PA2 Report

Miaoqiu Sun, Yingqi Cao

Section 1 - Development Flow

Q1.A

The kernel describes a mapping from the thread blocks to tiles in the matrix C. Each thread block processes a tile of C, where the dimensions of tile C are integer multiples of the thread block size in that dimension.

Since each thread may process multiple elements in a tile, proper offset and multipliers should be used to convert the indexing in the grid to index in the matrix. Such index conversion is initialized at the beginning of the computation to improve the code readability. Also, the 1-D array of shared memory that will contains blocks from A and B are also declared with the size of TILEDIM_M * TILEDIM_K and TILEDIM_K * TILEDIM_N at the beginning of this kernel, respectively.

There is a major loop used to iterate through a horizontal strip of A and a vertical strip of B, with each strip consisting of several rows or columns. The strips are further divided to blocks for the loop to multiply corresponding blocks from A and B and accumulate to the tile from C.

In each iteration of the outermost loop, there are two sets of nested for loops copying the corresponding block of A and B from the global memory to the shared memory, then another nested loop to compute the product of the blocks and accumulate to C. In the two loops copying A and B, there is an condition check for whether the matrix indexing we are using is out of the bound of the matrices. Zero will be assigned to the shared memory block in this case, which is the way zero padding is implemented for matrices with the size not divisible by TILEDIM.

After all the blocks from strip A and B are computed, i.e. the outermost loop finishes iterations, another loop will store tile C back to the global memory. During the computation, C is stored in the registers using `register _FTYPE_` specifiers.

In both the copying and computation process, each thread processes the element at the corresponding position in blocks. Thus, it should be strictly followed that tile dimensions are the multiples of thread block geometry, or would it result in computation error.

Q1.B Development Process

Naive Kernel with Shared Memory (FIXME)

When adding only the shared memory feature to the naive kernel, the calculation result is correct for all the matrix sizes less than 32. This limit is the smaller one of the thread block geometry. While the `TILEDIM` is large enough, it creates a grid with only one block.

With some debugging, it’s found that when the matrix dimension is larger than the thread block geometry, in the matrix copying process, all the index exceeding the block fails to copy. Thus the bug is caused by the wrong block and thread settings.

Shared Memory with 1-1 Mapping from Thread Block to Tile (e556821)

In this stage, the declaration of shared memory is added to the beginning of the kernel and the matrix elements is copied before in the loop iteration before the computation. However, when copying the matrices from the global memory and computing the product, each element in C is processed by one thread. In this case, the tile scale is 1 and the tile dimension is the same as the block dimensions.

Shared Memory with 1 to Multiple Mapping with Zero Padding (4c5f078)

This implementation makes each thread capable of processing multiple elements in a tile, which means the tile scale is not 1 and the number of elements the thread is processing is TILESCALE_M * TILESCALE_N. By indexing using the tile scale, it is convenient to change the elements processed by a thread by simply adjusting the two parameters.

Memory Coalescing (726d9b6)

The memory coalescing is realized when changing the iteration method from column by column to row by row in the process of copying elements from global memory to shared memory. Doubled performance is observed after switching the order of loops in the previous implementation.

Parameter Tuning with NSight Profiling

Tuning the parameters with NSight Compute and manually analysis the shared memory size.

Q1.C

Different Copying Scheme (6b9d4f5)

For each thread, rather than copying the corresponding element at the same position in each block, it copies a consecutive rectangular block having the size TILESCALE_M * TILEDIM_N. Implemented for both A and B, but only this variation of A is used when benchmarking for comparing how it can improve the performance. However, this actually reduces the performance. And it's probably because of the specific auto-optimization techniques used by the runtime environment or the hardware scheduler. When a row of threads in a block is copied from consecutive memory addresses, they will be treated as one single operation similar to SIMD. However, when the copying of a row in the block is scheduled in a chronologically sequential order as in the for loop, it's harder for the hardware or the runtime to optimize than parallel threads.

Padding Only at the End (0efb2df)

For cases that the matrix size n is not divisible by the tile dimensions, zero padding was done in the nested loop of copying A and B to the shared memory. There is a branch evaluating whether the current row index and column index of the matrix is larger than the matrix size. If it is, then 0 will be assigned the corresponding location in shared memory, else copy the element from the matrix. It is known that conditional statements make it harder to execute in full parallelism, removing the conditional statement in the frequently executed loop iteration would probably reduce cycles used in condition evaluation. Since the zero padding only happens for the last block in a strip, the outermost loop is changed to iterate through the blocks except the last one.

This makes it possible to remove all the if statements in the major for loop. An extra copying and computing code with zero paddings are appended to this for loop. Rather than doing the full zero padding which also considers the last several rows not able to constitute a full strip (requires the conditions to filter out the last row of blocks in the grid), removing conditions in the matrix copying in the major loop should theoretically already reduced most of the condition evaluation in the entire process, thus is used to verify the assumption with proper matrix size before implement the full zero padding.

However, this technique just barely increases the performance by less than 200 Gflops/s for the size of 1024. The reason for this attempt to fails is because the techniques used by the hardware scheduler (such as dynamic branch prediction in CPU #!!! NOT THE CASE FIX ME LATER) already "optimizes" this branch at low level.

Maximum Thread Used in Block

As a part of parameter tuning, to increase the level or parallelism, the performance of using maximum number of threads in a block is tested. The maximum number of threads per block is 1024 that is limited by the architecture. When the tiles are squares, the block size geometry to fully exploit available thread was 32 by 32. However, this actually reduces the performance due to the several `__synthreads()` between copying A and B and computing C. The synchronize thread function waits until the slowest threads in the block finish the execution, which means all the other threads have to wait. Thus, an unnecessarily large amount of threads in a block makes it more likely for all the threads to wait for a thread using a long time for whatever reason, which is a huge waste of resources. However, to fully utilize all the available threads in a block, even odd memory method could be considered (details see section 6).

Section 2 - Result

Q2.A

The relationship between the block size and tile size is described by the tile scale, fulfilling the following set of equations:

$$TILEDIM\_M = TILESCALE\_M * by$$
$$TILEDIM\_N = TILESCALE\_N * bx$$

However, the TILESIZE_K is not associated with any block geometry parameters directly, though it may reduce the performance in the case it is not divisible by bx and by at the same time when trying to map threads to multiple elements when copying the tiles to the shared memory.

We tried three different thread block sizes with each map to the same tile sizes (TILEDIM_M=128, TILEDIM_N=64) but different tile scales as the following:

1) bx=16, by=64; TILESCALE_M=8, TILESCALE_N = 1;
2) bx=16, by=16; TILESCALE_M=8, TILESCALE_N=4;
3) bx=32, by=32; TILESCALE_M=4, TILESCALE_N = 2.

We plotted the performance of our code for these different thread block sizes for matrix sizes N=256, 512, 1024, 2048 and 4096 as shown in Figure 1.

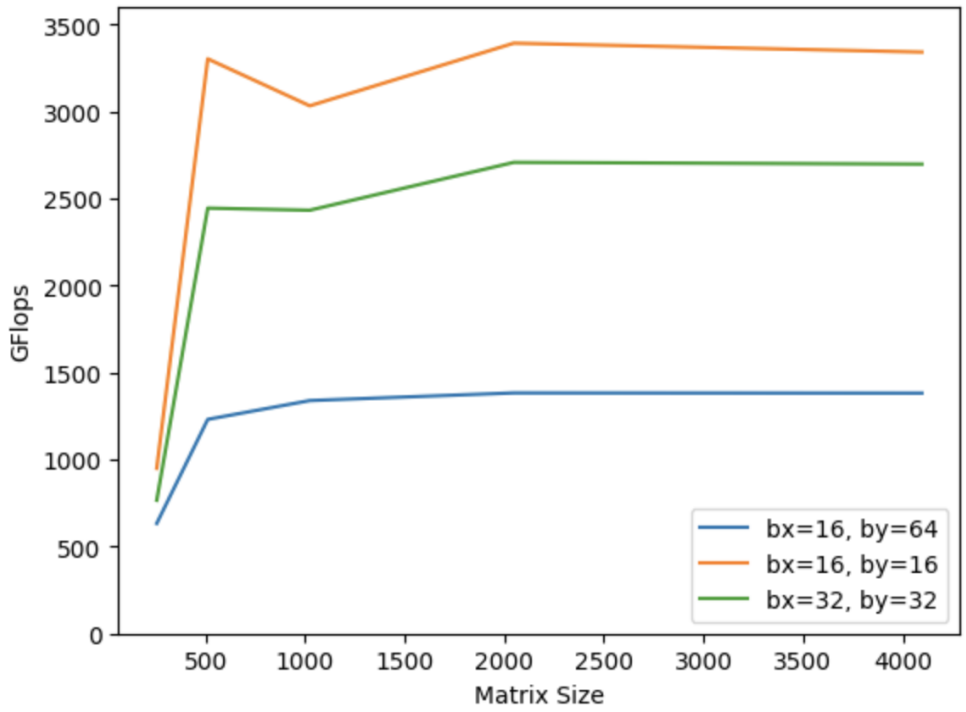


Figure 1.

Q2.B

Our optimal block size is chosen at bx=by=16 (TILESCALE_M=8, TILESCALE_N=4; TILEDIM_M=128, TILEDIM_N=64), which gives us the highest performance for each matrix size (N = 256, 512, 1024, 2048, 4096) among all others. One potential reason that this block size outperformed the others is that larger matrix sizes such as 32 * 32 or 16 * 64 will cause the occupancy to be too high such that we would not be able to fully utilize the SM resources for parallelism. 16 * 16, on the other hand, will have a lower but more reasonable occupancy, and therefore will fully utilize SM resources and do more parallel work.

Q2.C

N	Peak GF	Thread Block Size
256	950.389861	bx=16, by=16
512	3303.275220	bx=16, by=16
1024	3033.771437	bx=16, by=16
2048	3392.939265	bx=16, by=16
4096	3342.746455	bx=16, by=16

Section 3

Q3.A

N	Naive (GFlops)	Our Best Results (GFlops)
256	359.6	950.389861
512	334.5	3340.472536
1024	336.6	3071.396699
2048	376.1	3449.444116

Section 4 - Analysis

Q4.A

N	**BLAS (GFlops)**	**CuBLAS (GFlops)**	**Our Result (GFlops)**
256	5.84	2515.3	907.553877
384			2271.901143
481			2650.571515
512	17.4	4573.6	3340.472536
640			2621.087445
768	45.3	4333.1	3282.626092
1023	73.7	4222.5	2964.357549
1024	73.6	4404.9	3071.396699
1025	73.5	3551.0	2756.209318
1280			3449.129907
1471			3223.665607
1536			3236.554210
1792			3444.201906

1818			3067.851570
2000			3239.247349
2012			3176.256827
2047	171	4490.5	3368.803022
2048	182	4669.8	3449.444116
2049	175	4120.7	3002.215752
4096		4501.6	3389.128306

We plotted the performance for BLAS and our results (Figure 2.) for N within range (256 - 2049) as the following:

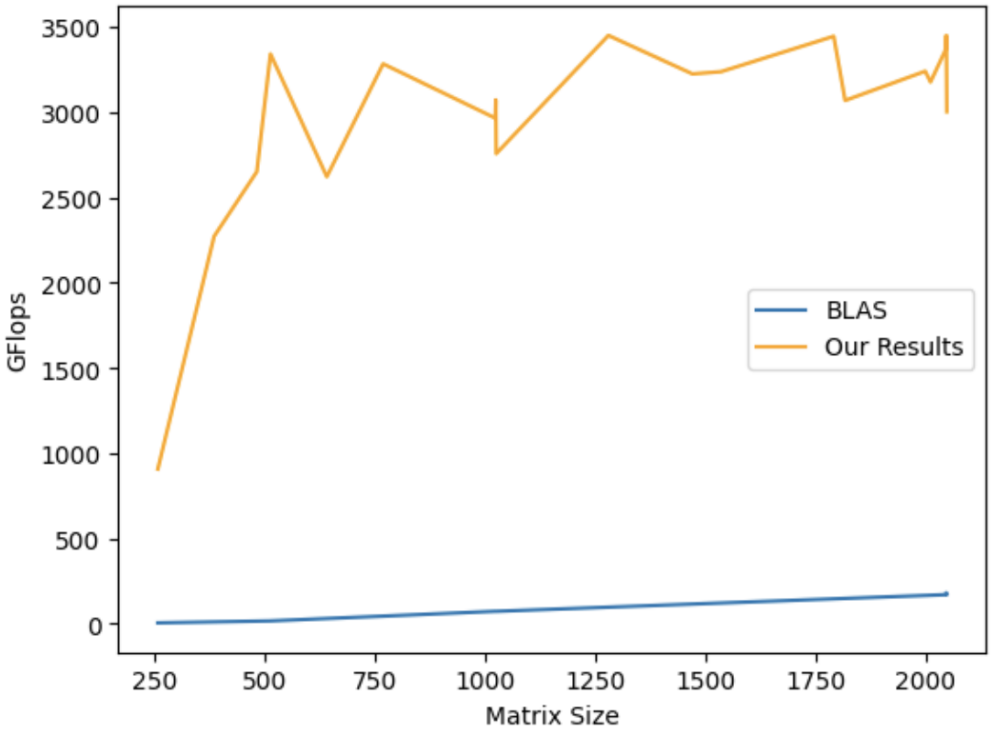


Figure 2.

Q4.B

The shape of our curve is quite different from the BLAS values. The BLAS's curve is nearly linear and stays increasing without any drops in performance as N gets larger and larger. However, our curve has several dips in performance at N=640, 1023, 1025, 1471, 1818, 2012, and 2049, where N is not divisible by the thread block size 16. Why this might be in this case is because of the zero padding mechanism we implemented for these non-divisible matrix sizes. The process of accessing and dealing with additional rows or columns of zeros could explain why these drops happened.

Q4.C

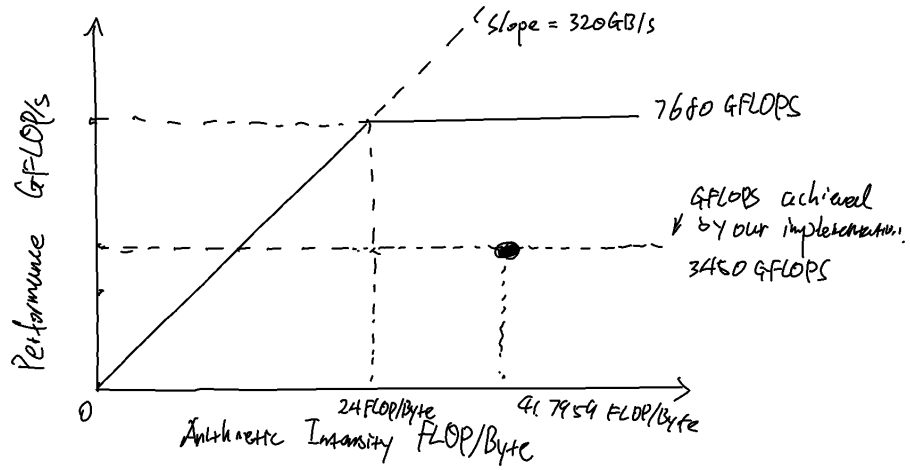
In addition to the unusual dips mentioned in Q4.B at non-divisible thread block sizes due to additional efforts of accessing zero matrix that lead to performance drop, we also noticed several peaks of overall performance at N=1280, 1792, and 2048. These are all relatively larger matrix sizes among all N, and this could be explained by the smaller amount of performance overhead of large matrices compared to that of small matrices. It is also noteworthy that these N at peaks are all divisible by thread block size, which again verified our theories on the impacts of zero padding in regards to performance drop.

Section 5

Q5.A

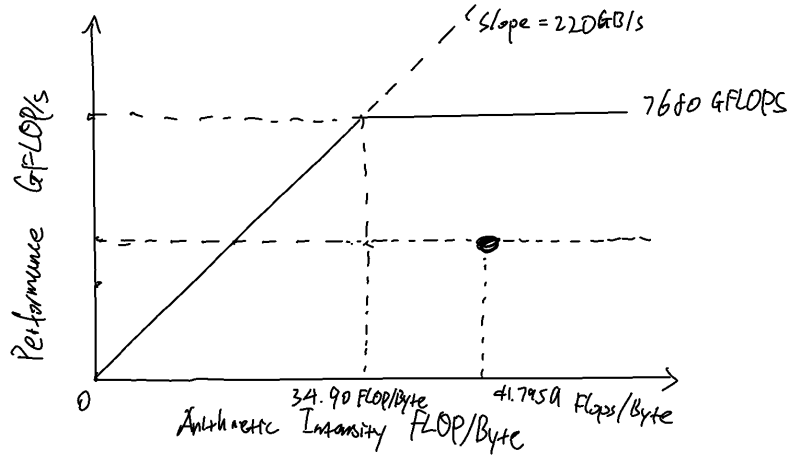
If an FMAD is considered as two floating point operations, then the theoretical maximum performance achievable by the Turing T4 GPU is 40SMs * 64Cores * 2Ops/Cycle * 1.5 * 10^9 Cycles/s = 7680 GFLOPS, which determines the horizontal limit part of the roofline model. Using this maximum can we compute the intersection of it and the slope segment. The bandwidth specified is 320GB/s. The data have to be fed into the cores to achieve this intensity can be determined by (7680GFLOPS/s) / 320GB/s = 24 FLOP/Byte, which corresponds to the position of the ridge point on the horizontal axes.
Our mode has a performance of 3450GFLOPS/s, which determines the vertical position.

The slop segment represents a memory bound, which means at the current computation intensity, the highest perforamcen achievable depends on how many data is transferred. Thus, to arrive the peak performance when the computation intensity is less than 24FLOP/Byte, feed more data to the SM when computing can increase the performance until the peak. At the flat segment is the performance bound, which means computation is not fast enough to consume the amount of data fed in. However, to achieve the peak performance in this segment can we also increase the data feed in the SM.



Q5.B

Rough estimation of the computational intensity in our program.
The point representing the performance of our implementation doesn't meet the roof since it is not optimized enough to be limited by the computation or bandwidth bottleneck.
The new q value on the roofline is (7680GFLOPS/s) / 220GB/s = 34.9 FLOP/Byte.
Assume each tile has the same computation intensity. Then roughly q = (128*64*2048)/(2048*(64+128)+128*64) = 41.7959, approximately 10.449 GLOPS/word.



Section 6 - Potential Future Work

Even Odd Memory Scheme

As mentioned in the previous section, using the maximum threads available doesn't necessarily result in a higher performance because of what `__syncthread()` does. However, using our current block size of 16 by 16 keeps only ¼ of the available threads busy at a time. Thus the idea of an even-odd memory scheme can potentially increase performance.
The scheme is to divide the shared memory into two sections, with even index indexing the even section and odd index indexing the odd section. The kernel first copies a block from the global memory to the odd section of the shared memory. Then it computes the matrix multiplication. As some threads are busy computing the product, the other available threads will be copying the next block into the even section of the shared memory. Those threads then multiply the just copied blocks in the even section. At the same time, the threads responsible for the odd section would finish the computation, and begin to copy the next block into the odd memory again.

Without this scheme, the bus is resting while computing, and also no computing happens while the data is moved back and forth. This method uses more threads without extensive waiting time resulting from synching those threads and also keeps the memory bus and computational cores always busy. As an analog to CPU pipelining, this method of keeping all the resources busy rather than waiting for the entire execution to be done would very likely boost the throughput. A bold intuition suggests a doubled theoretical performance. More realistically, the new runtime for a tile would be a number of blocks times the longer of copying and computing.

Task Divide for Threads

Different from the previous potential work utilizing pipelining, this emphasizes further dividing the current program into parallelizable parts.

As mentioned in the "Even Odd Memory Scheme" section, the optimized performance of keeping only one set of threads busy at a time is the result of using only ¼ of the available threads. This encourages identifying parts in the program that could be done in parallel, which is the part that has no data dependency. An easy-to-identify one is moving data to the shared memory. In the current implementation, copying A and B is sequential, which means B won't be copied until the copying of A is finished. However, since the copying of the blocks is independent, it can be done in parallel. This means using twice the threads, though only in the copying process.

However, the three steps, copy, compute, and store back are data dependencies that exist, it's hard to put the other steps in parallel.

Section 7 - Extra Credits

Section 8 - Reference

No reference is used in this PA.