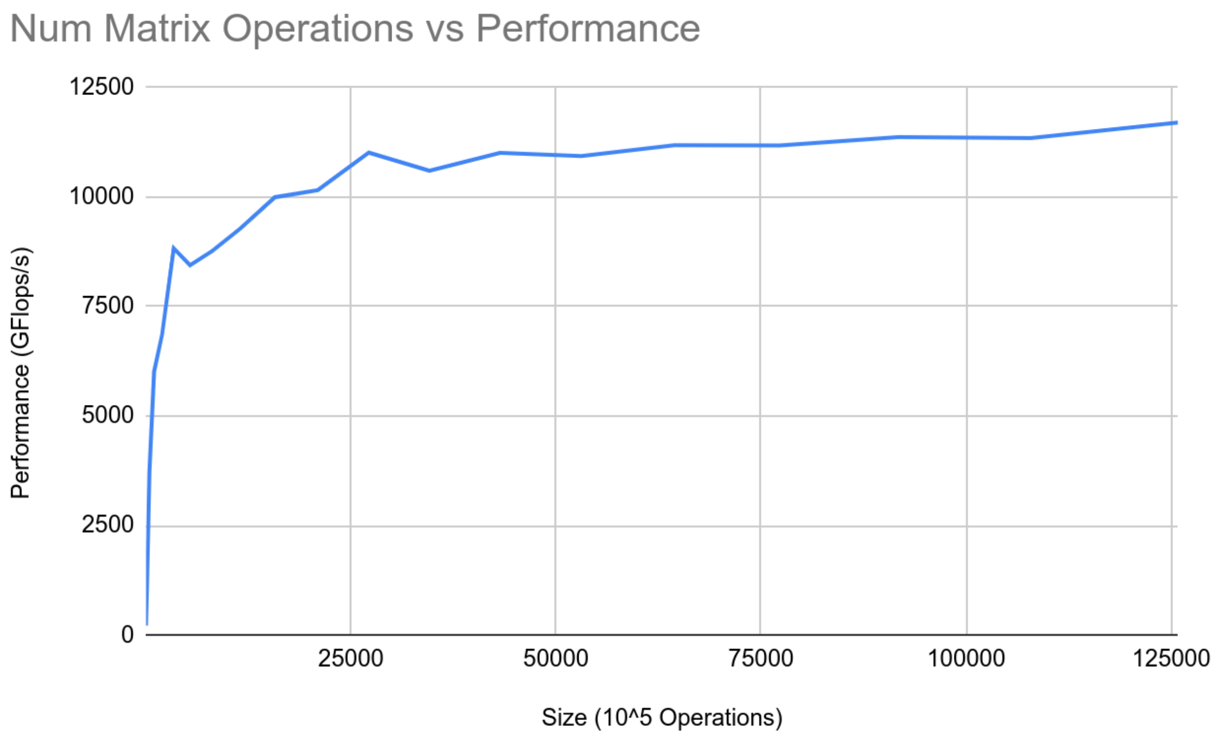
Grayson Byrd
CPSC 8200 - Parallel Architecture
Project 2

Part 1

Below you will see a graph of the Performance in GFLOPs vs the number of operations for matrix multiplications using CUBLAS. Here you can see that the performance gain is very steep at the beginning of the graph when the matrix is small and then gradually plateaus to a maximum value of around 11500 GFLOPs. This is expected, as initially, when there is not much computation to be done, the memory accesses are the bottleneck to performance. However, as the computational burden increases, we can parallelize the application by allowing computations to take place while we are waiting for memory operations to complete. In this way we can be more efficient with our time. We expect to see a plateau at some point because we will eventually reach limits to the GPU. For example, we will eventually reach a memory wall where caching more data will no longer be possible because we will run out of hardware space. Similarly, we will run into compute limitations as we reach the maximum number of threads that can be running at one time on the GPU.



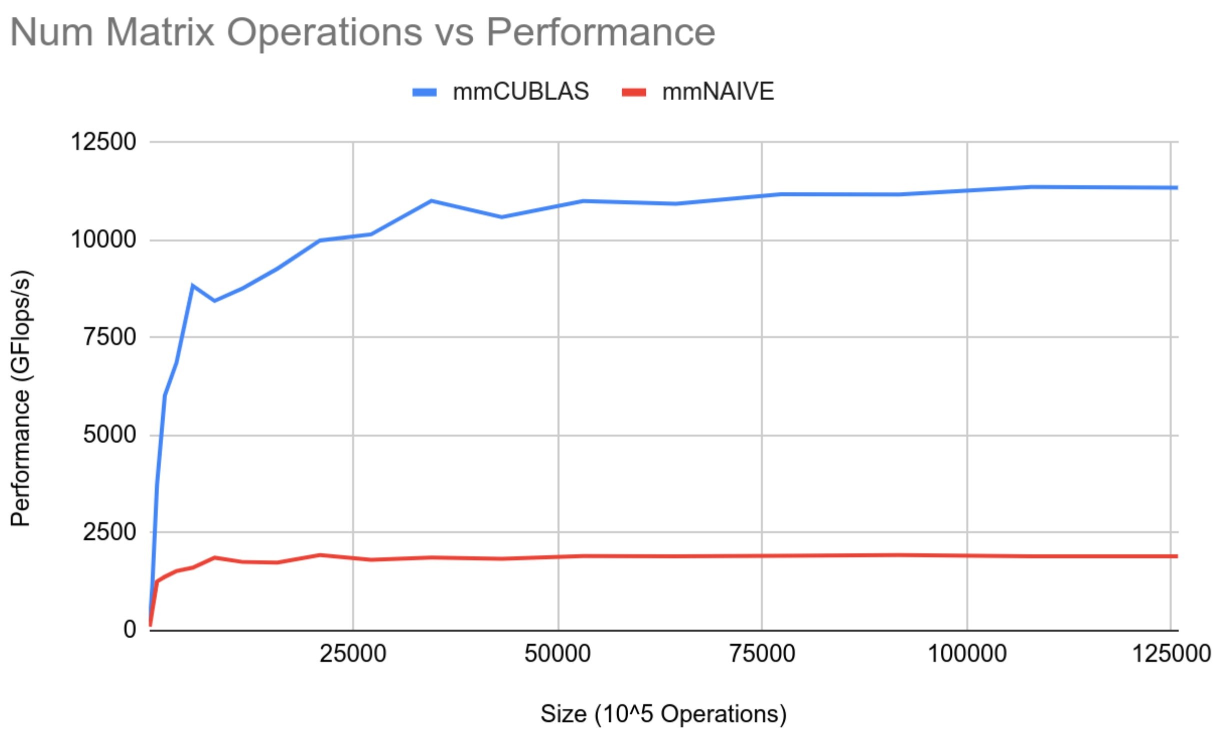
Part 2

For this part of the assignment, I decided to include my naive implementation as the first baseline portion of my part 3 optimization code. For this reason, I do not have a mmNAVIE.cu file, but you can run my naive implementation by running my optimization code with the following arguments:

```
./mmOPT –sizemult=3 0
```

Where the 0 represents the identifier of the naive kernel.

Below you will find the comparison between the CUBLAS implementation of matrix multiplication and my own naive implementation. As you can see, my naive implementation is much slower than CUBLAS. There are a wide variety of reasons for why this might be. Firstly, the naive implementation is not using threads in the most efficient way. It is simply having each thread compute one index of the final matrix, C, by loading in an entire row of A and an entire column of B and multiplying them together. This is not an optimized way to implement matrix multiplication and does not leverage our knowledge of how the GPUs work to push the highest performance out of this operation. To put it simply, there are both memory and algorithmic constraints in the naive implementation. These constraints and limitations will be explored and optimized in part 3.



Part 3

For part 3 of this assignment, I have implemented 4 different optimization methods to improve the performance of my custom CUDA matrix multiplication implementation. I will discuss each of them in depth below.

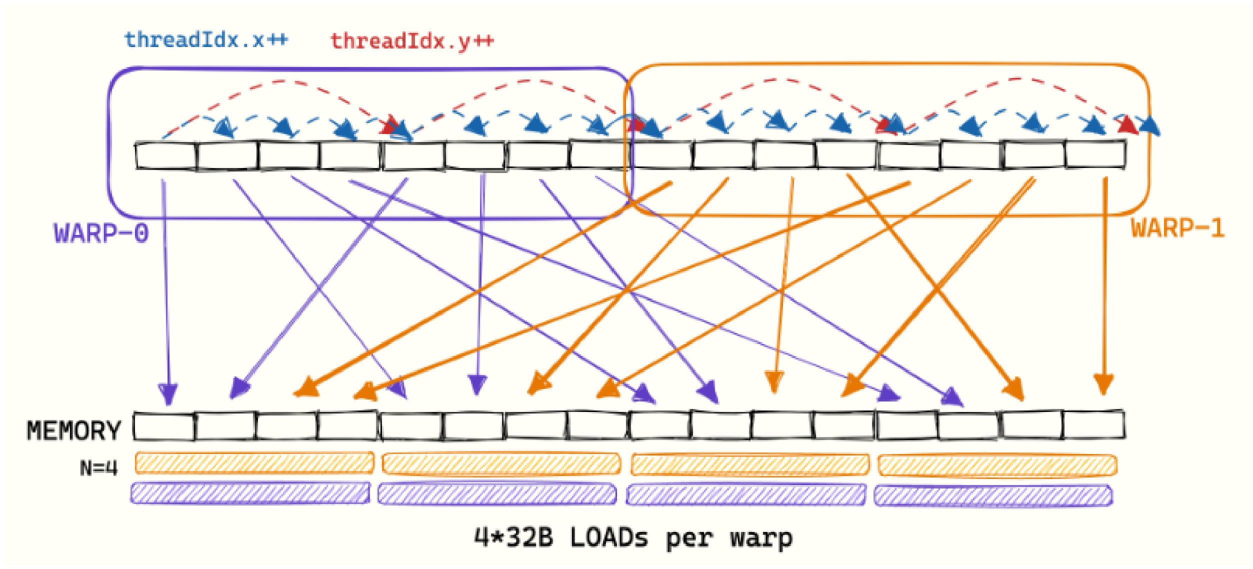
Method 1: Global Memory Coalescing

The first optimization method has to do with optimizing the way we access the global memory. In the naive implementation, each thread will access global memory and unnecessary amount of times. Global memory coalescing will help with this.

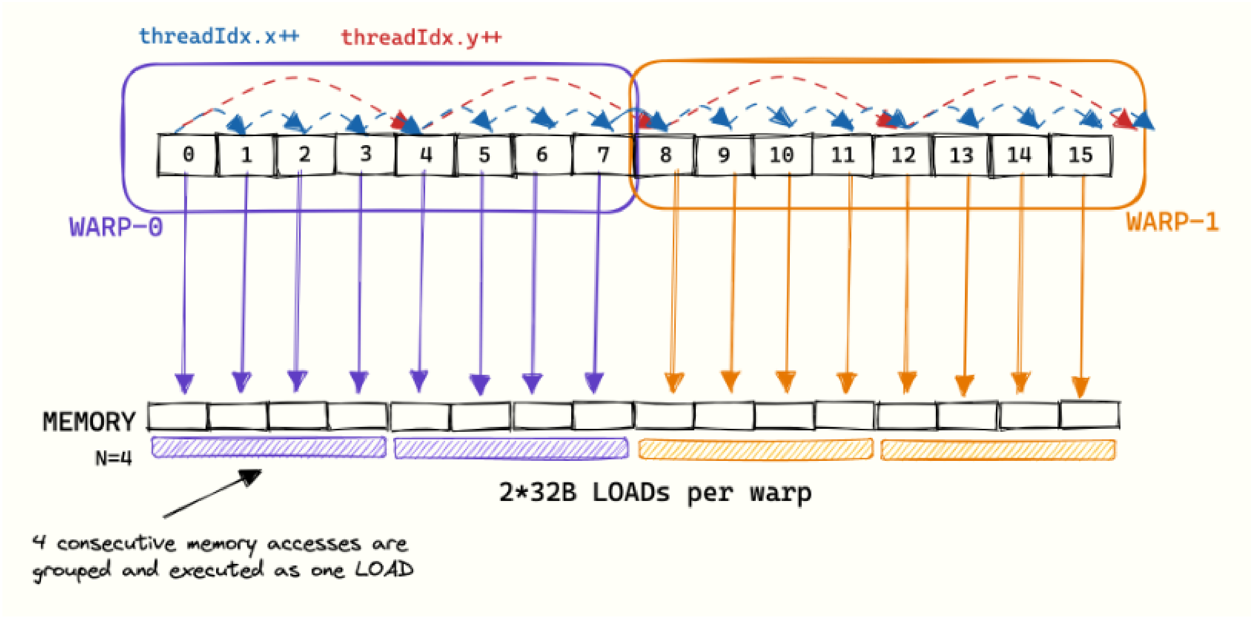
To explain this, we need to understand two things:

1. CUDA organized sequential threads inside a block into ‘warps’ of 32 threads.
2. Sequential memory accesses by threads that are part of the same warp can be grouped together and executed as one.

In our naive implementation, coalescing cannot occur because sequential threads are not accessing sequential data. Below is a diagram of how the memory is accessed by sequential threads in our naive implementation:

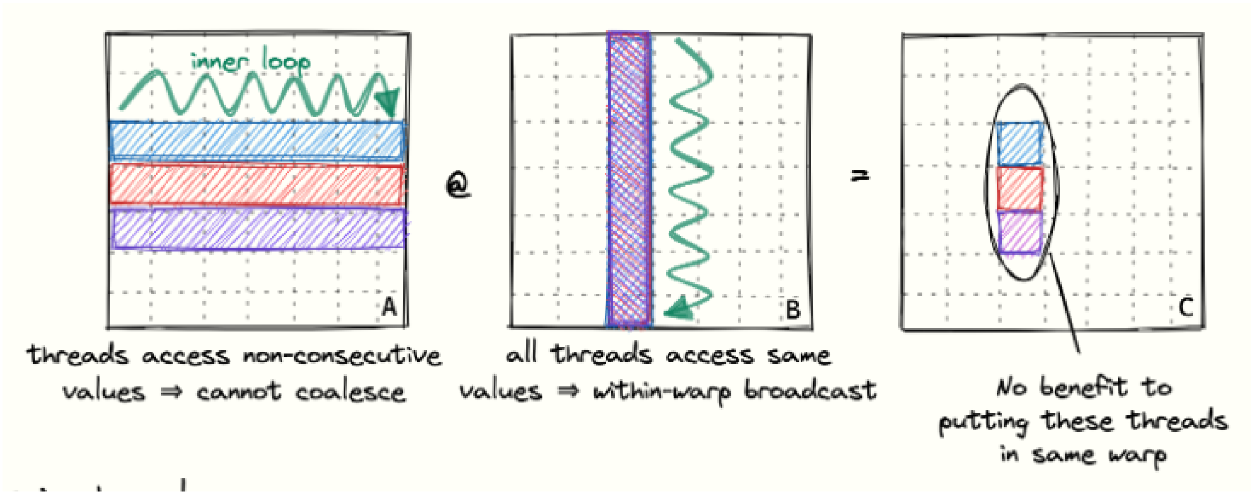


Notice how the blocks of memory on the bottom are being accessed by threads from multiple warps? This means that we have to access that memory multiple times (once for each warp accessing it). It is inefficient to access an entire chunk of memory for just one byte of information from that memory. If we instead change the order of our threads, we can then arrange them into warps that will access memory like so:

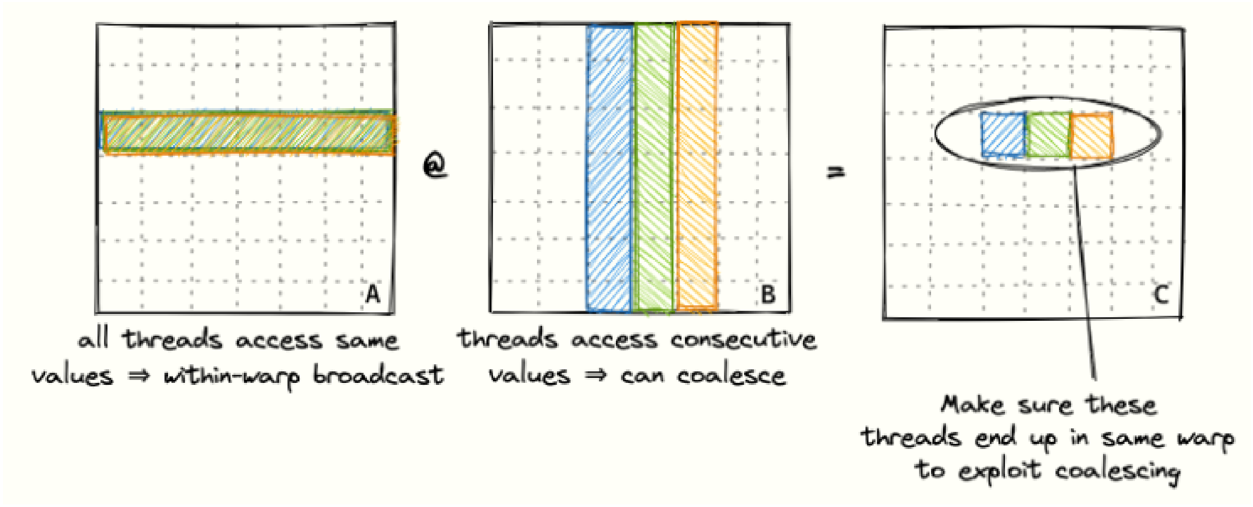


In this image, sequential threads will need to access sequential data, therefore we will only need to go to the global memory once per warp. This will increase the efficiency of our algorithm with respect to memory access.

In the naive implementation, threads are grouped together by rows. This is not conducive to coalescing as described by the below graphic:


inner loop
A
@
B
=
C
threads access non-consecutive values ⇒ cannot coalesce
all threads access same values ⇒ within-warp broadcast
No benefit to putting these threads in same warp

If we instead change the implementation of how our threads are iterated across sequentially, we can group them by column which will allow us to coalesce the memory access.


A
@
B
=
C
all threads access same values ⇒ within-warp broadcast
threads access consecutive values ⇒ can coalesce
Make sure these threads end up in same warp to exploit coalescing

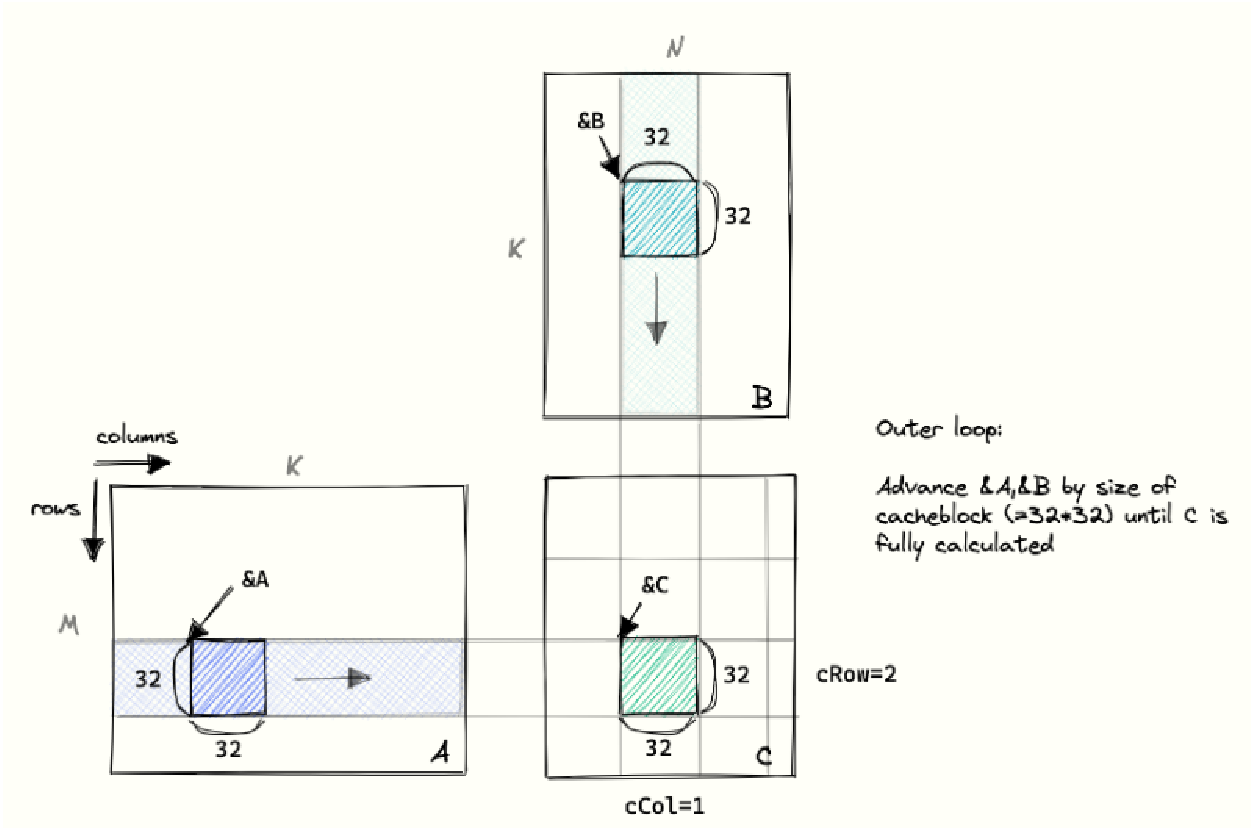
This method was used in the coalescing.cuh kernel and we achieved a speedup of 1.13x with respect to the naive kernel.

Method 2: Basic Tiling

Within CUDA, there are different hierarchies of memory. The slowest and largest memory is the Global Memory. We do not want to access this any more than we have to. Despite the fact that we coalesced the threads in our previous optimization method, we are still accessing global memory more than we need to. This implementation will leverage another memory type within the CUDA memory hierarchy: shared memory.

Shared memory is smaller than global memory, but is much faster to access. For this method, we will be breaking the matrix up into blocks and having each thread access global memory and save the global memory information into shared memory which can be accessed by every thread in the block. Each thread in a block will load one item from global memory into shared memory, so we can parallelize the memory access operation.

Then each thread will perform matrix multiplication using and entire row and column of the tile of shared memory. The tile will be swept column wise across the A matrix and row wise across the B matrix, repeating the above algorithm and incrementing the resulting idx in C for each tile matrix multiplication. By leveraging memory coalescing in addition to this method, we can see an even larger speedup as shared memory is much faster than global memory. A visualization of the basic tiling algorithm can be seen below:

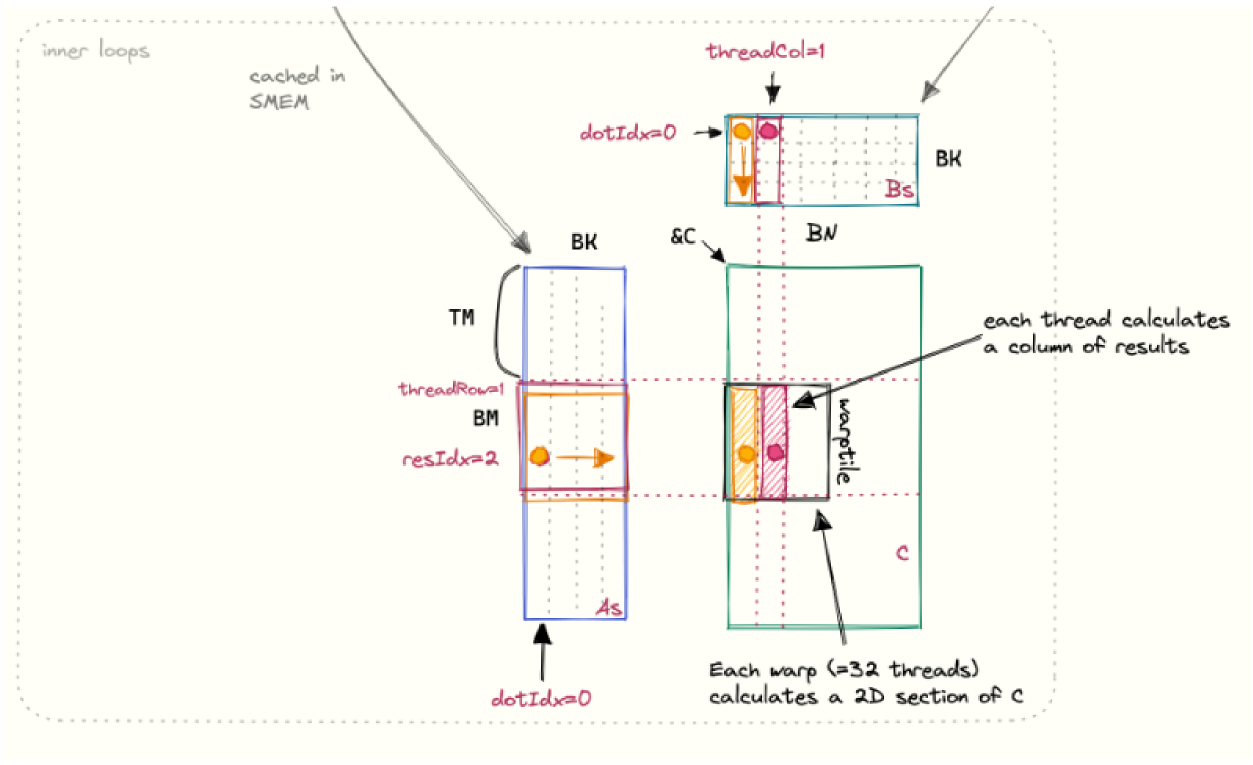

N
&B
32
32
K
B
Outer loop:
Advance &A,&B by size of cacheblock (=32*32) until C is fully calculated
columns
K
rows
&A
M
32
32
A
&C
32
cRow=2
32
C
cCol=1

Basic tiling paired with global memory coalescing gave us a speedup of 2x over our naive implementation.

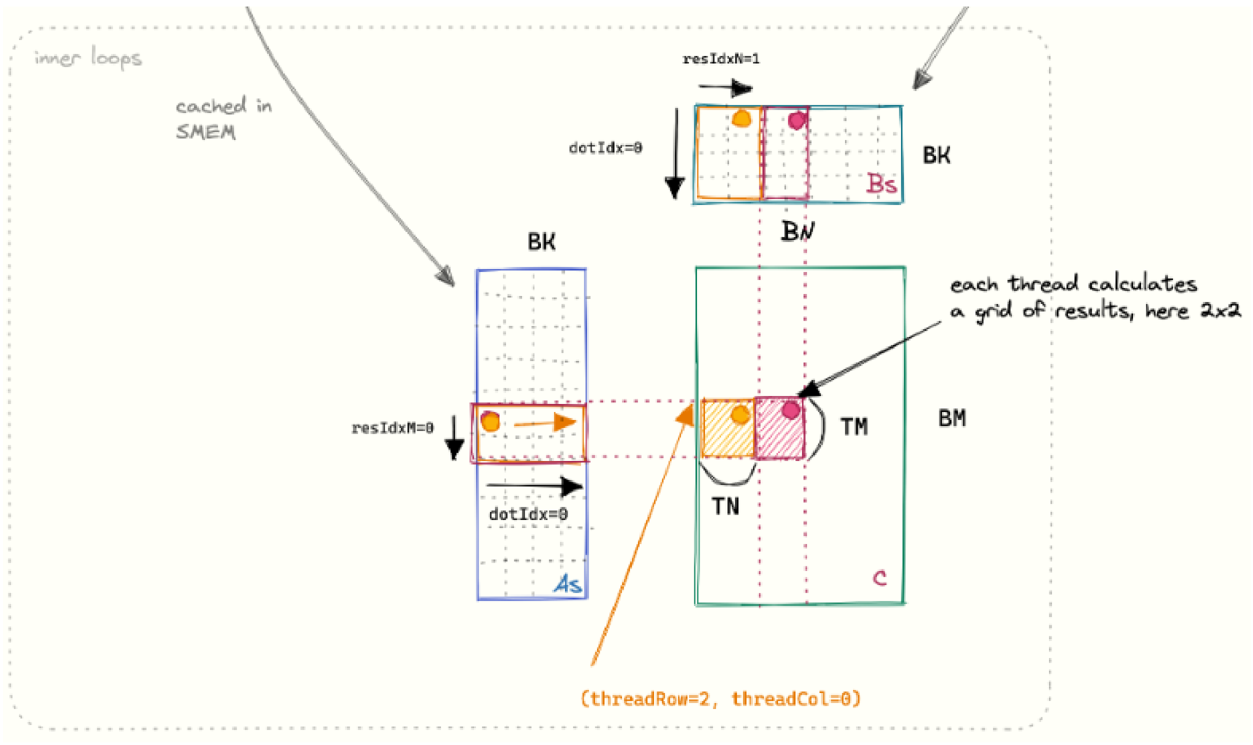
Method 3 and 4: 1D and 2D Blocktiling

The smallest and fastest memory in the CUDA memory hierarchy are the registers. These are blazingly fast, and luckily we can leverage them to further improve our optimization.

For this method, we will still load tiles of data into shared memory like before, but this time we will have each thread compute multiple values. By having each thread compute multiple values, we can cache the information into registers and allow for even faster memory access. I will first explain this algorithm in 1D. For this algorithm, after loading all of the global memory into shared memory, each thread will cache the column value. The thread will then use a for loop to iterate across rows to increment values in the C matrix. The tiles will then move across A and B as in the previous method, allowing each thread to compute one column of each tile. In this way, we eliminate the need for many shared memory accesses by caching in the register. Below is an illustration of the 1D approach to this algorithm.



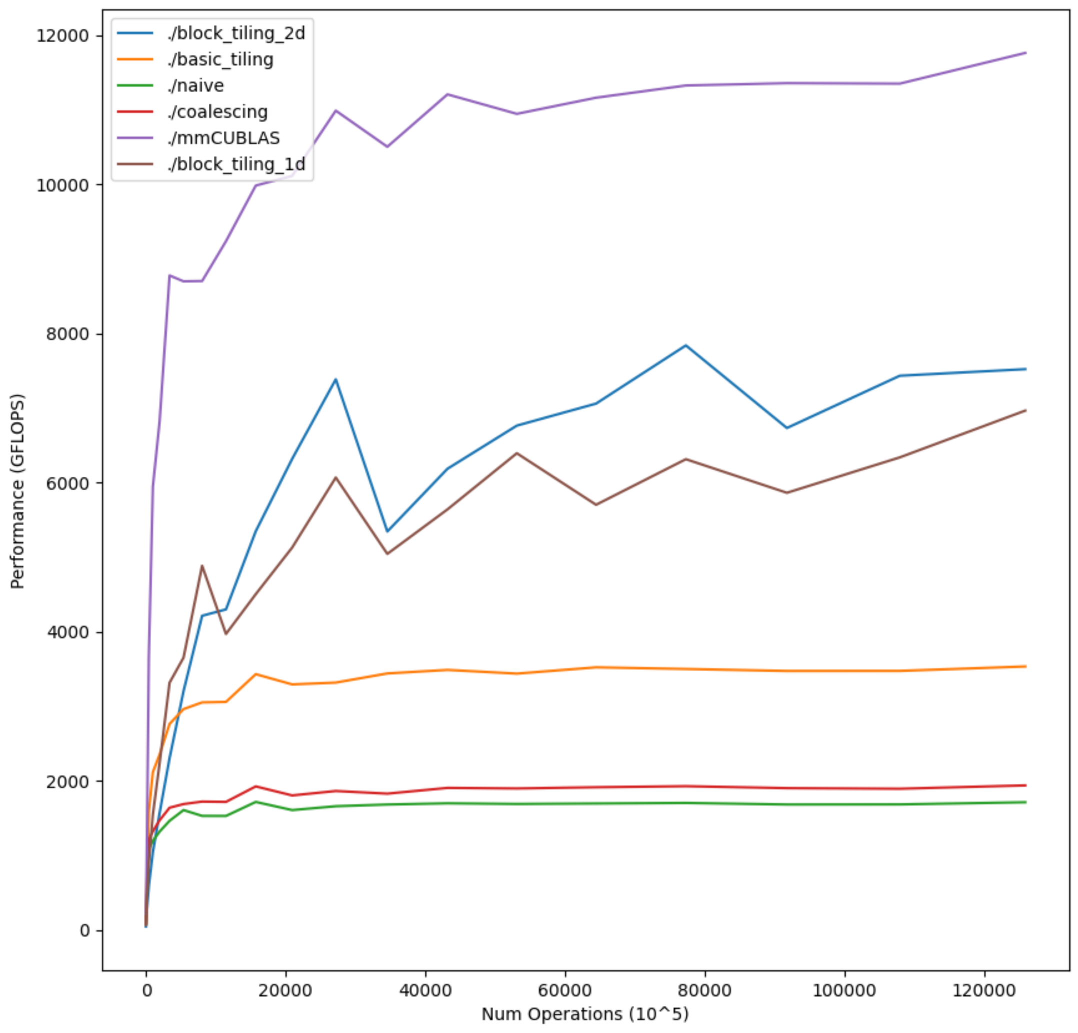
The 2D implementation of this algorithm is very similar, but is even faster. In the 2D algorithm we will load everything into shared memory in the same way, but each thread will now calculate multiple elements in 2D by caching multiple elements from a column and a row into registers. Then, two for loops will have the thread increment the values in the C matrix. The tiles will once again move across A and B and this algorithm will be repeated until all of the values in C are correct. The visualization of the 2D version of this algorithm can be shown below:


inner loops
cached in SMEM
resIdxN=1
dotIdx=0
BK
Bs
BN
BK
each thread calculates a grid of results, here 2x2
resIdxM=0
TM
BM
dotIdx=0
TN
As
C
(threadRow=2, threadCol=0)

The 1D and 2D implementation of block tiling improved performance by 4x and 4.4x respectively over our naive implementation.

Conclusion

To conclude, we greatly increased the performance of our naive implementation by leveraging many different tricks to improve the memory access speed and parallelization of our algorithm. Overall we achieved a speedup of 4.4x our naive implementation and achieved a performance that was approximately 65% of the CUBLAS implementation. Below you can see the graph of the performance of each algorithm.



References:

The bulk of this code and the resources used in this report were adapted from this repository and this blog:

https://siboehm.com/articles/22/CUDA-MMM

https://github.com/siboehm/SGEMM_CUDA