1 Development Flow

1.1 Working of the program

The crux of our program is to divide and subdivide the input matrices to take advantage of the large number of parallel computational units available in the GPU. Moreover, we have have also optimized our memory `LD`s and `ST`s to take advantage of the memory hierarchy in the GPU as well.

Tiling: The input matrices arr initially divided into tiles. Matrix A is divided into tiles of size $TILE\_M \times TILE\_K$, and matrix B is divided into tiles of size $TILE\_K \times TILE\_N$. Each pair of tiles from A and B are used to compute a portion of C of size $TILE\_M \times TILE\_N$.

Shared memory: Given the large number of parallel compute units in the GPU, each computing a sub-portion of the output matrix from sub-portions of the input matrices, it is important to ensure that the computation does not have a memory access bottleneck. We ensure that this does not occur by using *shared memory*. Our program loads A and B into shared memory for each thread in a thread block to be able to access it. Shared memory is orders of magnitude faster than global memory (since it is on-chip memory). Shared memory latency is about 100 times lower than that of uncached global memory[1].

Zero Padding: We padded the shared memory loading by checking if every memory access a thread does is a legal accesses into memory; if the access is legal, the data is loaded from global memory to shared memory, else it loads a 0. This ensures that during computation, even if the computation uses data that is out of bounds, it will insterad get a 0 and hence, there is no need to monitor computation of elements in matrix C. While storing data, there is another check implemented that ensures that garbage data (which exists outside the range of the matrix) is not stored back into memory; instead the store is skipped altogether.

Instruction-level parallelism (ILP): We deploy each thread to compute more than one element of the output matrix C, as opposed to each thread calculating just one value of C. We keep track of map threads to computations by indexing the shared memory accessed by the threads by the thread IDs. This is computed as an inner product that is stored in a register wherein we declare a 2D accumulator array which accumulates the partial sums for each of the elements of matrix C that a thread is responsible for. This is just as the naive algorithm which computes inner products to compute each element of the output matrix C and in contrast to the approach followed by the CUTLASS algorithm which follows the outer

product approach.

Lastly, we map the computations stored in the 2D accumulator housed in the register to its correct position in the global output matrix C.

1.2 Development process and ideas

There were four main things that we discussed and implemented in our pursuit of high performance GEMM: 1) the use of shared memory, 2) global memory coalescing, 3) instruction-level parallelism, and 4) implementing a variation of CUTLASS[2].

The first step in the development process was to use shared memory to store tiles of matrices A and B. This was fairly straightforward. What needed to be modulated were the values of $TILE\_M$, $TILE\_N$, and $TILE\_K$. These could not be arbitrarily big since the amount of shared memory is finite. We chose configurations of 64×16 and 128×16 as we found these to work quite well (see Section 1.3).

Next, we chose to implement instruction-level parallelism to have a thread compute multiple elements of the output matrix C. Here, we had two parameters to play with: 1) the number of instructions in parallel per thread, and 2) whether it would be interleaved or not (see Figure 1).

Once we were able to get a significant amount of performance using the above two approaches (and indeed, combining them), we chose to focus on implementing a variation of CUTLASS. Within our implementation of CUTLASS, we varied the following parameters: 1) size of shared memory, and 2) thread block size. We discussed the variation of shared memory size earlier (see Section 1.1), however, changing thread block size has a profound impact on the geometries of a lot of other structures that we used; but primarily it affects the number of threads available per block. With a smaller thread block size, one would have fewer threads per block and that in turn would require ILP to be increased in order to achieve the same level of performance. Thus, we sought to strike a good balance between the number of threads (by tuning the thread block size) and the ILP.

1.3 Ideas that worked, didn't work, and why

The first barrier we encountered was that we weren't able to access larger configurations for $TILE\_M$, $TILE\_N$, and $TILE\_K$ because the amount of shared memory is finite. This

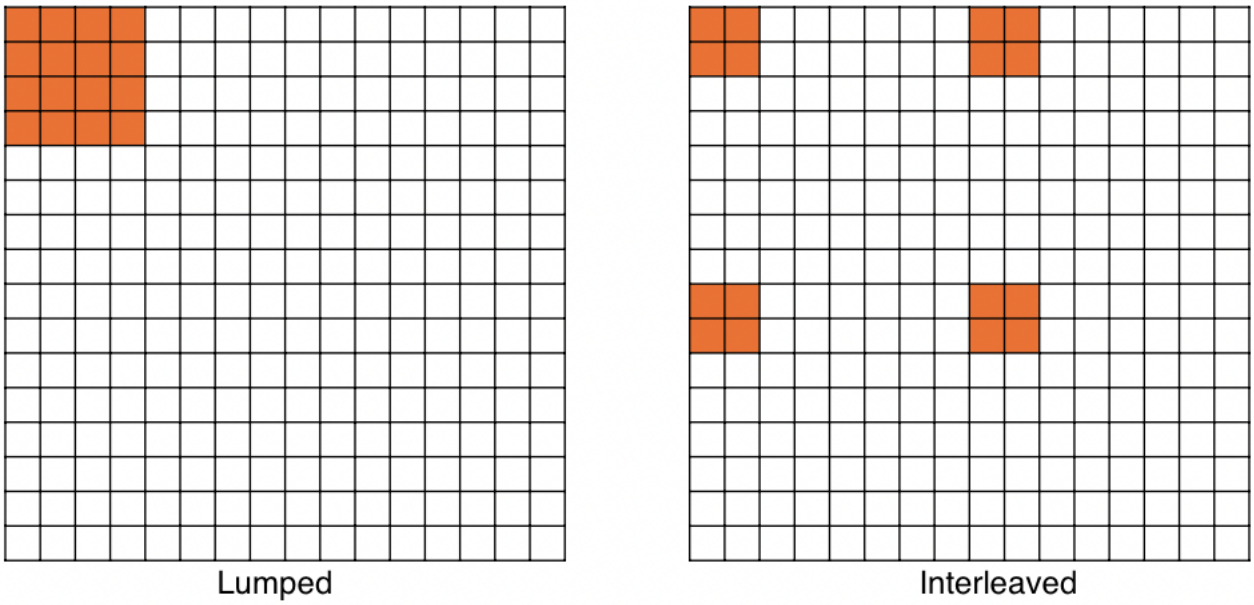

Figure 1: Instruction Level Parallelism configurations [4]

constrained our search for an ideal shared memory size configuration. We arrived upon configurations of 64×16 and 128×16 through a process of grid search and found that these sizes yielded the maximum amount of performance with respect to the rest of our configuration.

Next, we found that certain ILP configurations performed better than others. As mentioned earlier, we looked to vary two aspects of ILP: 1) the number of parallel instructions per thread, and 2) the configuration of ILP (see Figure 1). Through a process of grid search, were able to find a suitable number of parallel instructions per thread. We knew that am exorbitantly high number of parallel instructions per thread will cause too many shared memory accesses which will degrade performance[5]. Next, we chose an interleaved configuration for ILP over the lumped configuration[5].

Finally, when we we implemented CUTLASS using different configurations of shared memory and thread block sizes, we observed a few things:

- *Bank conflicts*: Let's consider one of our shared memory size configurations: $(TILE\_M, TILE\_N, TILE\_K) = (64, 64, 16)$. Now, if we look at Figure 2, we can see that B (SMEM) will have multicast shared memory access[1] because our GPU has a compute capability ¿= 2.0. Thus, this will eliminated the possibility of bank conflicts

occurring there. However, if we look at A (SMEM), then the order of data accesses clearly indicates the possibility of bank conflicts. This was hypothesis was validated in our earlier implementation wherein we weren't cognizant of bank conflicts and it led to a huge degradation of performance. In order to counter bank conflicts, in our code we declared A (SMEM) as a 2D array of 64×17

- *Smaller matrix sizes*: We observed that one of our shared memory configurations (128×16) yielded a poor result with matrices of size 256, while yielding better performances with matrices of sizes 512, 1024, and 2048 (see Table 1). We hypothesize this may be because CUTLASS is designed in such a way that it requires a large number of compute units and data in order to showcase its true effectiveness. This fact was corroborated when we used our CUTLASS kernel on the Tesla T4 GPU and the pattern seen in table 1 was reversed; the larger shared memory size configuration preformed better than the smaller shared memory configuration (see Table 8).

- *Inefficient global stores*: We expected CUTLASS to give us a performance > 800 GFlops for the bigger matrices for any shared memory configuration on the K80 GPU. However, as seen in table 1, we don't see those kind of results. We hypothesized that this may be because of a bottleneck caused by inefficient global stores occurring when the 2D accumulator in the register stores back the result into the global C matrix. To check our hypothesis, we removed the global storage part of the code and observed the performance jump from around 600 GFlops to >= 850 GFlops.

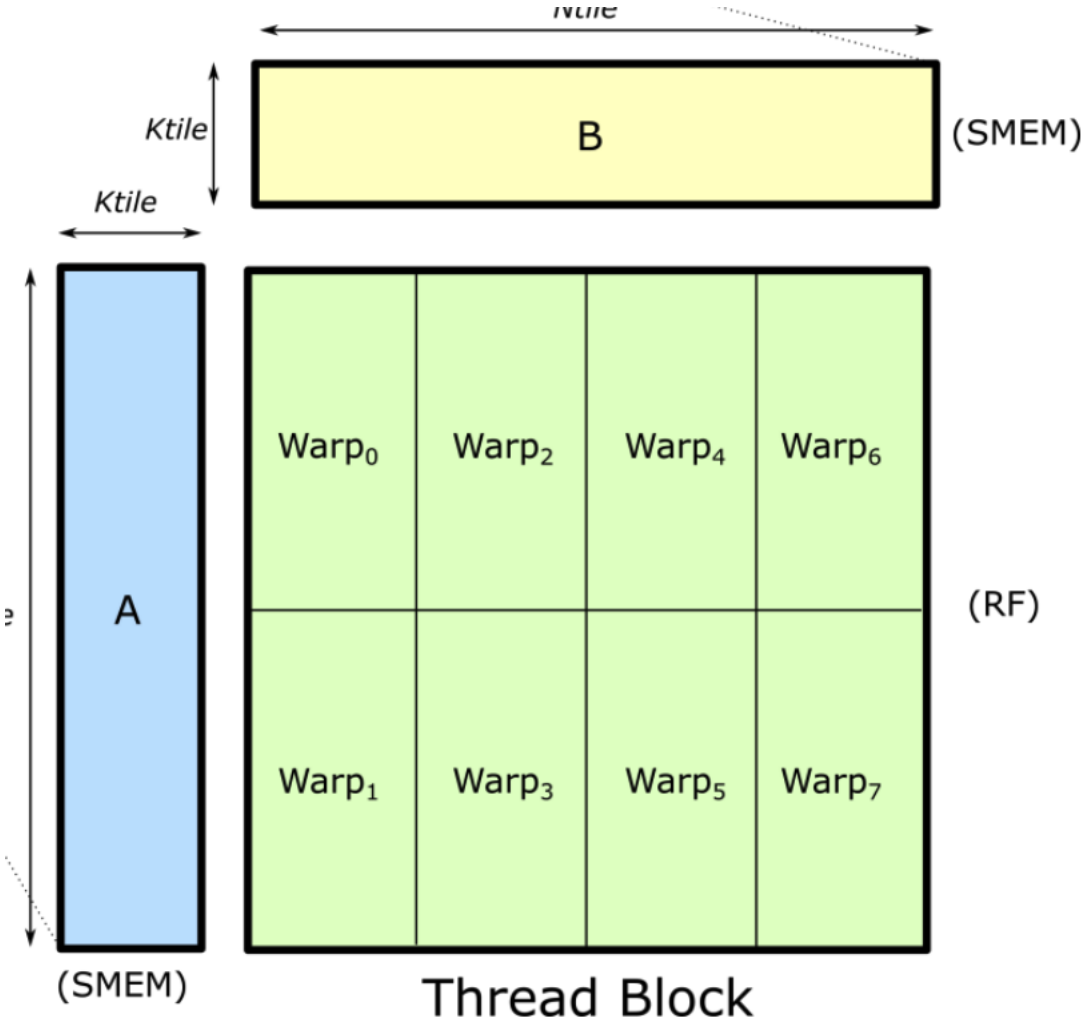

Figure 2: Shared memory accesses and bank conflicts [3]

	Peak GF	
N	*128 × 16*	*64 × 16*
256	84.1	330.8
512	415.7	415.9
1024	521.7	609.2
2048	560.8	651.2

Table 1: CUTLASS performance on a 16×16 thread block configuration using two different shared memory sizes using Tesla K80

2 Results

2.1 Performance for different thread block sizes

Experiment 1: Constant ILP

For this experiment, the number of computations was kept constant with a varying thread block size. This results in a varying shared memory size. For 2×2 computations per thread and with the shared memory sizes for A and B mentioned in the table such that each tile of C computes as $C \mathrel{+}= A \times B$. Clearly the 16×16 thread block performs the best for N=256 while the 32×32 thread block leads in the other sizes. This is due to the memory bottleneck which affects the 32×32 thread block the least. For N = 256, the number of computations relative to the number of memory loads is lower as compared to the larger sizes. A smaller thread block size can take advantage of this.

Shared memory size	**Thread block size**	**N**			
		256	**512**	**1024**	**2048**
A: 16, 16; B: 16, 16	8*8	331.5	354.4	357.1	356.5
A: 32, 32; B: 32, 32	16*16	400	424.2	440.2	442.9
A: 64, 64; B: 64, 64	32*32	371.2	558.3	579.5	601.2

Table 2: Performance for constant number of elements computed per thread

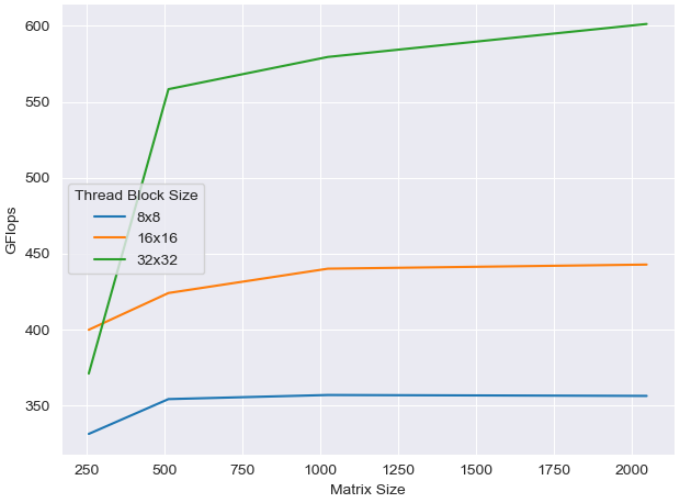

Figure 3: Comparison of performance over different thread block sizes for different matrix sizes over a constant ILP (see Table 2)

Experiment 2: Constant thread block size

For this experiment, the thread block size was kept constant at 16×16. This experiment reveals the bottleneck caused by memory. The number of memory accesses were kept constant for different block sizes which results in different number of elements computed by each block. For a lower block size, there is little variation in performance values for different matrix sizes. This is due to the huge memory bottle neck resulting from a small block size. The general trend indicates an increase in the performance as the matrix size increases and this is due to more computations per memory accesses in larger block sizes. The biggest shared memory size, 128×16 achieves the best performance at matrix size equals 2048 and computations was kept constant with a varying thread block size. This results in a varying shared memory size. For 2×2 computations per thread and with the shared memory sizes for A and B mentioned in the table such that each tile of C computes as $C \mathrel{+}= A \times B$. Clearly the 16×16 thread block performs the best for N=256 while the 32×32 thread block leads in the other sizes. This is due to the memory bottleneck which affects the 32×32 thread block the least. For N = 256, the number of computations relative to the number of memory loads is lower as compared to the larger sizes. A smaller thread block size can take advantage of this.

Computations per thread	**Memory accesses per thread**	**Shared memory size**	**N**			
			256	**512**	**1024**	**2048**
8*8 = 64	8 + 8 = 16	A: 32, 64; B: 64, 32	400	424.2	440.2	442.9
4*4 = 16	8 + 8 = 16	A: 64, 32; B: 32, 64	395.6	651.6	661.4	687.3
2*2 = 4	8 + 8 = 16	A: 128, 16; B: 16, 128	225.8	591.8	772.9	870.8

Table 3: Performance for constant thread block size

Experiment 3: Constant shared memory size

For this experiment, we kept the shared memory size was kept constant such that A consists 64×32 elements and B consists of 32×64 elements. The tile of matrix C calculated equals 64×64 elements. The block size was varied as shown in the table. In this configuration, the 8×8 thread block performs very poorly for all sizes with the 16×16 thread block achieving the best performance across all sizes.

Computations per thread	Memory accesses per thread	Thread block size	N			
			256	**512**	**1024**	**2048**
8*8 = 64	32 + 32 = 64	8*8	151	317.4	391.4	407.6
4*4 = 16	8 + 8 = 16	16*16	395.6	651.6	661.4	687.3
2*2 = 4	2 + 2 = 4	32*32	371.2	558.3	579.5	601.2

Table 4: Performance for constant shared memory size

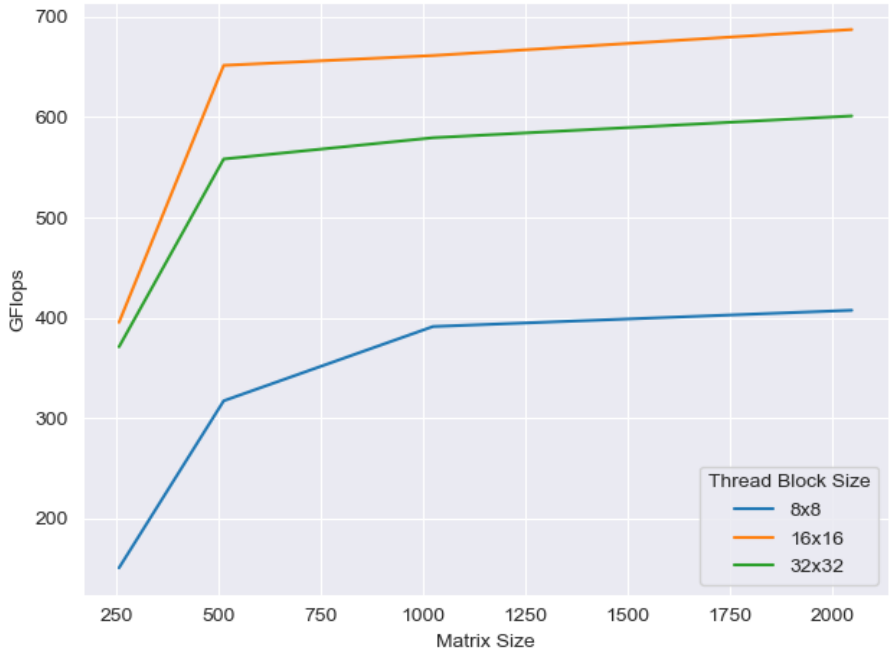

Figure 4: Comparison of performance over different thread block sizes for different matrix sizes over a constant shared memory size (see Table 4)

2.2 Optimal thread block sizes for different matrix sizes

The thread block sizes tested by us were: 8×8, 16×16 and 32×32 because as explained in class, the number of memory loads increases as the perimeter of the thread block size whereas the number of computations increases as the area of the block. In order to maximize the number of computations to memory accesses in our matrix calculaiton, we consider square shaped thread blocks.

a) $\underline{8 \times 8}$ performs the worst for all sizes as there are not enough threads available for the

scheduler to schedule. Each thread in this thread block configuration has to do more memory accesses which is the inherent bottle neck and causes a dip in the performance for all sizes. Even though each thread can compute more elements, as the number of threads is limited it cannot make use of the available parallelism in a GPU.

b) $\underline{16 \times 16}$ performs the best overall for multiple values of the matrix size with the 32×32 thread block size beating it in very few configurations. The 16×16 thread block has a good balance between the number of threads available, the number of computations each thread performs and the number of memory accesses performed by each thread. Our solution has a thread block of 16×16 with 64×64 total shared memory split as 64×32 and 32×64 split between A tile and B tile respectively. The C tile calculated has a dimension of 64×64. With each thread calculating and storing 4 elements; loading 4 elements of both A and B matrices into shared memory. Although the maximum number of threads that can be used in a block is 1024, 16×16 gives us the best performance.

b) $\underline{32 \times 32}$ is the maximum available number of threads that can be used in a thread block. This configurations achieves better performance than 16×16 thread block for a limited number of configurations, when the number of computations performed by each thread is constant. Ideally, the 32×32 thread block should achieve better performance than the 16×16 thread block but this is not the case most probably because of memory conflicts which lead to inefficient memory accesses while loading and storing data.

2.3 Peak performance for different matrix sizes

N	**Thread block size:** 8×8	**Thread block size:** 16×16	**Thread block size:** 32×32
256	331.5	395.6	371.2
512	354.4	651.6	558.3
1024	357.1	772.9	579.5
2048	356.5	870.8	601.2

Table 5: Peak performance for different matrix sizes

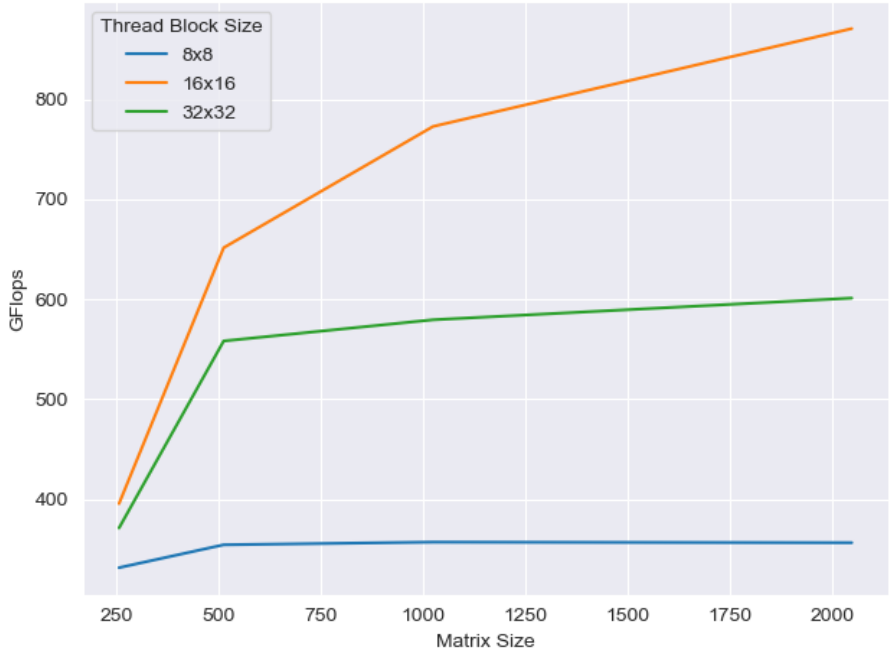

Figure 5: Comparison of peak performance over different thread block sizes for different matrix sizes (see Table 5)

3 Performance compared to naive implementation

3.1 Performance for naive and custom implementation

N	Naive	Custom
256	93.4	381.0
512	96.2	602.5
1024	95.3	614.6
2048	95.4	621.7

Table 6: Peak performance for Naive vs custom kernel

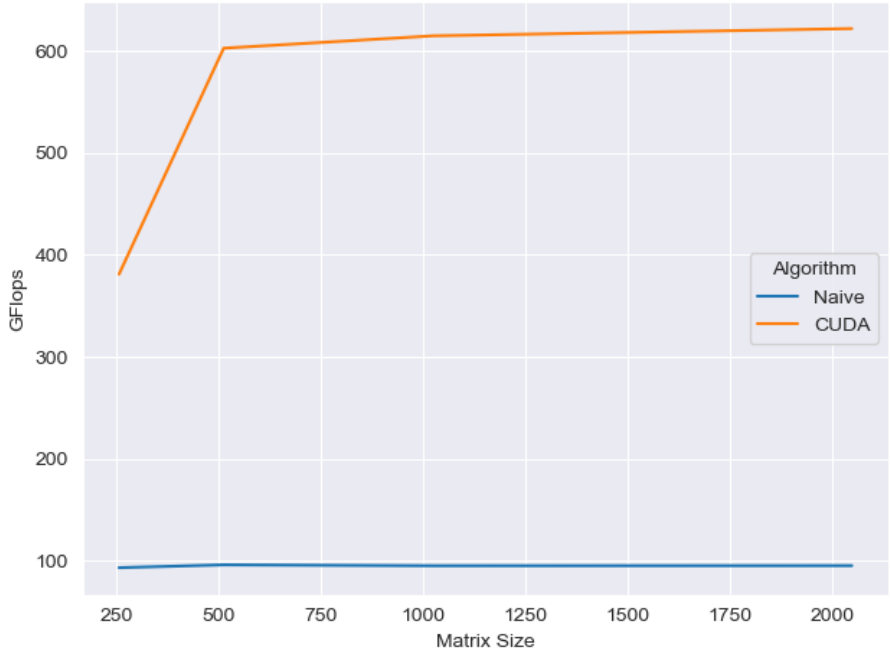

Figure 6: Comparison of performance of CUDA over Naive implementation (see Table 3.1)

4 Analysis

4.1 Performance

Our best thread block size for $n = 1024$ is 16×16. Using that thread block size, we obtain the following plot of performance

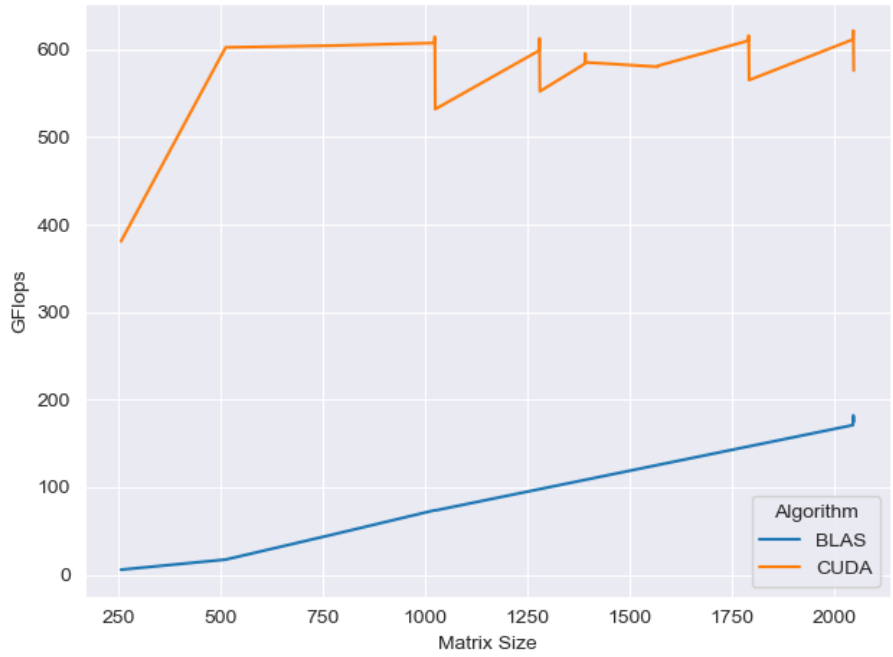

Figure 7: Performance for thread block $= 16 \times 16$, CUDA vs. BLAS

4.2 CUDA vs. BLAS

As wee see in Figure 7, the CUDA curve exhibits a roofline model behaviour, following which we see some dips for certain matrix sizes, yielding a sawtooth like shape. In contrast, the BLAS curve is linearly increasing. We see a plateau in the CUDA curve after we hit $n = 512$. This is because between $n = 256$ and $n = 512$, there was a lot of scope to utilize more compute units and hence achieve more parallelism. It plateaus after that because there isn't much room to take advantage of any more parallelism from this point on. BLAS on the other hand employs a blocking strategy which scales well with increasing matrix size, hence explaining the linearly increasing nature of the BLAS curve.

4.3 Anomalies

The reason the CUDA curve is shaped that way is because we see drop in performance at sizes of $n + m$ where $m << n$. This happens because for sizes $n + m$, the matrix does not break into block sizes completely and there are additional elements that lie outside the

blocks. As these blocks also need to be calculated to achieve the correct result, we need to add additional padding to enable the matrix to be split into proper block sizes. These padded elements not only result in wasted calculations, they also cause divergence in the code dues to checks required to ensure no illegal memory accesses and checks required to ensure that stores do not access wrong memory locations. Due to this overhead, we have a huge performance drop when going from matrix sizes 2^n to $2^n + 1$. There is a performance drop from 2^n to $2^n - 1$ too, but this is not as significant as the previous one because the number of useless computations is much lesser as compared to the previous case.

4.4 OpenBLAS vs. cuBLAS vs. CUDA

N	OpenBLAS	cuBLAS	rprasad-bsaldanha
256	5.84	472.5	381.0
512	17.4	756.5	602.5
768	45.3	1002.7	604.5
1023	73.7	1063.7	607.6
1024	73.6	1045.4	614.6
1025	73.5	1014.2	532.0
1279	-	-	598.9
1280	-	-	612.8
1281	-	-	552.3
1391	-	-	583.9
1392	-	-	595.6
1393	-	-	585.4
1569	-	-	580.4
1570	-	-	581.5
1791	-	-	610.2
1792	-	-	615.9
1793	-	-	565.2
2047	171	984.2	611.6
2048	182	1021.6	621.7
2049	175	937.8	576.1

Table 7: CUDA Performance vs OpenBLAS and cuBLAS

4.5 Speedup of CUDA versus OpenBLAS and cuBLAS

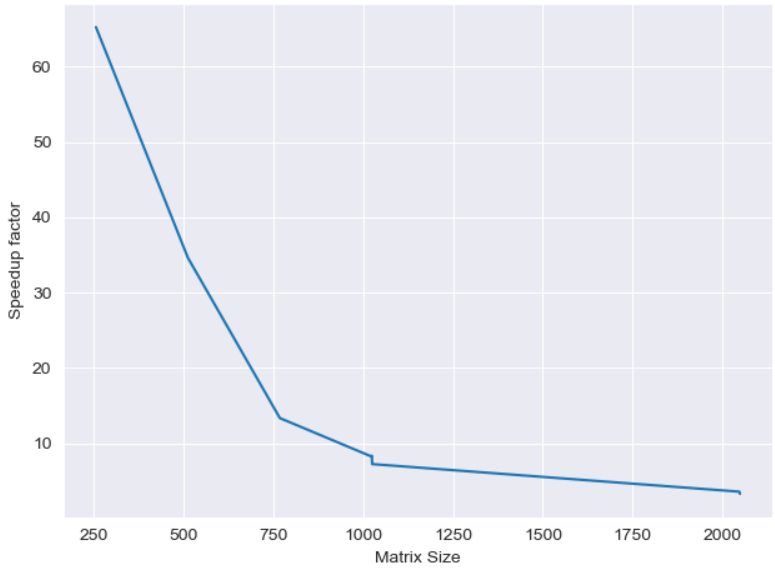

Figure 8: Speedup factor of CUDA compared to OpenBLAS

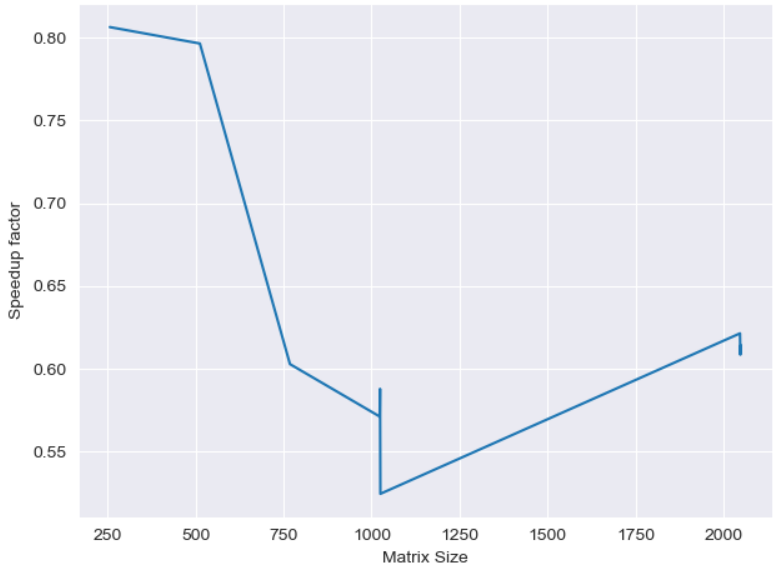

Figure 9: Speedup factor of CUDA compared to cuBLAS

5 Ropofline plots

5.1 Roofline plot @ 240 GB/s

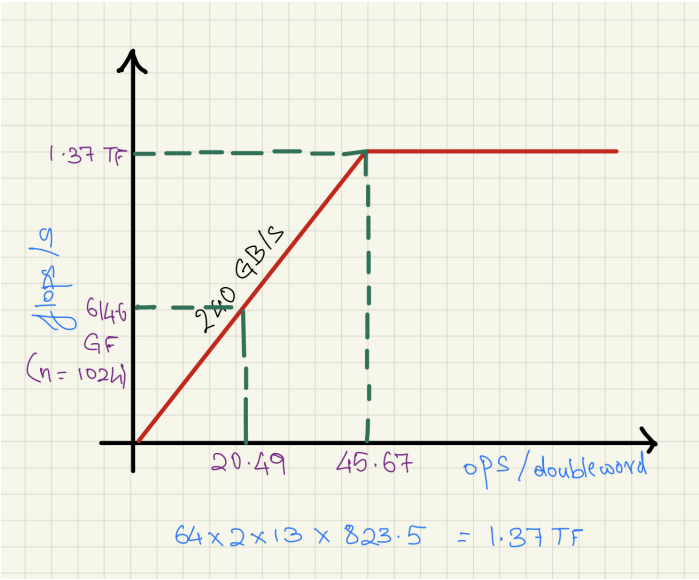

Figure 10: Roofline plot at 240 GB/s bandwidth

The peak performance is calculated as $peak\ performance\ =\ number\ of\ double\ precision\ cores \times number\ of\ SMs \times clock\ frequency \times number\ of\ operations\ per\ cycle\ per\ core(1fma)$ For our configuration, this evaluates to $64 \times 13 \times 823.5 \times 2\ =\ 1.37\ TF$

For the peak performance achieved by our kernel at N=1024, i.e. 614.6 GFLops, we evaluate our q to be 20.49 *ops/doubleword* (see Figure 10).

5.2 Roofline plot @ 154 GB/s

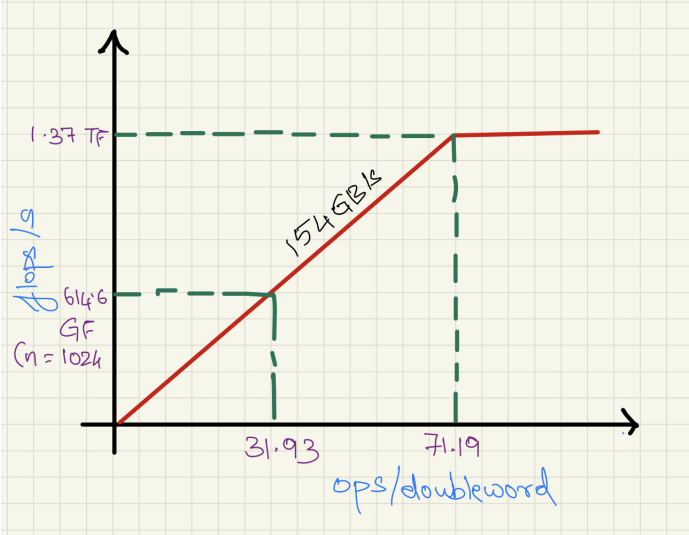

Figure 11: Roofline plot at 154 GB/s bandwidth

At bandwidth = 154 GB/s, we evaluate our q to be 31.93 *ops/doubleword* (see Figure 11). Thus, we see that the value of q has increased with a decrease in bandwidth for the same performance.

6 Future work

The CUTLASS algorithm we implemented for the K80 machine did not yield the expected performance values. We believe that the bottle neck in our code is in the store phase for the matrix C elements after the computation. We came to this conclusion by testing multiple shared memory block shapes and sizes and experimentation with the block sizes. When the store to memory is skipped, a drastic increase in performance is observed which is typical because memory accesses are the operations that yield the most latency in matrix multiply. We feel that tuning the CUTLASS algorithm that we implemented will definitely achieve better performance on a K80 machine too.

7 T4 Extra credit

For extra credit, we ran our implemented cutlass kernel on a T4 machine which yielded us the following numbers when the matrix multiplication was repeated enough times to achieve a 5 second computation time. The shared memory blocks used have sizes 128×16 and 16×128 for A and B tiles respectively. The C tile calculated has a dimension of 128×128 which is divided into 8 warps which each have 32 threads allocated to them. Each warp has 2048 elements to be calculated i.e. each thread handles 64 computations. As described in section 1, we used a hybrid computation model which uses both a clumped and spread out computation approach to achieve better global memory coalescing better shared memory bank accesses. Each thread brings a 8×1 strip of A and a 1×8 strip of B into registers and computes a 8×8 block of C.

N	**Peak TF**	**Thread block size**
256	419.2	16×16
512	1788.8	16×16
1024	3547.9	16×16
2048	3662.6	16×16

Table 8: Peak performance for different matrix sizes for Tesla T4

References

[1] Mark Harris. Using shared memory in cuda c/c++, Jan 2013.

[2] Andrew Kerr, Duane Merrill, Julien Demouth, and John Tran. Cutlass: Fast linear algebra in cuda c++, Dec 2017.

[3] Andrew Kerr, Duane Merrill, Julien Demouth, and John Tran. Cutlass: Cuda template library for dense linear algebra at all levels and scales, May 2022.

[4] Tanmay Anil Patil. Matrix multiply using simt, May 2022.

[5] Vasily Volkov. Understanding latency hiding on gpus. 2016.