Section (1) - Development Flow

Q1.a) Describe how your program works.
Default Parameter Setting:

- **Block Size:**
    - bx (16): number of threads in x direction per block (equal to blockDim.x and BLOCKDIM_X)
    - by (16): number of threads in y direction per block (equal to blockDim.y and BLOCKDIM_Y)

  Therefore, one block consists of 16*16=256 threads in total.

- **Thread Size:**
    - TILESCALE_M: 8
    - TILESCALE_N: 8

  In our final implementation, each thread can calculate multiple elements in C. For each thread, it will calculate a TILESCALE_M*TILESCALE_N submatrix in C, which is 8*8 in our setting.

- **Tile Size:**
    - TILEDIM_M (by*TILESCALE_M=128): number of elements in x direction per block
    - TILEDIM_N (bx*TILESCALE_N=128): number of elements in y direction per block
    - TILEDIM_K (64): the width of subpanels in A and B tiling, which is similar to Kc in PA1

  One entire block is responsible for a TILEDIM_M*TILEDIM_N=128*128 submatrix in C. More specifically, one block calculates a dot between a TILEDIM_M*TILEDIM_K=128*64 submatrix in A (called As) and a TILEDIM_K*TILEDIM_N=64*128 submatrices in B (called Bs). In our implementation, As and Bs will be loaded into the shared memory region. Furthermore, we have implemented a 2D tiling (TILEDIM_K > 1) method. Therefore, each thread in the block calculates a dot between a TILESCALE_M*TILEDIM_K=8*64 subpanel in As and a TILEDIM_K*TILESCALE_N=64*8 subpanel in Bs, which generates a 8*8 submatrix in C, described in our “mmpy_kernel.cu” line #129 to #162.

Program Flow in matMul():

- **Step 0: Initialization**
  In initialization, our job is to allocate the correct size of shared memory for As and Bs. As mentioned in parameter settings, the size of As should be 128*64 float numbers and the size of Bs should be 64*128 float numbers. Because the size of each float value is 4 byte, the total size of As and Bs will be 64KB, which helps them make full use of the shared memory on our Tesla T4.
  Besides, we also declare the submatrix C (Cij) calculated by each thread, whose size is 8*8.

- **Step 1: Load A and B subpanel into shared memory**

In the core implementation, the first step is to load data from A and B matrices into the pre-allocated shared memory region As and Bs respectively.
In the loading process, each thread loads one point at one time. Take As as an example. Each block has 256 threads in total, so we can load 256 elements from A to As simultaneously at one time. Because the width of As is TILEDIM_K=64, 256 threads can fill 4 rows of As at one time. That is to say, in the i-th iteration, all the 256 threads can fill Row #4i - Row #(4i + 3) of As. Therefore, to fill the entire As, we need TILEDIM_M/4=128/4=32 iterations for each thread. For loading Bs, the method is similar. Our code in "mmpy_kernel.cu" line #101 to #125 describes this process.

- **Step 2: Calculate submatrix in C**
  The next step is to carry out a matrix dot between As and Bs. Each thread handles a dot between a 8*64 sub-region in As and 64*8 sub-region in Bs. For example, Thread (0, 0) calculates dot between As(0,0)(7,63) and Bs(0, 0)(63,7), Thread (0, 1) calculates dot between As(0,0)(7,63) and Bs(0, 8)(63,15), and Thread (1, 0) calculates between As(8,0)(15,63) and Bs(0, 0)(63,7). For each thread, the entire calculation costs 64 iterations. For each iterations, the thread calculates the dot between one column (8 elements) in the 8*64 As sub-region and one row (8 elements) in the 64*8 Bs sub-region, and accumulates the results to Cij.

- **Step 3: Store the dot results to the final C matrix**
  Finally, we calculate the exact location of the elements in C based on the current block ID and thread ID, and store the results in Cij back to the C matrix.

How we handle edge cases:
To handle cases when N does not divide evenly into a natural tile size (128 in our default setting). We use a zero padding method. For example, for N=128k-1, we pad 0 after the last row and last column to and turn it into a 128k*128k matrix, which can make the program use k*k blocks in total. However, for N=128k+1, we have to pad 0 and expand it into a 128(k+1)*128(k+1) matrix, which will cost (2k+1) more blocks than N=128k-1 or 128k cases.

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

1. We first get familiar with the naive implementation and test performance with different block sizes, which helps us understand the relationship between the performance and block size.
2. We implement a basic shared memory-based method to allow faster access for A and B matrices. In this basic method, each thread handles one element in C.
3. We make each thread calculate multiple elements, which optimizes the register use.

4. We set the TILEDIM_K to 64 and achieve 2D tiling, which can help memory coalescing when loading elements to As and Bs.
5. We use a zero padding method to handle edge cases when N can not be divided by a tile size, as described in Q1.a.
6. We add `#pragma unroll` keyword to unroll the loop and optimize iteration performance.
7. As described in Q1.a step2, we find that in each iteration, the thread needs to calculate dot between one column of a 8*64 sub-region in As and one row of a 64*8 sub-region in Bs. Therefore, to optimize register use, we choose to preload the column and row separately into 2 1D arrays and then perform dot between corresponding elements in the arrays.
8. Our zero padding method introduces if-else statements, but GPU is not good at branch prediction. Therefore, we replace them with conditional operators to eliminate branches as much as possible.
9. Finally, we tune other parameters including bx, by, TILESCALE_M and TILESCALE_N to make As and Bs make full use of the shared memory on the GPU and achieve best performance.

Q1.c) What ideas worked well, what didn't work well, and why.

Most ideas listed in Q2.c work well and improve the performance.

The highest performance improvement we have gained is because of making good use of the memory hierarchy in GPU architecture.

Assume A is M * K and B is K*N and C is M*N.

For naive implementation, every time we calculate one element of C, we need to load one row of A and one column of B from global memory, therefore the total load from global memory is M*N* 2K. This is really costly as we are repeatedly loading the same row and column from global memory.

Then we implement shared memory and repeatedly use the elements in the shared memory (As and Bs). We can save the amount of time that we load from the global memory by 1 / (TILEDIM_M*TILEDIM_K + TILEDIM_K*TILEDIM_N). After implementing shared memory, every element only needed to be loaded from main memory once.

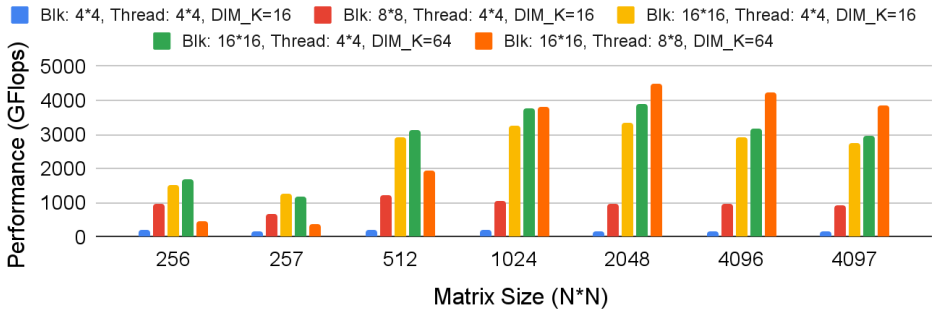
And finally, we realize using shared memory is also slow, we could load the elements from the register more often. Inspired by what is shown in the class, we could load elements from shared

memory to the register and access what is in the register more often. Assume the size of register is TILESCALE_M * TILESCALE_N, we are saving the amount of time that we load from the shared memory by 1 / (TILESCALE_M * TILESCALE_N). The detailed process is from Q1(a).

To summarize, we went from loading less from global memory to less from shared memory to more from register.

Section (2) - Result

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.



We compare the performance under different block sizes, thread sizes and TILEDIM_K. Please refer to Q1.a for their meaning and relationship.

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.

When n=257, the performance is 17% lower when n=256. n=4097 is 10% lower than n=4096. This is because we use the zero padding method to handle edge cases. Therefore in n=257 and n=4097 cases, we expand the matrix to 384*384 and 4224*4224 matrices. We will waste some computation resources to calculate dots between 0 and 0, which harms the overall performance. What's more, the ratio of padding area size to the entire matrix size after padding when n=257 is bigger than that of n=4097. Therefore, the performance drops more when n=257.

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 performance than others?

For N<1024, smaller thread size can achieve better performance because we can use more blocks to increase the parallelism. However, for N>=1024, the performance of smaller thread size (4*4) is lower than that of 8*8. We believe that this is because for smaller thread sizes, we will allocate more blocks. When N is larger, the number of blocks may exceed the limitation of block numbers that one SM can have.

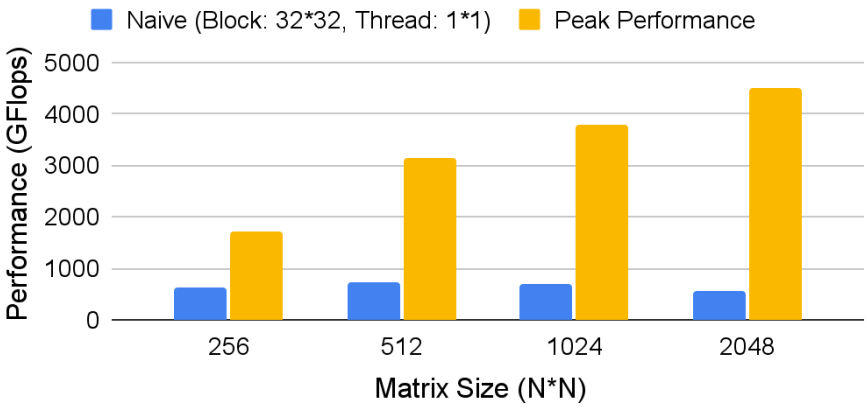
Q2.d) Mention the peak GF achieved and the corresponding thread block size for each matrix size analyzed in the previous question in a table like this.

TILEDIM_K is set to 64 for all.

N	Peak GF	Thread Block Size
256	1728.9	16*16 (Thread: 4*4)
512	3128.3	16*16 (Thread: 4*4)
1024	3802.1	16*16 (Thread: 8*8)
2048	4506.0	16*16 (Thread: 8*8)
4096	4231.6	16*16 (Thread: 8*8)

Section (3)

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



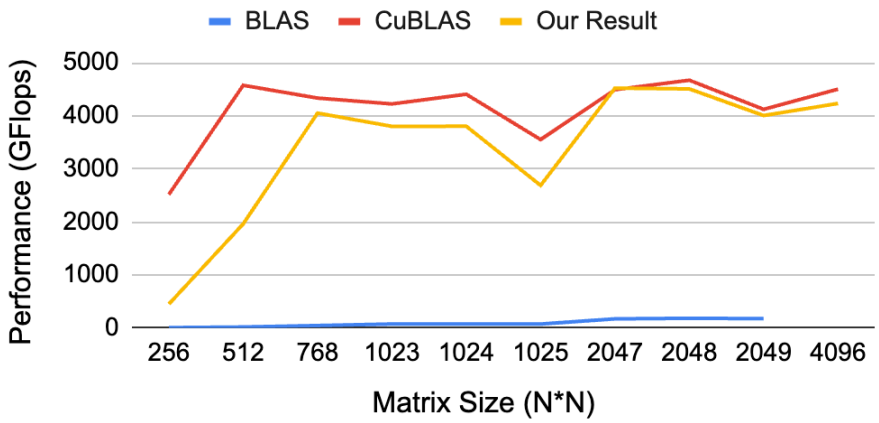
As it shows, it improved the performance by 7.96 times when N=2048. Mainly due to the fact that naive multiplication gets all the data from the global memory every time, while for my implementation, it reduces the time for getting the data from global memory but more from shared memory and register.

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).

N	BLAS (GFlops)	CuBLAS	Your Result (GFlops)
256	5.84	2515.3	449.2
512	17.4	4573.6	1960.8
768	45.3	4333.1	4049.7
1023	73.7	4222.5	3799.9
1024	73.6	4404.9	3802.1
1025	73.5	3551.0	2686.5
2047	171	4490.5	4520.5
2048	182	4669.8	4506.0
2049	175	4120.7	4003.9

4096		4501.6	4231.6



And for other values of n, please refer to q4c.

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.

It is quite similar to the Cublas value in the table. It starts with small GFlops. And then it increases. The it decreases with value of n that is not the multiple of (16*8=128=block size * scale).

It is similar in the sense that for the N value divisible by 32, it generally has more Gflops than the one that is not divisible by 32. The reason being is because of the following picture from the textbook:

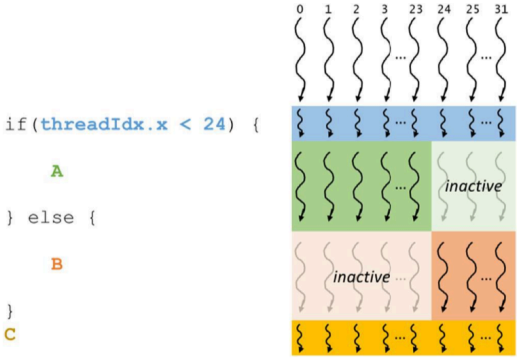


FIGURE 4.9 Example of a warp diverging at an if-else statement.

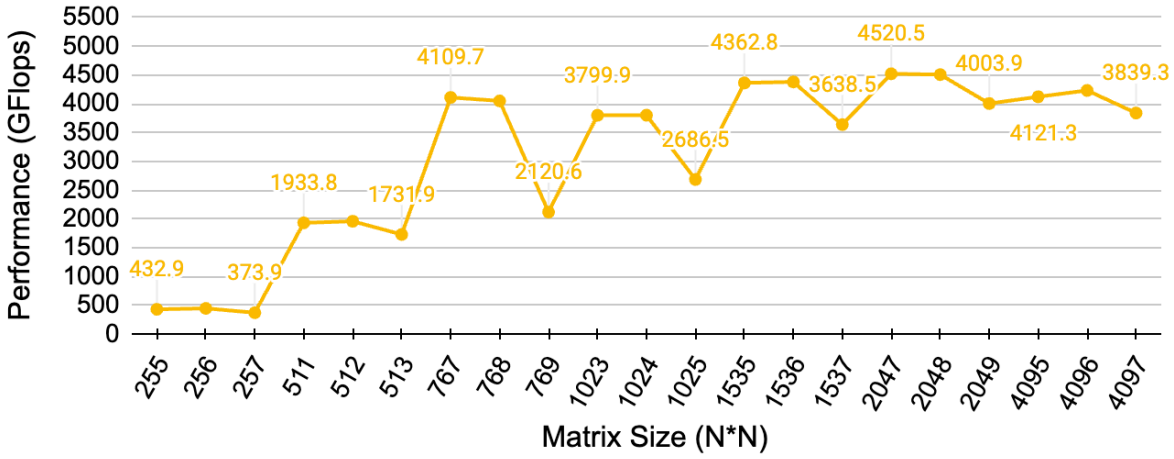
As we can see, some threads are inactive when it executes the for loop. This happens especially when we are trying to deal with the case when N is not divisible by 128.

```
if (I < N && J < N) {
    C[I * N + J] = Cij;
}
```

Since we need to allocate a ceiling of N/tilesize, then we allocate a whole tile(block) just to allocate that one thread, or that whole tile is not filled. We partially mitigate this situation by using a ternary operator so that for the size of N(especially when N % 128=0 and N % 128 = 1) , the performance is rather close.

```
#if USE_IF_STATEMENT == 0
As[(tileRowA + i) * TILEDIM_K + tileColA] = (row < N && col < N)? A[row * N + col] : 0;
#else
if (row < N && col < N) {
    As[(tileRowA + i) * TILEDIM_K + tileColA] = A[row * N + col];
}
#endif
```

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



Again refer to q4(b), we can see that initially for smaller N, the Gflops is very small. I guess it is due to the reason that the matrix size is small and since the tile size for our method is 128*128, there are a total of around 4 blocks in the size of 255,256 and particularly, for 257, we need total of 3*3 blocks though there are indeed some improvements, based on the fact that we still need to load the global memory to the shared memory. Basically, the advantage of using shared memory and register is not fully utilized.

And then we can see that for the number that is not divisible by 128, we can get lower performance. This is basically due to the thread being idle in the tile or even worse, only has one active thread in the whole tile! Let's take the values of 511,512 and 513 for example, 511 mod 128 = 127. This is not that bad since we need to allocate a ceiling of 511/128, which is 4. And for the one more block we allocated, only one thread is idle. But consider the case of 513, we allocate one more block just for that one thread since 513 % 128 = 1, which is bad, and compared to size of 512(we need 4*4 blocks), we need 5*5 blocks and at most one thread in the extra block is working. This is why the performance of 513 is much worse than the performance of 512.

Section (5)

Q5.a) Calculate the peak performance of the roofline plot and explain how you arrive at the peak.

Peak_performance = 1.5 GHz×(40 SMs×64 cores/SM×2 ops/core/cycle) = 7680 GFlops

We use the value of q in ops/word (1 word = 4 byte), therefore:

Q_theory = (7680 / 320) * 4 = 96 ops/word

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?

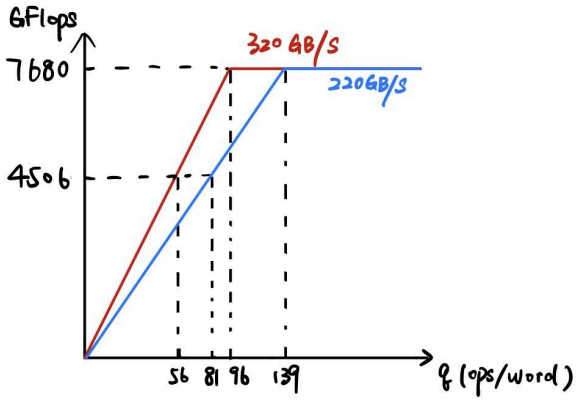
Q_actual = (7680 / 220) * 4 = 139.6 ops/word

Q_our_theory = (4506 / 320) * 4 = 56.3 ops/word

Q_our_actual = (4506 / 220) * 4 =81.9 ops/word

The q value increases with lower bandwidth, indicating that the program will take longer to reach its peak performance.

The roofline figure is as below:



Section(6) - Potential Future work

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

We find that the memory access pattern of loading A/B into As/Bs is sequential. If we have enough time, we plan to implement vectorization on this, which enables each thread to load >1 elements at one time, reducing the total number of iterations needed to load the entire As/Bs.

Section(7) - Extra credits

We did not implement the block size that is not square. However, the reason why our performance is pretty good is because inspired by the lecture content for which it only stores 4 elements, our implementation let each thread handle 8*8 c_ij grid and moreover, as the lecture slides described, we implemented the 2D tiling, achieving memory coalescing and taking advantage of the characteristics of warps in GPU, increasing the parallelism when loading the elements into shared memory, which is a huge improvement compared to 1D tiling where TILEDIM_K=1 .

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

1:Xiuxia Zhang, Guangming Tan, Shuangbai Xue, Jiajia Li, Keren Zhou, and Mingyu Chen. 2017. Understanding the GPU Microarchitecture to Achieve Bare-Metal Performance Tuning. In Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '17). Association for Computing Machinery, New York, NY, USA, 31–43. https://doi.org/10.1145/3018743.3018755

2:G. Tan, L. Li, S. Triechle, E. Phillips, Y. Bao and N. Sun, "Fast implementation of DGEMM on Fermi GPU," SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, Seattle, WA, USA, 2011, pp. 1-11. keywords: {Graphics processing unit;Registers;Bandwidth;Instruction sets;Optimization;Memory management;high performance computing;GPU;CUDA;matrix-matrix multiplication},

3:Jia, Zhe, et al. "Dissecting the NVIDIA volta GPU architecture via microbenchmarking." arXiv preprint arXiv:1804.06826 (2018).