Matrix Multiplication using CUDA

1 Development Flow

1.1 Algorithm

Our program tries to optimize matrix multiplication on GPUs by splitting up the matrices into smaller tiles which are operated on by GPU thread blocks. Each thread block computes one tile of the output matrix C using a matrix inner product by iterating through the tiles of input matrices A and B, and accumulating results along the way.

Naively computing the inner product results in multiple accesses to GPU slower global memory as compared to shared memory, which is attached to every streaming multiprocessor (SM). To avoid this problem, our algorithm first loads the matrix tiles from global memory to shared memory and then operates on the data available in shared memory. However, care must be taken to ensure that global memory accesses are coalesced. The Kepler K80's memory bus fetches 128 bytes of data from global memory per transaction. This means that optimally utilizing the memory bus requires minimizing the number of transactions and also making sure to use all 128 bytes of data fetched. For instance, our algorithm does this by ensuring that consecutive threads in a warp access consecutive memory addresses.

Moreover, our algorithm attempts to improve instruction level parallelism by requiring each thread to perform more work by loading multiple elements to shared memory and also by computing multiple entries for the output C matrix. To preserve coalesced accesses to global memory, our algorithm is designed such that we split the matrix tiles into smaller sub-tiles such that each thread loads an element from each of these sub-tiles (essentially, the thread block size acts as a stride for each thread to load elements). In this way, threads with consecutive thread IDs would access sequential global memory addresses even though each thread loads multiple elements from global memory to shared memory.

Finally, we leverage the fact that Kepler K80's shared memory is divided into 32 different memory banks such that we can make 32 parallel requests to shared memory. Optimizing our algorithm to make use of multiple banks requires that we avoid bank conflicts i.e. every thread in a warp must load data from a different shared memory bank. To achieve this, we once again make use of striding across matrix sub-tiles so that threads with consecutive IDs access different shared memory banks.

The pseudocode for our algorithm is shown in Algorithm 1.

1.2 Development Process

1. The first step was to take the naive matrix multiplication algorithm and modify it to use shared memory on each SM. Doing this gave us a modest performance boost from 100 GFlops to around 150 GFlops.

2. The next thing we tried was to increase ILP by assigning more work per thread such that each thread was responsible for loading 16 elements into shared memory.

3. Finally, we tried to implement an algorithm based on Nvidia's CUTLASS [1] by using register tiling and computing matrix outer products. Unfortunately, the above two steps only gave us a performance boost to around 250 GFlops. Despite several attempts at tuning this implementation, we weren't able to improve.

4. We then took a step back and abandoned the Cutlass approach. Instead, we reverted to the ILP stage and began trying to allow our code to handle variable matrix tile size parameters.

Algorithm 1 Matrix Multiplication Pseudocode

```
function MatMulGPU(N, C, A, B, BLOCKTILE_M, BLOCKTILE_N, BLOCKTILE_K)
    sharedA ← ALLOCATE_MEMORY()
    sharedB ← ALLOCATE_MEMORY()
    for kk = 0 to N do
        for i = 0 to BLOCKTILE_M do
            for j = 0 to BLOCKTILE_K do
                load sharedA[threadIdx.y + i][threadIdx.x + j]
                j ← j + BLOCKDIM_X
            end for
            i ← i + BLOCKDIM_Y
        end for
        for i = 0 to BLOCKTILE_K do
            for j = 0 to BLOCKTILE_N do
                load sharedB[threadIdx.y + i][threadIdx.x + j]
                j ← j + BLOCKDIM_X
            end for
            i ← i + BLOCKDIM_Y
        end for
        __syncthreads()
        for k = 0 to BLOCKTILE_K do
            for i = 0 to (BLOCKTILE_M/BLOCKDIM_Y) do
                for j = 0 to (BLOCKTILE_N/BLOCKDIM_X) do
                    Accumulate product of sharedA and shared B into C_ij[i][j]
                    j ← j + 1
                end for
                i ← i + 1
            end for
            k ← k + 1
        end for
        __syncthreads()
        for i = 0 to (BLOCKTILE_M/BLOCKDIM_Y) do
            for j = 0 to (BLOCKTILE_N/BLOCKDIM_X) do
                Store accumulated product C_ij[i][j] into C
                j ← j + 1
            end for
            i ← i + 1
        end for
        kk ← kk + BLOCKTILE_K
    end for
end function
```

5. Finally, we used a python script to find out optimal configurations for matrix block sizes as well as thread block sizes.

1.3 Ideas That Did Not Work

Broadcast with Shuffling Instructions We tried to leverage the shuffling instructions which exchanges data between threads within a warp, to reduce number of shared memory accesses. As we are iterating over the k dimension, doing the outer product, all threads are accessing the same column of the A matrix and the same row of the B matrix. So we can first load the value from a thread, then use the `__shfl_sync()` instruction to share it among other threads. However, this introduces a conditional branch identifying the loading thread and it hurt the performance. Generally it got 65% of the efficiency of the program without this shuffle.

Reduction with Shuffling Instructions Another way we came up to use the shuffling instructions is reduction along the k dimension. For example, we can double the number of threads in a block, so that each thread still has the same kernel size but aggregate the outer product k/2 times. After computing the results, we use a shuffling instruction to add the results from the second half to the first half, then store that back to the C matrix. This also led to the degradation of performance, as later through parameter tuning we found out that actually we need each thread to do more work, so this idea worked in the wrong direction. In our measurements, programs implemented this idea obtained 80% of the performance of the ones without.

8B Memory Bank Size As the elements in the matrices are 8 byte each, it seems that increasing the memory bank size from 4B to 8B would help the performance. Nonetheless, that was not case, increasing the memory bank size achieved roughly same or slightly lower performance. The reason is unclear, but our guess is that as the memory access pattern is unchanged, breaking the loading into half might give the compiler more opportunity to shuffle around the loading operations to optimize.

2 Result

2.1 Performance for Different Thread Block Sizes

The performance for different thread block sizes is shown in Figure 1. The correspondence between thread block sizes and matrix tile sizes for the graph are shown in the table below:

Thread Block Size	BLOCKTILE_M	BLOCKTILE_N	BLOCKTILE_K
8x8	64	96	16
8x32	96	64	32
16x16	64	96	16
32x16	48	64	32

2.2 Optimal Thread Block Sizes

In all cases, we observe that the thread block of size 16x16 is optimal. According to our understanding, smaller thread block sizes result in lower occupancy and each thread must perform a lot of work for larger matrix tiles. On the other hand, larger thread block sizes improve occupancy, but __syncthreads() barrier incurs a lot of additional latency because if we have a lot of threads, there is a high variance in the speeds of execution of each thread. As a result, all threads would have to wait for the slowest thread and we are therefore bottlenecked by the slowest thread.

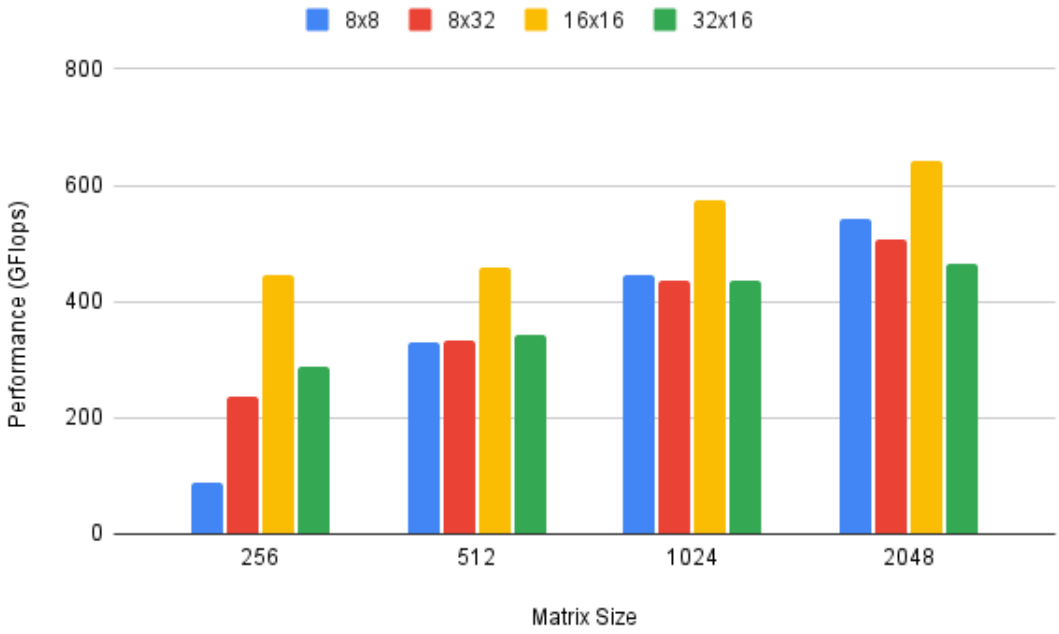

Figure 1: **Performance of Different Thread Block Sizes**

2.3 Peak Performance

N	Peak GF	Thread Block Size
256	436.9	16x16
512	558.3	16x16
1024	608.3	16x16
2048	628.3	16x16

3 Compare Results

Table 1 presents the comparison between our implementation and the naive algorithm on various metrics. First of all, the naive algorithm does not utilize the shared memory at all, while ours has high shared memory efficiency. Our shared memory throughput increases over matrix sizes, in line with the increases in SM Efficiency, Occupancy, and Instructions per Warp.

The higher arithmetic function unit utilization of our method is also indicative of our superior performance, as we can do more arithmetic operations without having to wait as much as the naive method for memory operations. We can see that with the exception of size 256, generally our method has higher IPC.

For the naive algorithm, the distribution of stall time is consistent across different matrix sizes, with execution dependency hurting the performance the most, followed by instruction fetch and data loading. On the other hand, our implementation spends little time waiting for data, but has costly thread synchronizations.

4 Analysis

4.1 Table and Graph

The performance plot is shown in Figure 2, and detailed measurements are presented in Table 2.

Size	256		512		1024		2048	
	Naive	Ours	Naive	Ours	Naive	Ours	Naive	Ours
Shared Memory Efficiency	0%	95.58%	0%	95.09%	0%	95.33%	0%	95.54%
Shared Memory Load Throughput	0B/s	500.87GB/s	0B/s	729.17GB/s	0B/s	767.37GB/s	0B/s	806.30GB/s
Arithmetic Unit Utilization	Mid	Mid	Mid	High	Mid	High	Mid	High
SM Efficiency	95.22%	85.64%	98.27%	89.97%	99.53%	93.31%	99.90%	98.27%
IPC	1.68	1.26	1.69	1.75	1.66	1.77	1.66	1.77
Occupancy	86.73%	12.49%	96.82%	44.66%	99.00%	58.53%	99.47%	61.85%
Instructions per Warp	2969	973	5897	19266	11753	38338	23465	76482
Stall (Instruction Fetch)	4.87%	9.62%	4.79%	2.57%	4.73%	2.03%	4.73%	1.94%
Stall (Execution Dependency)	14.97%	27.96%	13.41%	9.36%	12.90%	7.10%	12.85%	6.74%
Stall (Data Request)	2.02&	0%	2.14%	0%	2.33%	0.01%	2.29%	0.02%
Stall (Synchonization)	0%	4.81%	0%	7.53%	0%	10.14%	0%	11.06%

Table 1: Comparison between our implementation and the naive one.

4.2 Curve Shape

Our implementation generally achieved a constant fraction of CuBLAS for increasing matrix sizes, and both of them have their performance plateaued after size larger than 1024. This is not the case for BLAS, as its performance of size 2048 is roughly two times that of size 1024. The possible reason for the performances of CuBLAS and our implementation to not increase further for larger sizes is that we might have reached the point where all SMs are utilized and also possibly due to the fact that we may be seeing the effects of memory bottlenecks. Hence for BLAS its performance may also stabilize after a size much larger than 2048, where it hits the limit of cache sizes.

4.3 Irregularities

From the plot we can see spikes and dips of performances among different matrix sizes. If N is a multiple of block size, then the performance of N will be higher than that of N - 1, since we will be doing no zero padding, and there will be no branch divergence within a warp. On the other hand, the performance of N + 1 will be lower, because we will need to add blocks that almost consisted of zeros and waste more efforts. Moreover, matrix sizes that are not powers of two result in sub-optimal memory bus utilization since we either perform an additional transaction to fetch some more data or don't fully utilize the data fetched in a given transaction.

5 Evaluation

5.1 Roofline Model

To calculate peak performance, we use the hardware characteristics of Kepler K80, which has 64 double precision (DP) cores, with 13 streaming multiprocessing (SM) units such that each DP core can do 1 FMA (fused multiply-add) per cycle. Since each core is capable of executing 2 floating point operations per cycle, the total number of floating point operations per cycle is 64 * 2 * 13 = 1664 flops/cycle.

Since we set the GPU compute cores to operate at their maximum frequency of 875MHz, this gives us a peak performance of 1.456 TFlops.

The results are shown in figure Figure 3.

5.2 Arithmetic Intensity

Memory Bandwidth = 240 GB/s - This case corresponds to 30 G doubles/sec, for which we obtain a peak Q value of (1456 / 30) = 48.53 doubles/sec. For our implementation's matrix size of 1024, we obtain a performance of 608.3 GFlops, which corresponds to a Q value of 20.27.

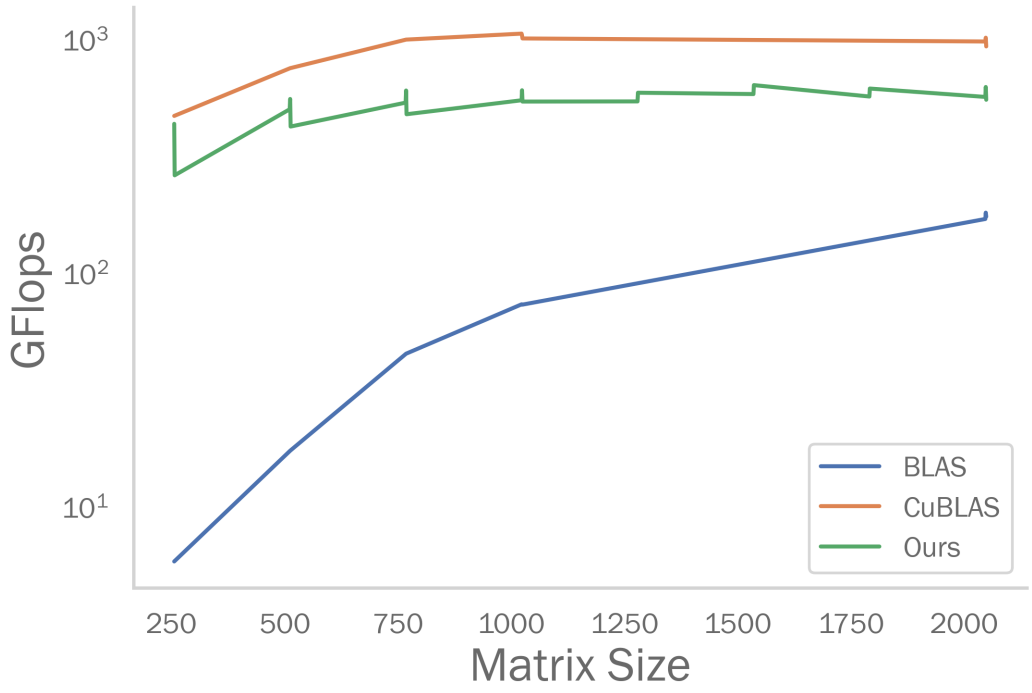

Figure 2: **Performance**

N	BLAS (GFlops)	CuBLAS	Your Result(GFlops)
256	5.84	472.5	436.9
257			263.1
511			504.5
512	17.4	756.5	558.3
513			425.5
767			538.6
768	45.3	1002.7	606.9
769			480.4
1023	73.7	1063.7	551.3
1024	73.6	1045.4	608.3
1025	73.5	1014.2	544.3
1279			545.1
1280			593.9
1535			586.1
1536			640.1
1791			572.1
1792			618.8
2047	171	984.2	569.8
2048	182	1021.6	628.3
2049	175	937.8	554.1

Table 2: Detailed performance between different implementations.

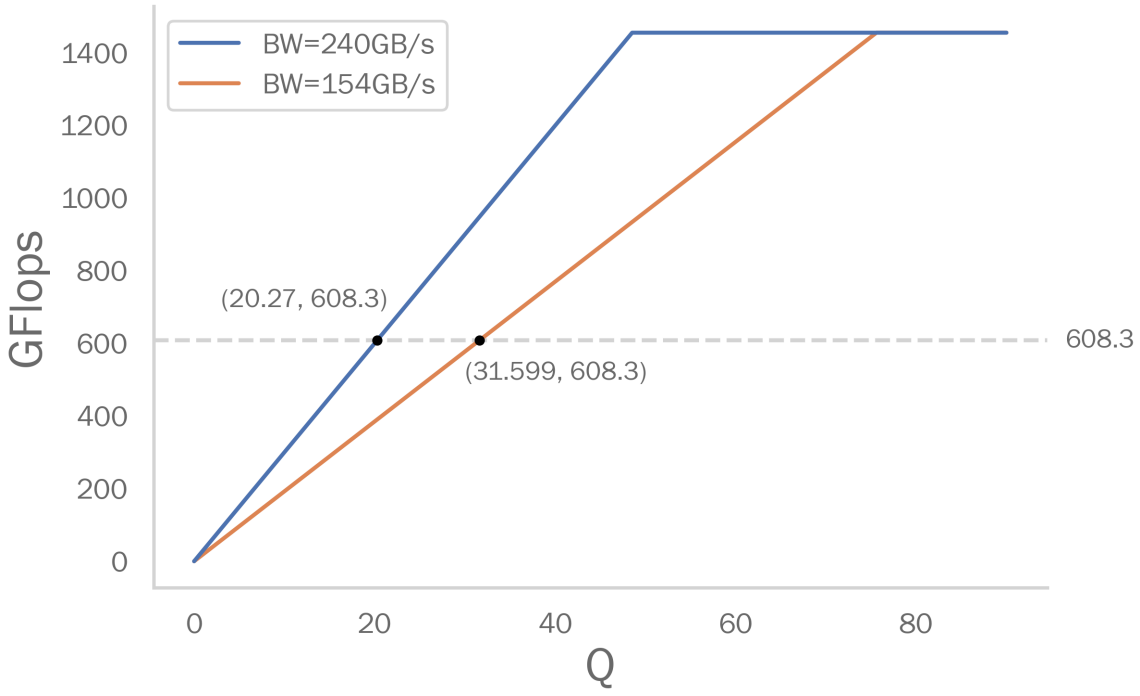

Figure 3: **Roofline**

Memory Bandwidth = 154 GB/s - This case corresponds to 19.25 G doubles/sec, for which we obtain a peak Q value of (1456 / 19.25) = 75.6 doubles/sec. For our implementation's matrix size of 1024, we obtain a performance of 608.3 GFlops, which corresponds to a Q value of 31.599.

Decreasing the memory bandwidth increases the arithmetic intensity because in order to compensate for the lower bandwidth, we have to perform more floating point operations to achieve the same level of performance.

6 Future Work

Permuted Shared Memory It is possible that in our implementation the read/write operations of the shared memory can encounter memory bank conflicts. Adjustments to the addresses where each thread accesses might mitigate this issue, albeit it requires extensive designing and calculating of the indices.

Zero Padding at Host Whenever the matrix size is not divisible by specified block sizes, we have to check the indices and conditionally set elements to zero inside the GPU. If these `if` statements are hurting the performance, we could possibly pad the matrices in host code, before launching them into GPUs, and truncate the returned result subsequently.

CUTLASS We would have definitely liked to implement a correct and fully functional version of the Nvidia Cutlass algorithm. Unfortunately, our implementation did not give us the benefits we expected to see, which is probably an artifact of a sub-optimal implementation. If we had more time, we could have explored solutions to the problems we faced.

References

[1] Andrew Kerr, Duane Merrill, Julien Demouth, and John Tran, "CUTLASS: Fast Linear Algebra in CUDA C++."

[2] Andrew Kerr, Timmy Liu, Mostafa Hagog, Julien Demouth, and John Tran, "Programming tensor cores: native volta tensor cores with cutlass."

[3] Julien Demouth, "Shuffle: Tips and Tricks."

[4] Yuan Lin and Vinod Grover, "Using CUDA Warp-Level Primitives."