CSE 260 PA2 Report

Mohit Shah(A59005444)

Monil Shah (A59012111)

Github repo link : [Code]

Introduction

This report is a brief documentation and analysis of the steps we did in improving matrix multiplication performance on GPU boards namely K80 and T4. The report is organized to document the procedure of optimizing the kernel from naïve and instruction level parallelism all the way to implementing CUTLASS inspired cross product

Development Flow

1.a **How program works**

The main logic of this assignment is within the GPU kernel that implements matrix multiplication between matrix A and B and stores it in matrix C. CPU takes care of transferring the required array of matrices and the GPU kernel code that can run-on the GPU. CPU later copies back this result C and compares it with the reference result. There are three versions of implementation

- Naïve – Multiply a single row of A with a single row of B and store in the corresponding location of C. This works for all sizes of N
  - Example Pseudocode

```
For(i=0; i<N; i++)
   For(j=0; j<N; j++)
      For(k=0; k<N; k++)
         C[i][j] = A[i][k] * B [k][j]
```

- ILP – In this method, we break the bigger matrix into smaller matrices of dimension BLOCKTILE_M x BLOCKTILE_N. For our case this becomes 32x32. Each thread block (16x8) is responsible for computation on this tile. Each thread loads **Tile[Tid_y + k*stride][Tid_x+0]** and **Tile[Tid_y + k*stride][Tid_x+16]** location elements of the tile from 4 different rows in strided form (stride=8). The matrix tiles are brought in shared memory, then few elements of C are computed and written into matrix C. Shared memory is padded with 0 for the tile which is not a multiple of the blockTile dimension.
  - Example pseudocode

```
For(k=0; k < BLOCKTILE/BLOCKDIM_Y; k++)
      As[SHM_Y][SHM_X] = (GMEM_Y < N && GMEM_X < N ) ?
A[GMEM_Y][GMEM_X] : 0
    As[SHM_Y][SHM_X+16] = (GMEM_Y < N && GMEM_X+16 < N ) ?
A[GMEM_Y][GMEM_X+16] : 0
//Load B similar to A
For(k=0;k<BLOCKTILE;k++)
   For(j=0; j<BLOCKTILE/BLOCKDIM_Y; ++)
      Cr[J][0] = As[Tidx+0][k] * Bs[k][Tidy +j];
      Cr[J][1] = As[Tidx+16][k] * Bs[k][Tidy +j];
```

```
For(k=0; k< BLOCKTILE/BLOCKDIM_Y; k++)
    C[GMEMY+Tidy+k*BLOCKDIM_Y][GMEM_X+Tidx+0] = Cr[k][0];
    C[GMEMY+Tidy+k*BLOCKDIM_Y][GMEM_X+Tidx+16] = Cr[k][1];
```

- CUTLASS inspired Outer Product
    - In this method we break bigger matrices into smaller rectangular matrices. These tiles are loaded into Shared Memory. Within these smaller matrices, we create smaller sub-tiles which will generate the outer product. Working sets of A and B are first brought from Global Memory into Shared Memory. A subset of values are then brought into registers from the shared memory,then the outer product is computed on registers and finally stored back into the matrix.
        - Example Pseudocode
            - Load Tile A and B into Shared Memory from Global Memory
            - Take a fragment from shared memory into registers
            - Compute Outer product using these registers
            - Store the computed values to C matrix in Global Memory

1.b Development Process

- **First Step was to implement a working tiled matrix multiplication**
    - For this we first tried to visualize what is the required Tile size, how much shared memory we need, how many threads we need to compute a tile
    - Next, we figured what should be the global memory address to load the tiles into shared memory and computed the inner product by loading values from shared memory
    - Once the kernel worked for powers of 2, we worked on how to handle the fringe case where matrix size was not divisible by tile size and pad shared memory with zeros for out of bound access.
    - After having the code work for all sizes of N, we started optimization steps
- **Second step was to optimize the configuration and kernel to achieve better performance**
    - For this we started with doing more work per thread. We tried both types of configurations of multiple block loading per thread.
        - We implemented interleaving with a stride of 16 and contiguous access to adjacent elements (i.e., 2*tidx and 2*tidx+1) in X-dimension
            - Interleaving gave better results, so we proceeded with that
    - Next, we implemented more work per thread, by reducing the Y dimension threads. Since assignment works on double precision, we wanted 128 Bytes of access per warp. And then implemented 4 reads of rows per threadIdY (i.e., 4*tidy+[0,1,2,3])
    - Next, we played around with tile size and still found 32x32 to be the most optimal for our case.
- **Third Step was to implement CUTLASS inspired code**
    - We went through the documentation of CUTLASS and used that idea to implement 2-D tiling and making use of registers which are faster to access than shared memory and can also be reused by threads.
    - Started with a square shared memory size of 32*32 for both A and B which comes to a total share memory usage of 16KB which is much lower than the total available shared memory of 48kB per block.
    - So we tried with 64*64 tiles of A and B but that led to over utilization of shared memory and limited the number of thread blocks that can be active in an SM at one time.

  - Then we tried out a tile size of 96*32 for A and 32*64 for B which brings the shared memory utilization to 40KB which led to a shared memory efficiency of 93% as indicated by the profiler.
  - Calculated the addressing scheme such that threads in a warp send a coalesced global memory access and load data into shared memory from global memory.
  - Implemented the cross-product by first loading A and B's elements into registers from shared memory. So we experimented with different thread block sizes starting with 8*8 for the same tile size mentioned above. Due to a small number of threads calculating a large tile size, each thread had to load a large number of values of A and B from global to shared memory and then from shared memory to register, which also caused bank conflicts in shared memory and register spillage at thread level. Then we also tried a thread block dimension of 16*16 leading to optimal performance from high thread occupancy and efficient register usage. Now each thread was calculating 24 (6 * 4) elements of C which were then transferred to global memory.
- **Fourth step was to port code from K80 to T4**
  - We first made sure that code worked for K80.
  - Then we setup T4 instance and checked Naïve implementation
  - Next, we ported Interleaving ILP code, and modified the thread block configuration as well as kernel from 16 to 32 in X dimension. We checked the performance here.
  - Next, we ported the CUTLASS inspired code which calculated the outer product, modified the thread block and block tile configurations for the most optimal performance.

1.c What worked well and what didn't

- Initially we spent a lot of time thinking about how the kernel can work for non-powers of 2. And after a lot of time, we realized that the condition to check is almost already present in the access itself. Implementing tiled multiplication with 32x32 tile and 32x32 thread block was quite easy.
- We later faced a tough time implementing optimized out-of-bounds checks for interleaving ILP based kernel. We thought this would lead to too much thread divergence and would hurt performance, however, we could not get around a better solution to avoid those branch conditions.
- We realized later that we will need to implement a CUTLASS type cross product to get extra credits. We already learned that we first need to understand the accesses and then start coding. Hence, we first thoroughly read the logic , came up with an access pattern and then coded it
  - Debugging this was again challenging since it was using a cross product. So first we tried to rule out any issues with accesses by using dot product style, and then used cross product.

Result

2.a **Performance for different thread block configurations**

Size	Thread Block	Tile	Perf (GF)	Thread Block	TIle	Perf (GF)	Thread block	Tile	Perf (GF)
256	BY = 8 BX = 8	Bm = 96 , Bn = 64, Bk = 16	111.9	BY = 16 BX = 16	Bm = 96 , Bn = 64, Bk = 32	383	BY = 32 BX = 32	Bm = 96 , Bn = 64, Bk = 32	372
512	BY = 8 BX = 8	Bm = 96 , Bn = 64, Bk = 16	327.3	BY = 16 BX = 16	Bm = 96 , Bn = 64, Bk = 32	551	BY = 32 BX = 32	Bm = 96 , Bn = 64, Bk = 32	417
1024	BY = 8 BX = 8	Bm = 96 , Bn = 64, Bk = 16	485.9	BY = 16 BX = 16	Bm = 96 , Bn = 64, Bk = 32	623	BY = 32 BX = 32	Bm = 96 , Bn = 64, Bk = 32	481
2048	BY = 8 BX = 8	Bm = 96 , Bn = 64, Bk = 16	513.8	BY = 16 BX = 16	Bm = 96 , Bn = 64, Bk = 32	629	BY = 32 BX = 32	Bm = 96 , Bn = 64, Bk = 32	487

2.b **Explain why some geometry is higher performance than others**

The main idea behind parallelism is to have multiple independent computations to hide latency. With smaller tile dimensions like 8x8 we need more global accesses for the same amount of computation as accesses of 128 Bytes are from 2 different thread blocks now, which can not be coalesced. If thread block size is too small, we might end up reaching a limit on how many thread blocks can be mapped on a single SM. This is where our parameter sweeping helps. To find the optimal size that gives the most performance by hiding latencies arising due to stalls as well has more global access.

The other constraint is the usage of big thread blocks. Kepler has 13SMs and each SM can have a maximum of 2048 threads, which limits the number of blocks that can be scheduled on all SMs. For a 1024x1024 we need ~160 thread blocks which can not be mapped on all SMs on a single hand, thereby incurring latency. Thus a 16x16 thread block size is the most optimal as it has maximum GPU occupancy.

2.c **Peak GF for configuration in 2.b**

Size	Perf	Thread block size	Block Tile
256	383	16x16	Bm = 96 , Bn = 64, Bk = 32
512	551	16x16	Bm = 96 , Bn = 64, Bk = 32
1024	623	16x16	Bm = 96 , Bn = 64, Bk = 32

2048	629	16x16	Bm = 96 , Bn = 64, Bk = 32

Quantitative analysis

3.a **With respect to Naïve**

Perf comparison

Size	Naive Perf GFLOPs	Reps	Thread block size	Our code Perf GFLOPs	Reps	Thread block size
256	95.54	1300 (4.6s)	16x16	389	5000(4.3s)	16x16
512	97.25	175(4.83s)	16x16	572	1000(4.6s)	16x16
1024	96.34	20 (4.45s)	16x16	634	140(4.73s)	16x16
2048	99.41	2(3.45s)	16x16	637	15(4.04s)	16x16

As can be seen from performance numbers, Naive performance almost flattens over large sizes. Whereas, the performance for our code increases to 1024 and then starts to flatten.

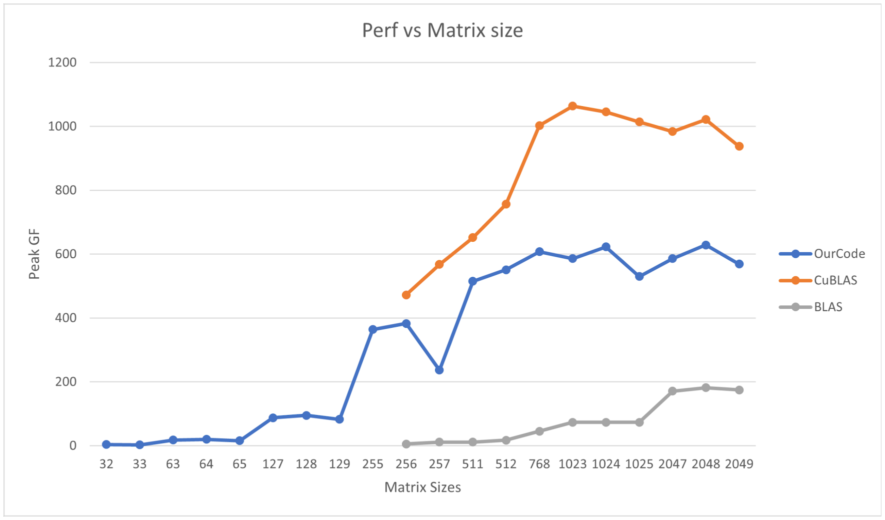
Profiled Runs

Size	Type	Req GLD throughput (GBPs)	Req GST throughput (GBPs)	SM Efficiency	Stall Reason (Execution Dep)	Sync Stall	Fetch Stall	Stall Reason (mem Dep)
256	Our	~27	~3.8	~85%	~26%	~2.5%	~17%	~0%
	Naive	~48	~0.3	~82%	~50%	0%	~8%	~2.56%
512	Our	~41	~2.9	~93%	~17%	~3.8%	~5.8%	~0%
	Naive	~121	~0.4	~90%	~29%	0%	~4.3%	~3.8%
1024	Our	~46	~1.7	~97%	~17%	~3.4%	~4.6%	~0%
	Naive	~130	~0.2	~93%	~15%	~0	~4.6%	~3.7%

From the above table, it can be seen that the throughput requested by the Naive implementation increases a lot in comparison to our code, showing reuse opportunities in our code and thus less latency for our code. Due to use of shared memory the stall due to syncthreads is present in our code but not naive; however this is offset by the memory dependency stall on naive code. The SM efficiency that is achieved by our code is higher than naive implementation that gives us the performance boost.

Analysis

4.a **Plot graph for different N**



4.b **Explain with respect to BLAS**

Comparison between BLAS and our code shows the following things.

- BLAS starts with very low performance for sizes 511 and lower, whereas our implementation gives higher performance at lower sizes.
- Our code has lots of dips near powers of 2, whereas BLAS has similar or very low difference among its neighbor sizes
- The dips in performance between one higher size than power of 2 is more in our code but not so much visible in BLAS
- Our code seems to saturate around 768 and above sizes that are close to power of 2, but BLAS doesn't saturate so much yet.

4.c **Identify irregularities, dips, peaks from 4.a**

It is visible from the figure that our code has performance dips for size which is one greater than the nearest power of two. This is attributed to the fact that we rely on BLOCKTILEs which are 96x64 for us. So a single element still requires computation from the whole tile even though it is padded with 0s. As we slowly go towards the power of 2, the performance increases.

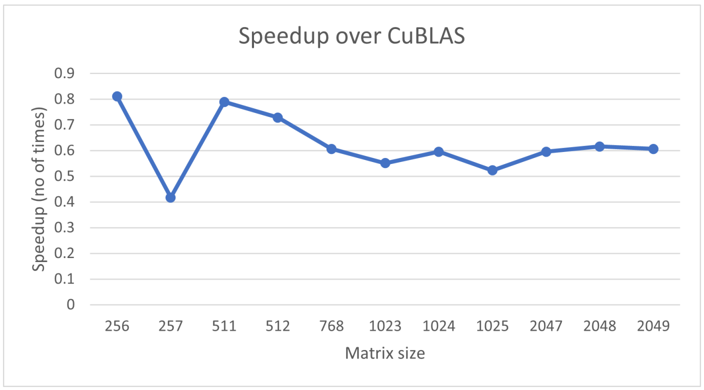
It is also visible that performance is saturating over sizes 1024 and above. This is possible due to the fact that at higher sizes we do not have sufficient occupancy due to hardware resource limitation that can help with throughput . This is in contrast with BLAS which does not saturate at higher matrices, showing there is still scope of reuse of matrices among cache hierarchy

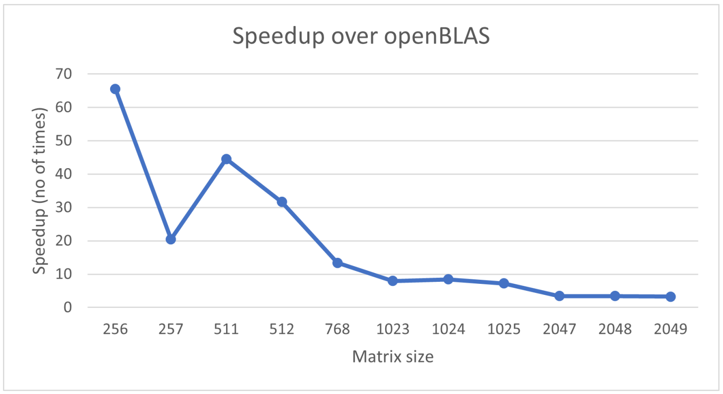
4.d **Show Table**

Matrix Size	BLAS Perf	CuBLAS	CUTLASS Perf GFLOPs	Reps

32			3.84	25000(4.26s)
33			2.83	19000(4.82s)
63			17.93	15000(4.18s)
64			20.19	17000(4.41s)
65			15.84	12000(4.16s)
127			87.41	10000(4.68s)
128			94.99	10000(4.41s)
129			82.55	8000(4.16s)
255			364	5000(4.55)
256	5.84	472.5	383	5000(4.36s)
257	(Interpolated in graph)	(Interpolated in graph)	237	3000(4.29s)
511	(Interpolated in graph)	(Interpolated in graph)	515	900(4.65s)
512	17.4	756.5	551	1000(4.87s)
768	45.3	1002.7	608	300(4.46s)
1023	73.7	1063.7	586	120(4.38s)
1024	73.6	1045.4	623	120(4.13s)
1025	73.5	1014.2	530	120(4.86s)
2047	171	984.2	586	15(4.38s)
2048	182	1021.6	629	15(4.09s)
2049	175	937.8	569	15(4.52s)

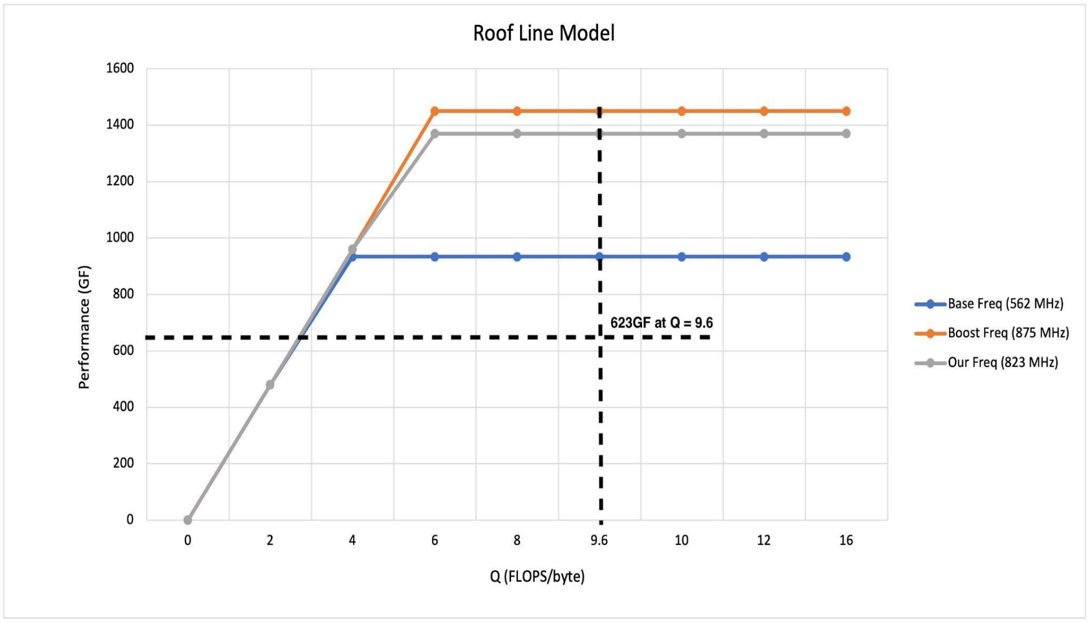
4.e **Plot in comparison with OpenBLAS and CuBLAS**





Calculation

5.a **Roofline model plot**



The thread block size for our best performance is 16X16. Each thread block calculates a C tile of size 96 X 64. So the total number of operations (multiply and add) for C would be = 1024 * 2.

Total amount of Global memory accessed = (96*1024 + 64*1024) * (8 bytes)

FLOPS = 2 * 1024 * 96 * 64. Therefore Q (Flops / Mop) = FLOPS/ Total Mem = 9.6

At Base Freq = 562MHz, Perf = 13 SM * 64 DP * 2 Ops * 562 = 934 GF/sec

At Boost Freq = 875 MHz, Perf = 13 SM * 64 DP * 2 Ops * 875 = 1.45 TF/sec

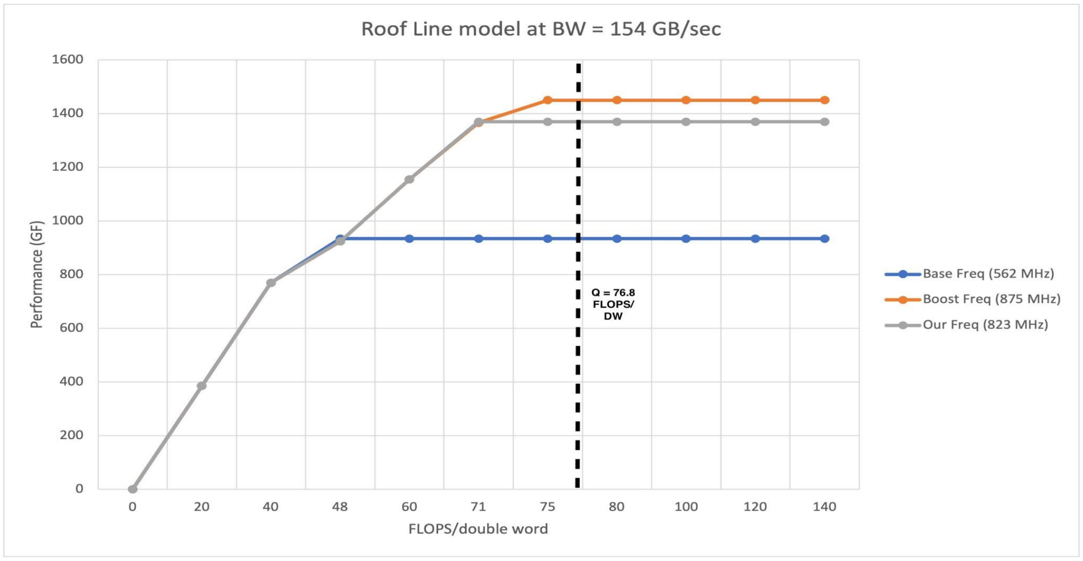
We are getting a Frequency of 823.5 MHz when running simulations.

At Our Freq = 823.5 MHz, Perf = 13 SM * 64 DP * 2 Ops * 823.5 = 1.37 TF/sec

At the given BW of 240 GB/s, peak performance for our Q is = BW * Q = 240 * 9.6 = 2.3 TF/s

This value is greater than the peak performance achievable at the base, boost and average observed frequency, so we should be able to reach the **max performance** of **1.37 TF/sec** at 823.5 MHz.

5.b **Estimate Q**



Peak performance at new BW = BW * q = 154 * 9.6 = 1.47 TF/sec. So a **Q of 76.8 FLOPS/double word** that we already have can help achieve peak performance at new BW.

Calculating **minimum Q value** required for peak performance at given BW for different frequencies:

At 562 MHz, $\mathbf{Q_{min}}$ = 934 / 154 = 6.07 FLOPS/byte = **48.51 FLOPS/double word**

At 823.5 MHz, $\mathbf{Q_{min}}$ = 1.37TF/ 154 = 8.89 FLOPS/byte = **71.11 FLOPS/double word**

At 875 MHz, $\mathbf{Q_{min}}$ = 1.45 / 154 = 9. 41 FLOPS/byte = **75.32 FLOPS/double word**

Q is already more than the minimum required Q so no need to change for new bandwidth.

Future Work

- Try different block dimensions, block tiles for CUTLASS because we didn't get enough time to optimize kernels. We got a good performance that will get us some extra credits is where we started writing our report.
- Try with rectangular thread block dimensions and tile sizes.
- Try Shuffle operation in interleaving case. It was brought to our attention that shuffle instruction helps in interleaving cases but we weren't very sure on how to use it.

T4 extra credit

7.a **Explain implementation**

- We first setup the instance on AWS for T4 using AMI cse260-sp22-hw2 keypair name i-0a4bec6f282a8eac2 (pa2-m3shah)
- Next, we tried a naive implementation using single point precision and noted performance.
- Next, we ported the interleaving access code to T4. Here we had to change the code a little since we wanted 32 SP accesses by a warp. So our Block dimension became 32*4. The performance wasn't crossing 2TFLOPs for sizes 2048 and above
- Next, we ported the CUTLASS code and got good results. In some cases it gave more than 3TFLOPs as well.

7.b **Show results**

N	Peak TFLOPs	Reps	Thread block size
256	0.71	10000 (~5s)	32 x 16
512	2.53	4000 (~5s)	32 x 16
1024	3.194	700 (~5s)	32 x 16
2048	3.62	100 (~5s)	32 x 16

References

1. CUTLASS blog : https://developer.nvidia.com/blog/cutlass-linear-algebra-cuda/
2. K80 Spec : https://www.techpowerup.com/gpu-specs/tesla-k80.c2616
3. T4 specs : https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-t4/t4-tensor-core-datasheet-951643.pdf
4. CUDA Programming guide : https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
5. Volkov talk sides: https://www.nvidia.com/content/GTC-2010/pdfs/2238_GTC2010.pdf