PA2 Report

Section (1) - Development Flow

Q1.a) Describe how your program works (pseudo code is fine, do not include all of your code in the write-up). Small snippets of code are fine as long as they help understanding. Be sure to include a description of how your program deals with edge cases (e.g. when N does not divide evenly into a natural block size in your program).

First we copy from host memory on CPU to device memory (GPU). Then the CPU invokes functions for the kernel grid containing all the blocks of threads. The grid is divided into (math.ceil(N/TW)) x (math.ceil(N/TL)) number of TW x TL thread blocks. Blocks are distributed to streaming multiprocessors (SM). Each SM will handle multiple blocks in parallel. Firstly we initialize 4x4 accumulators to be all 0s. Within each block, we loop through K dimension, for each TK, we load [TL][TK] of A (tile A) and and [TK][TW] of B (tile B) into shared memory of SM (SMEM). If N is not divisible by TL or TK, we pad tile A and tile B by zeros where there is no data from the original matrices. We wait for all the threads in this block to finish the loading before proceeding. Then we loop through 0 to TK, for each k, we store 4 elements of A and 4 elements of B from SMEM into the registers. Each streaming processor (SP) concurrently handles one thread from a thread block. For each thread, the 4x4 accumulators compute the outer product of fragment A and fragment. After we have traversed all k for that thread (sync threads), we load the accumulator result back into matrix C which resides in the global memory. If N is not divisible by TL or TW, when loading the result of accumulators to C, we only load those that are in the boundary.

Q1.b) What was your development process? What ideas did you try during development?

Firstly, we implemented loading squared tile A and tile B into shared memory in a coalesced way. Each thread, we load *A* [(TL*by+ty)*N + (k*TK + tx)]. For *B*, we load *B* [(k*TK+ty)*m + (TW*bx+tx)] . Adjacent threads in a warp have adjacent tx values so we have coalesced access to A and *B*. This bumps our peak performance to 200 Gflops/sec for N=256, and over 400 Glops/sec for N >=512. The performance can still be improved.

Secondly, each time we load 4 elements of A and 4 elements of B from shared memory. Each thread now computes the fragment of A and fragment of B instead of only one multiplication operation. This creates more work for each thread to do before syncthreads, and decreases the amount of stall for syncthreads. We further increased the speed by utilizing locality. While still having tile A and tile B in shared memory, we loaded fragment A and fragment B into the

register so we can perform faster retrieval when doing the outer product. Now for squared thread blocks of 16 x 16, we are able to achieve 360+ for N=256 and over 550 for N >= 512. This doesn't handle edge cases.

Thirdly, we utilized ILP (instruction level parallelism) by using "unroll" operation to concurrently execute parallelizable operation when Areg[i] doesn't depend on any other values in Areg. Utilizing register and parallelism increased our performance from 200 Gflop/sec to 360 Gflop/sec for N = 256.

To cover the edge cases, we implemented padding code, and made adjustments to loop variables to handle non-squared block sizes and TK.
Lastly, we tuned the parameters according to the hardware layout for optimal performance.

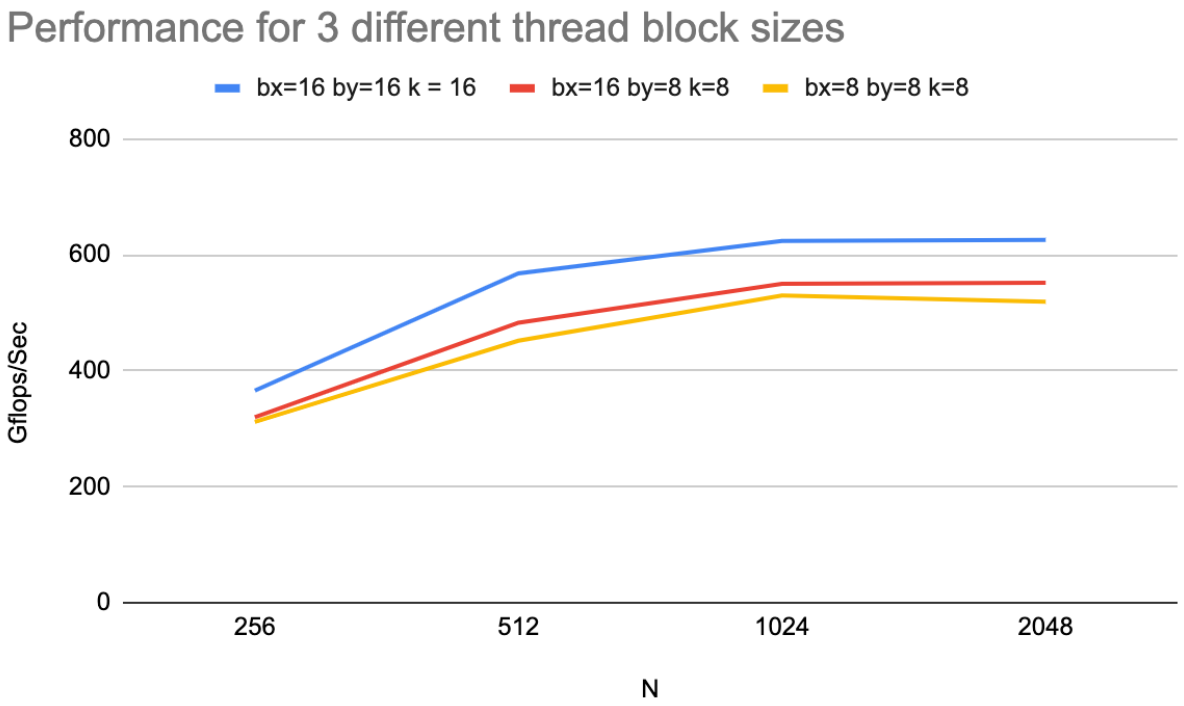
Q1.c) What ideas worked well, what didn't work well, and why. Feel free to plot or chart results from experiments that did or did not end up in your final implementation and, as possible, provide evidence to support your theories.

One of the things we tried was using length non divisible by 16 when loading A and B to shared memory to avoid memory bank conflicts. For example, instead of loading [TL][TK] of A into shared memory, we loaded [TL][TK+1] of A because TK is set to 16. When the shared memory As has a width of 16, accessing As[j*TL+ty][k] will create a memory conflict because for each instruction of j = 0,1,2,3, each will fetch data from bank k, and thus the loop of j will be serialized instead of parallelized. However, it only improves performance with smaller blockDim. In fact, it even hinders performance for N=1024. The reasoning is that for larger blockDim. Because when the arrays are longer, now every time loading from SMEM to the register file takes a longer time and requires non-coalesced access to shared memory.

Section (2) - Result

Result Your implementation will be graded on its performance at the following matrix sizes: n=256, n=512, n=1024, n=2048

Q2.a) For the problem sizes n=256, 512, 1024 and 2048, plot the performance of your code for a few different (at least 3) different thread block sizes.



These thread block sizes may map to different tile sizes. Please mention the relationship between thread block sizes and tile size in your report. If your code has limitations on thread block size, please state the reason for that limitation.

The tile size of A and B is determined by the thread block sizes and how much computation we want to do on a single thread. The thread block sizes TW and TL determine how many threads each block contains. If the block sizes are 16 x 16, then each block can contain 16 x 16 threads. The tile size of A and B is [TL][TK] and [TK][TW]. However, in our implementation, each thread computes a 4x4 matrix. So the actual amount of tile size we require needs to quadruple. Within each block, we allocate shared memory of [4*TW][TK] of A and [4*TK][TW] of B. Our code has a limitation of 20 x 19 because of limitations in shared memory. The shared memory size is 48KB for each block. 20 x 19 x 4 x 2 x 8byte does not equate 48KB.

Q2.b) Your report should explain the choice of optimal thread block sizes for each N(matrix size - 256, 512, 1024, 2048). Why are some sizes or geometries higher performance than others?

For all of the matrix sizes, 16 x 16 performs the best. This geometric shape performs the best because it creates less number of blocks. It can fit the most amount of A and B into shared memory, so it requires less coalesced global memory reads. N is divisible 16 so it doesn't require zero padding and requires less overhead.

Q2.c) Mention the peak GF achieved and the corresponding thread block size for each matrix size analyzed in the previous question in a table like this.

N	Peak GF	Thread Block Size
256	366	16*16
512	568	16*16
1024	624	16*16
2048	626	16*16

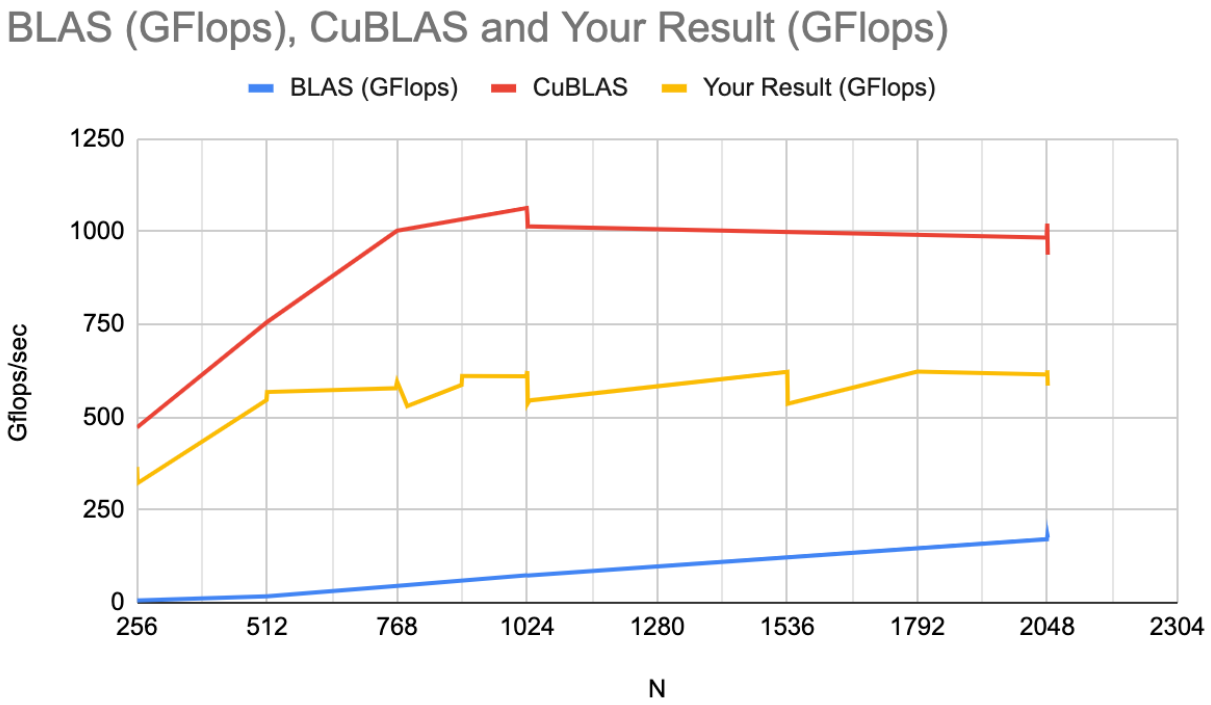
Section (3)

Q3.a) For n=256, 512, 1024, and 2048 quantitatively compare your best result with the naive implementation.

As we can see in the line chart in Q4.a, our implementation has peaks in performance when it is the multiple of 64. Because that is the size of thread size (4) times block size (16). In the naive implementation. For naive implementation, each thread is solely responsible for one entry of C. So even though all the threads execute in parallel, the amount of time to calculate C would be N^3. When N = 256, our code's performance is 72 times greater than naive implementation. When N = 512, our code's peak performance is only 14 times greater than the naive implementation. 32 times greater than naive implementation. When N = 1024, our code's performance is When N = 2048, our code's peak performance is only 3 times greater than naive implementation (table in Q4.c). The high difference of peak performance when N is small is due to the limited bandwidth for global memory reads and low Arithmetic intensity. As N increases, the performance of our implementation is also limited by the processor speed and memory bandwidth.

Section (4) - Analysis

Q4.a) For at least twenty values of N within range (256 - 2049) inclusive, plot your performance using the best block size you determined for n=1024 in step (2). Use at least the values in the table below, but add other values too. Compare your results to the multi-core BLAS results in the table below. (for the values we gave you in the table. For other values, just report your cuda numbers).



Q4.b) Explain how the shape of your curve is different or the same to the BLAS values and theorize as to why that might be. You may refer to either the plot from Q4a or the plot of speedup S (Q4e).

The shape of our curve is similar to that of BLAS values. The Gflops increases with the size of the matrix until reaching a plateau. This can be explained by the compute-to-memory ratio. When the matrices are small, memory access is the bottleneck. Few computations are needed to generate the results, while loading matrices into shared memory and registers takes up a large portion of overall time, leading to low compute-to-memory ratio. As the matrix becomes larger, the compute-to-memory ratio increases and the GFlops increases as a result. After the matrix size reaches a certain number(512 for our implementation and 768 for CuBLAS), memory access is no longer the bottleneck, the curve reaches a plateau and the Gflops remains stable.

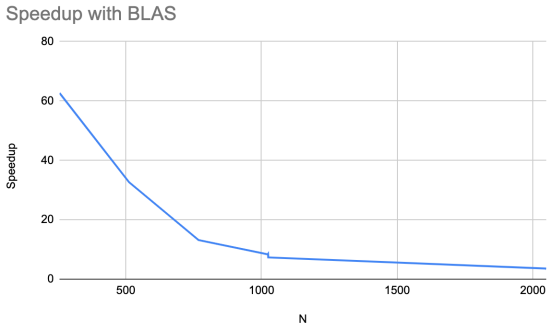
Q4.c) For the twenty or so values of performance, identify and explain unusual dips, peaks or irregularities in performance with varying n.

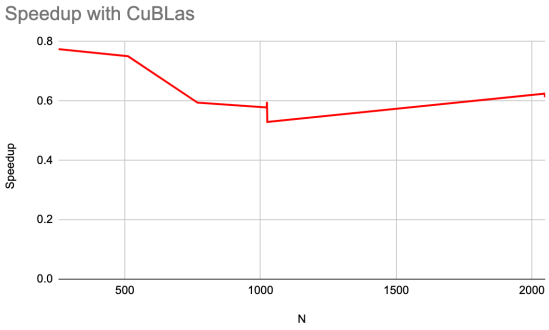
We observe a sharp drop of performance at N=788 compared to N =768, N = 1025 compared to 1024 and N = 2049 compared to N = 2048. This is because we pad matrices whose size is not divisible by the thread block size. For the cases mentioned above, we have to pad whole blocks and perform many unnecessary multiplications of zero and zero for computing only one extra value. This significantly decreases the performance.

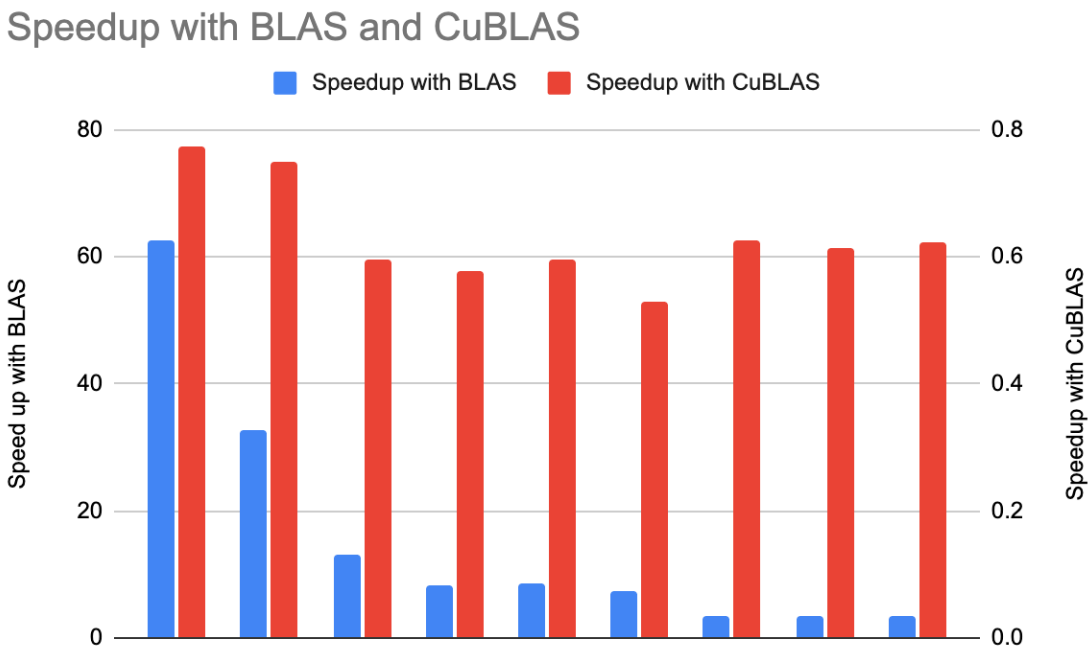
Q4.d) Please include the following table in this format in your report as well as a graph mentioned above (Q4a): (you must have these n values but you should have more (20)).

N	BLAS (GFlops)	CuBLAS	Your Result (GFlops)
256	5.84	472.5	366
257			322
511			546
512	17.4	756.5	568
765			578
768	45.3	1002.7	596
788			530
895			587
896			611
1022			610
1023	73.7	1063.7	615
1024	73.6	1045.4	624
1025	73.5	1014.2	537
1028			544
1029			545
1536			622
1537			536
1792			623
2047	171	984.2	615
2048	182	1021.6	626
2049	175	937.8	585

Q4.e) Plot the above results comparing your implementation with OpenBLAS and cuBLAS (from table).

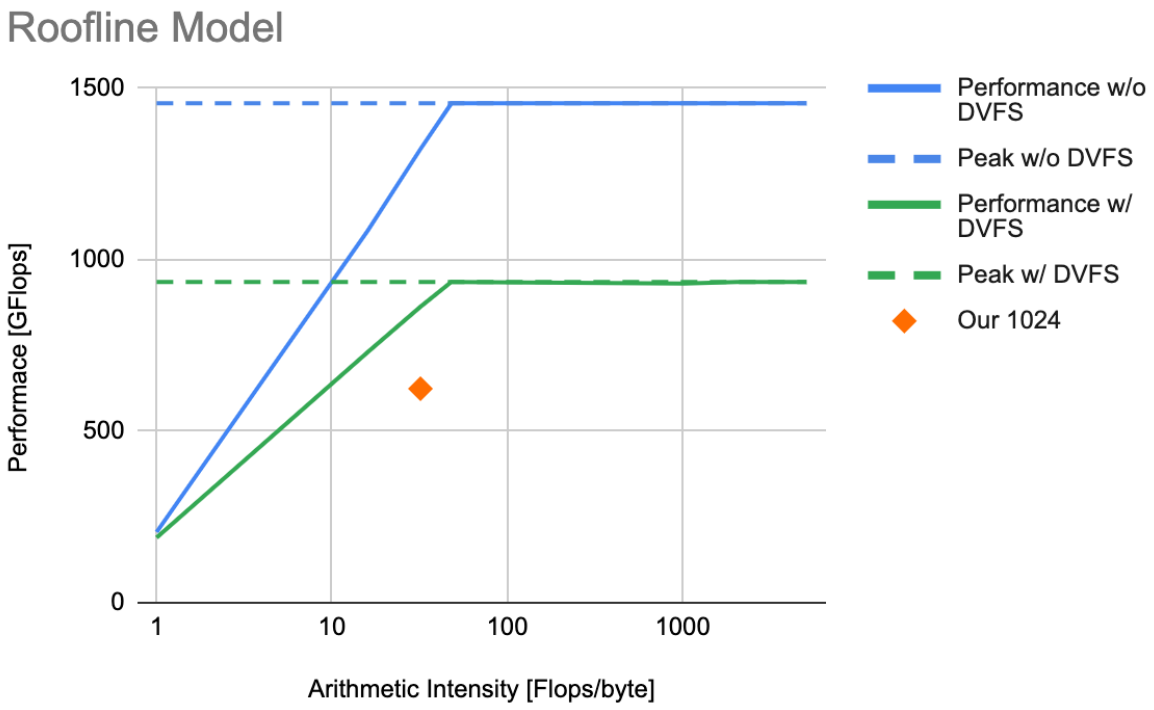

Speedup with BLAS
80
60
40
20
0
Speedup
500
1000
1500
2000
N


Speedup with CuBLas
0.8
0.6
0.4
0.2
0.0
Speedup
500
1000
1500
2000
N


Speedup with BLAS and CuBLAS
Speedup with BLAS
Speedup with CuBLAS
80
60
40
20
0
Speed up with BLAS
0.8
0.6
0.4
0.2
0.0
Speedup with CuBLAS

Section (5)

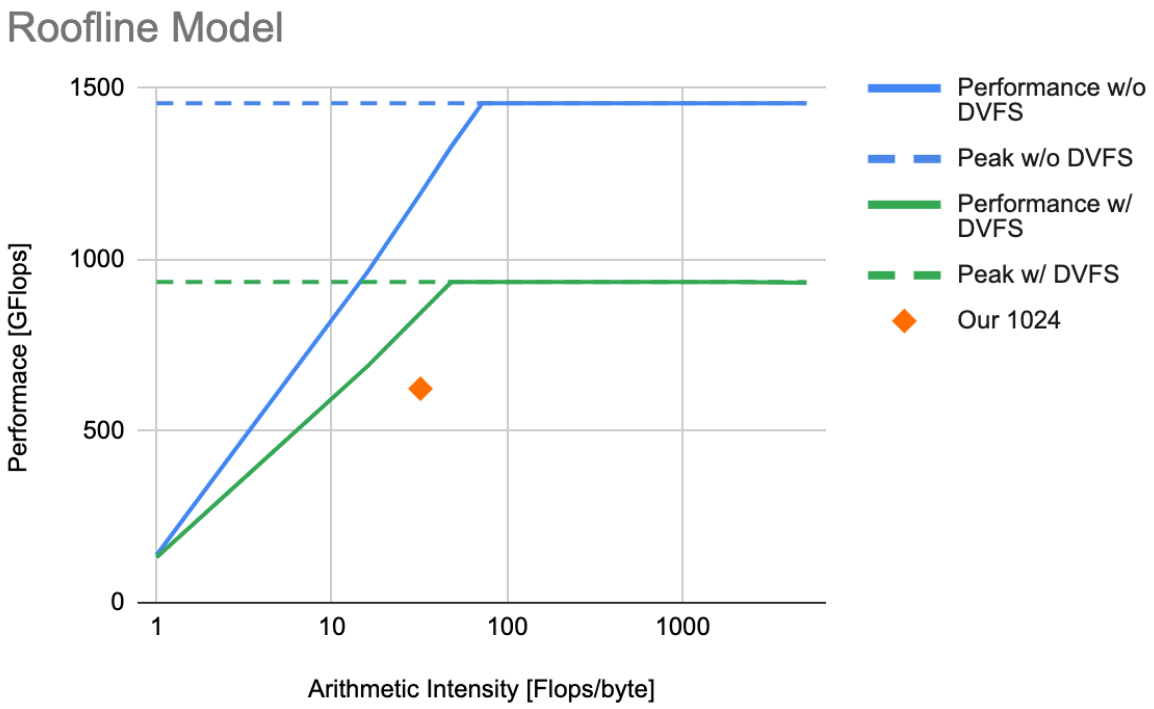
Q5.a) Using the raw bandwidth of 240GB/sec to global memory(theoretical max), plot a roofline model (log-log) or (lin-lin) for the GPU and plot your achieved n=1024 number on this plot. Calculate the peak performance of the roofline plot and explain how you arrive at the peak. Take into consideration DVFS effects when calculating the peak performance for n=1024.



Peak performance without DVFS:
The K80 has 13 SM, with boost frequency 875 MHz. Each SM has 64 DP cores with each performing 1 FMA(2flops) per cyle. Thus the theoretical peak performance is: 2 * 64 * 13 * 0.875M = 1456 GFlops.
Peak performance with DVFS:
Taking DVFS into consideration, the frequency should be 562 MHz, and thus the peak performance is: 2 * 64 * 13 * 0.562M = 935 GFlops.

Q5.b) Estimate the value of q in ops/doubleword. Consider that the actual BW is less than 240GB/sec - Volkov thesis measures 154 GB/sec, plot this roofline and calculate the new "q" value. How has the value of q been affected by the change in BW?

Q can be estimated with peak performance and BW as: q = 1456G / 240G = 6.06 ops/word



The new q is: q = 1456 / 154G = 9.45 ops/word. The decrease in BW leads to larger q.

Section(6) - Potential Future work

What ideas did you have that you did not have a chance to try?

The performance for N=256 is always very slow compared to other matrix sizes with the same thread block sizes and thread tile sizes. CUBLAS is designed to optimize large, batched GEMMs. However, the same algorithm might not be suitable for smaller matrix sizes. The ideas should be similar to large dimension multiplication in terms of optimizing locality and parallelism. Possible implementation would be only loading B into registers to and compute C and reuse B at each iteration [6].

In addition, we did not implement shuffle which “allows the threads of a warp to exchange data with each other directly without going through shared (or global) memory”[7]. Allowing threads to share data with other threads without going through shared memory of the warp should theoretically improve performance.

Section(7) - T4 Extra credits

Q7. For single precision matrix multiplication kernel extra credit, please explain your implementation and results. N Peak TF Thread Block Size 256 512 1024 2048

Section (8) - References

(cite all references used)

[1] "*Cutlass: Fast Linear Algebra in Cuda C++.*" NVIDIA Technical Blog, 25 Aug. 2020, https://developer.nvidia.com/blog/cutlass-linear-algebra-cuda/.

[2] Volkov, Vasily. 2010. "*Better Performance at Lower Occupancy*." https://www.nvidia.com/content/GTC-2010/pdfs/2238_GTC2010.pdf.

[3] "*5KK73 GPU Assignment Website 2014/2015*." n.d. Www.es.ele.tue.nl. Accessed May 18, 2022. https://www.es.ele.tue.nl/~mwijtvliet/5KK73/?page=mmcuda#TOC-Increasing-Computatin-to-Memory-Ratio-by-Tiling

[4] Volkov, V., and J.W. Demmel. "*Benchmarking Gpus to Tune Dense Linear Algebra.*" 2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis, 2008, https://doi.org/10.1109/sc.2008.5214359.

[5] *Bla Final | Liferay Content - University of Illinois Urbana-Champaign*. https://bluewaters.ncsa.illinois.edu/liferay-content/image-gallery/content/BLA-final.

[6] Masliah, I., et al. "Algorithms and Optimization Techniques for High-Performance Matrix-Matrix Multiplications of Very Small Matrices." *Parallel Computing*, vol. 81, 2019, pp. 1–21., https://doi.org/10.1016/j.parco.2018.10.003.

[7] *Tuning Cuda Applications for Pascal - Rice University*. https://www.clear.rice.edu/comp422/resources/cuda/pdf/Pascal_Tuning_Guide.pdf.