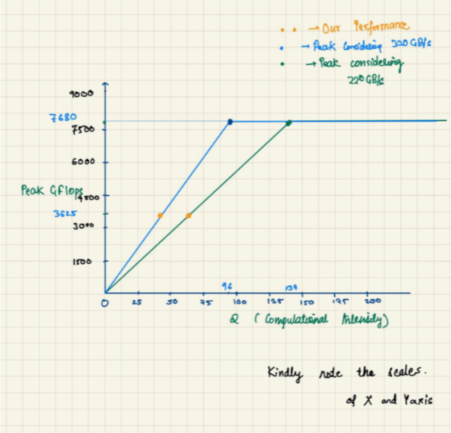
Peak performance would be (1.5 GHz)*(40 SMs)*(64 SP cores)*(1 FMAD/cycle == 2 ops/cycle) = > **7680.0 GHz**

Matrix Size (2048, 2048) - Performance achieve by our implementation 3625.177626

Q theoretical (considering max bandwidth = 320 GB/s) -> 7680 GHz / (320 GB/s / 4 bytes per word) -> **96 ops/word**

Considering this, we achieve -> (3625.177626 / 7680) * 96 = **45.314 ops/word**

Qv (considering actual bandwidth = 220 GB/s) -> 7680 / (220 / 4 bytes per word) -> **139 ops / word**
Considering this , we achieve -> (3625.177626 / 7680) * 139 = **65.6119 ops/word**



Section(6) - Potential Future work

- We wanted to try warp level tiling after reading the CUTLASS article. It is stated that it achieves even a higher performance then that is possible by thread level computation model and mapping of the problem.
- Also we would like to simplify our code so that it can be maintained easily.
- Avoid logic to minimize bank conflicts by offsetting the way data is stored in shared memory.
- Also write flexible kernels such that parameters are chosen according to the dimensions of the matrix.
- Finally, do micro optimizations for e.g. use bit operations

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. (2 points). What optimizations can you make with this kernel that would improve performance as compared to a simple square matrix multiplication? (2 points)

We have implemented a non-square matrix multiplication kernel that takes matrices A (M, K) and B(K,N) and computes C(M,N).

The source code is in the __bonus__ folder in the repository and should be run as our normal code. Run `./compile_run.sh 1` to compile it and run evaluations of sizes mentioned in `./gen_results.sh`

- If the matrices are rectangular we are at much more liberty at tweaking the parameters that determine the ILP and TLP in the computation.
- For example, we can do thread scaling that resembles the shape of the matrices. For example for a wider A we can do more thread scaling along X. Also the same goes with 2d tiling.

- We believe that for taller A and wider B, having more threads per block will be helpful and for wider A and taller B, doing more ILP operations would be helpful. This needs to be checked though.

Section (8) - **References**

[1] Andrew Kerr, Duane Merrill, Julien Demouth and John Tran , CUTLASS: Fast Linear Algebra in Cuda C++, December 2017
[2] Volkov, Demmel, Benchmarking GPUs to Tune Dense Linear Algebra, SC2008
[3] CSE 260 SP 22 Lecture and discussion slides
[4] Cuda documentation - https://docs.nvidia.com/cuda/index.html
[5] Cuda C++ programming guide - https://docs.nvidia.com/cuda/cuda-c-programmingguide/index.html
[6] Better Performance at Lower Occupancy (nvidia.com)