cse260: HW 2 Report Template

Here is a general rubric. You may get credit for additional things not on the rubric (e.g. extra graphs, introduction, other stuff we haven't explicitly thought of so think of this as a general guide. Depth of analysis and clarity are key objectives.

Follow these guidelines to create a well-written and organized report. * Your report must follow the format given here so we can easily find the different sections. Reports that require extensive time for us to read will be penalized. (up to -10 pts if the report is not organized well).**

- **Full credit will be awarded only if these guidelines are followed. *Your report must clearly label the sections below. You may include additional sections (e.g Introduction) if you wish.***

****Document your work in a well-written, 5-10 page (including figures) report [the length is just a guide, if you need fewer pages that's fine as long as you report the required information]. More than 10 pages is probably overkill. (Reports that require extensive time for us to read will be penalized up to -10 pts if the report is not organized well). The report presents your results, analyzes them, and offers insights as to why things behaved the way they did. If you improved your code in a sequence of steps, document the process.****

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).

The matrix multiplication computation kernel is implemented as the matMul function in mmpy_kernel.cu file. The basic idea is to divide the result matrix C into tiles so that each tile is assigned to a thread block. Every thread block has a dimension of (BLOCKDIM_X, BLOCKDIM_Y) defined by users, and thus has bx times by threads in total. A thread is responsible for computing TILESCALE_M * TILESCALE_N elements out of the tile. From the relationship between thread blocks and tiles, we have the dimensions of tiles as TILEDIM_N = BLOCKDIM_X * TILESCALE_N and TILEDIM_M = BLOCKDIM_Y * TILESCALE_M.

In order to reduce the number of memory accesses, we utilized the shared memory within a thread block. To compute the result tile, a (TILEDIM_M, N) submatrix from input matrix A and a (N, TILEDIM_N) submatrix from input matrix B are needed. We further introduce another parameter TILEDIM_K to restrict the size of matrices loaded into shared memory. For each iteration, threads in a thread block together load a (TILEDIM_M, TILEDIM_K) submatrix from A and a corresponding (TILEDIM_K, TILEDIM_N) submatrix from B. As a result, each thread will need to load (TILEDIM_M * TILEDIM_K) / (BLOCKDIM_X * BLOCKDIM_Y) elements from A and (TILEDIM_K * TILEDIM_N) / (BLOCKDIM_X * BLOCKDIM_Y) elements from B to the shared memory.

After loading elements into shared memory, the program first makes sure every thread in the same block has done their loading by synchronizing. Then, it starts to compute the outer product of vectors from A and B stored in the shared memory. The values are accumulated over TILEDIM_K multiplications and another synchronization within the thread block is needed before the next iteration. By repeating the memory loading process from global memory to shared memory, the synchronization, the computation, and another synchronization, the result tile becomes completed. Finally, the result values are written into the global memory.

There are several tiny details in the program. First, TILEDIM_K is set to 32 in order to utilize the coalescing loading from the global memory to shared memory. Second, to handle the cases that the dimensions of input matrices N are not evenly divided by the block size, we check the index while loading A and B to shared memories and writing back result values to C. When the index is out of boundaries, then store value zero.

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

First, the utilization of shared memory is implemented. Then, a newer version of the program handles the case that N is not evenly divided by the block size. At this time, 2D tiling is not implemented yet. Each thread only loads one value and computes one element. The block dimension is also assumed to be the same as the tile dimension and TILEDIM_K. The performance has improved compared to the naive implementation.

Then, the 2D tiling is implemented in order to better utilize the computation capabilities of each thread. At the same time, the program is also able to handle more general cases in terms of different dimension sizes. The scheme of elements each thread is responsible for is also changed from an adjacent method to evenly divided. Finally, the loop unrolling is included to further improve the 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.

Comparing the three approaches: naive, shared memory without 2D tiling, and shared memory with 2D tiling, the performance has a really huge difference. You may see the results for N=2048 and block size = 32x32 as below:

Naive	Shared memory without 2D tiling	Shared memory with 4x4 tiling
576.9 GFLOPS	967.3 GFLOPS	3341.8 GFLOPS

It proves the design idea mentioned in the previous question. The utilization of shared memory reduces the amount of memory access from a slower global memory. The 2D tiling method on the other hand increases the ratio of computation to memory access.

As for the scheme of loading values from shared memory, we tried the adjacent scheme and the evenly distributed scheme. There is no huge difference between the two schemes. Our observation finds that the adjacent method has a higher average performance, while the later scheme yields a higher peak performance. These two schemes actually have different impacts on the memory bank usage. When adopting the adjacent scheme, the program is more stable but threads might be competing for the access for the same memory bank. On the other hand, when loading a row of data having a fixed distance from the previous one, the behavior is harder to predict, it might get better utilization of the available memory banks but might even worsen the situation of collision between threads. Last but not least, loop unrolling helps the performance to increase about 50 to a hundred GFLOPS across different N. Due to the parallelism nature of GPU, loop unrolling is a desired way to improve performance.

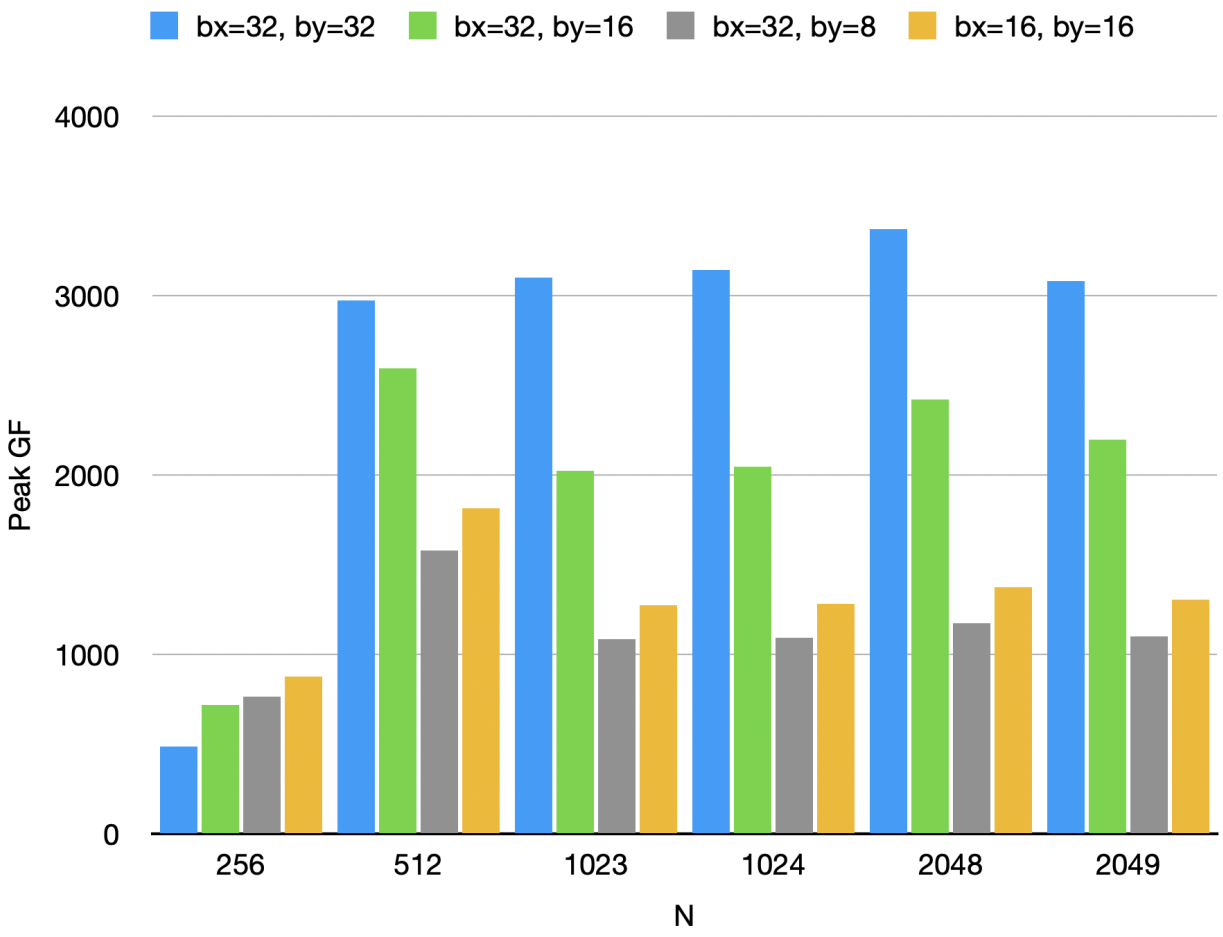
Section (2) - Result

Your implementation will be graded on its performance at the following matrix sizes: n=256, 512, 1023, 1024, 2048, and 2049.

Q2.a) For the given sizes n, 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.

When the 2D tiling scale is set to 4 by 4, we get the performance from the following chart. As mentioned in the answer to Q1.a, the relationship between thread block sizes and tile size is that the tile size is block size multiplied by the tile scale. The thread block size is limited by hardware resources.



Q2.b) Explain the choice of optimal thread block sizes for each N. Why do some sizes or geometries have higher performance than others?

For N>=512, a thread block size of 32 by 32 is preferred since larger block size reduces the amount of global memory access and thus improves performance. However, for N=256, there would be an under utilization of hardware resources. If the block size is 32 by 32, the tile dimension is 128 by 128, which means there are only 4 thread blocks resulting in a low GPU occupancy. Therefore, a 16 by 16 thread block is a better choice. From the results, we learned that there is a balance between increasing occupancy and utilizing the shared memories more efficiently.

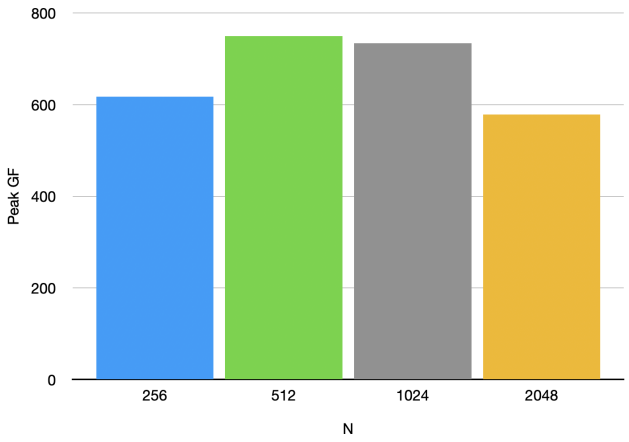
Q2.c) Document the peak GF achieved and the corresponding thread block size for each matrix size in the table.

N	Peak GF	Thread Block Size
256	878.5	16 by 16

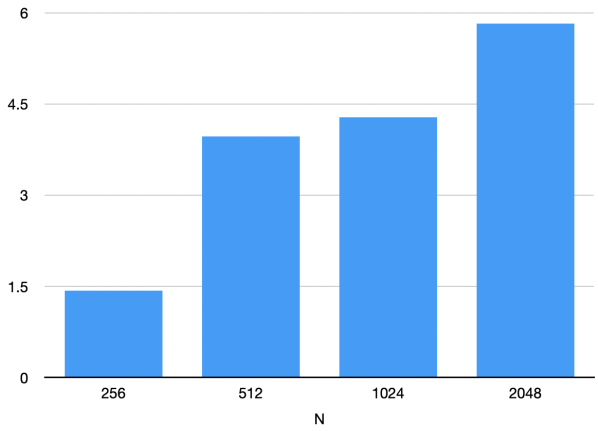
512	2972.7	32 by 32
1023	3103.2	32 by 32
1024	3144.8	32 by 32
2048	3370	32 by 32
2049	3082.3	32 by 32

Section (3) - Comparison to Naive

Q3.a) For n=256, 512, 1024, and 2048 graph your best result on each size with the naive implementation.



Q3.b) Graph the speedup for n=256, 512, 1024 and 2048 for your best result on each size versus naive.

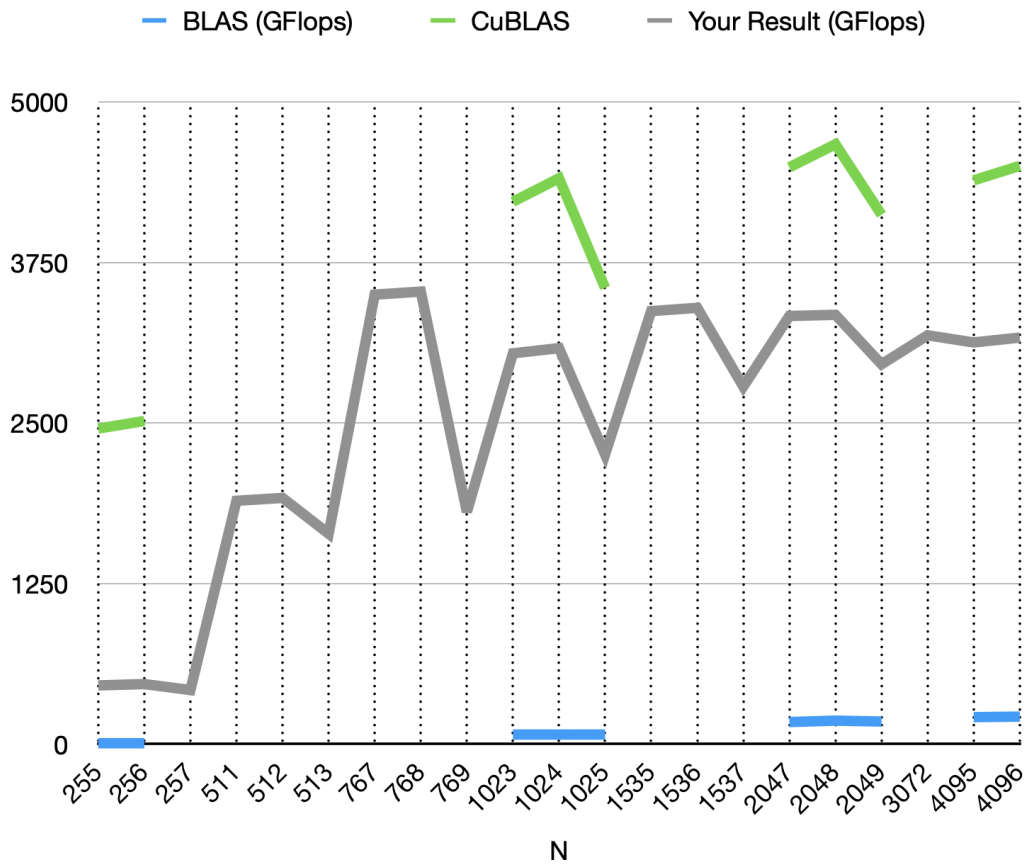


Section (4) - Analysis

Q4.a) For at least twenty values of N within range (256 - 2049) inclusive, graph 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(around 20 values total). Compare your results to the multi-core BLAS and cuBlas results in the table below. (for the values we gave you in the table. For other values, just report your cuda numbers.

N	BLAS (GFlops)	CuBLAS	Your Result (GFlops)
255	5.93	2456.2	456.2
256	5.84	2515.3	466.8
257			420.2
511			1893.9
512	17.4	4573.6	1916.1
513			1637.5
767			3499.2
768	45.3	4333.1	3522.7
769			1807.1
1023	73.7	4222.5	3042.0
1024	73.6	4404.9	3082.3
1025	73.5	3551.0	2254.8
1535			3370.6
1536			3396.3
1537			2782.8
2047	171	4490.5	3332.9
2048	182	4669.8	3341.8

2049	175	4120.7	2957.0
3072			3182.5
4095	209.2	4389.0	3126.7
4096	213.4	4501.6	3164.8



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 the plot from Q4a.

The trend of the three curves are similar. Generally, larger N gets better performance. This is because larger matrix sizes usually have a higher hardware utilization rate. The peaks at Ns that are evenly divided by block size is also the case both in the results we get and the CuBLAS. This results from the fact that there is less resource waste in those cases. We also observe that N-1 gets better performance than N+1 does, where N is the multiple of block size. This observation also supports the reasoning since the resource waste on N-1 is second to N, while N+1 has the most waste.

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

As mentioned previously, the resource wastes including computation and hardware are huge when the matrix size is not a multiple of the block size. Take N=1025 as an example, when the tile dimension is 128, 17 additional thread blocks are required and only one out of the 32 rows/columns is used. Many of the operations are done on the padding zeros, which is not related to the results.

Section (5)

Q5.a)

Zhe Jia, Marco Maggioni, Jeffrey Smith, Daniele Paolo Scarpazza , "Dissecting the NVidia Turing T4 GPU via Microbenchmarking((https://arxiv.org/pdf/1903.07486.pdf) specifics that this GPU has a maximum memory bandwidth of 320 GB/sec and an actual bandwidth of 220 GiB/sec. Using the 320 GiB/sec figure, plot a roofline model (log-log) or (lin-lin) for the GPU and plot what you achieved for the n=2048 number on this plot.

Assume that the T4 GPU has 40 SMs and each SM has 64 SP FP cores that can do one FPMAD/cycle. Assume the GPU runs at 1.5GHz and each core can do 2 ops (1 multiply and 1 add per cycle).

Calculate the peak performance of the roofline plot and explain how you arrive at the peak.

Given:

- 40 SMs
- Each SM has 64 SP FP cores
- Each core does 2 ops per cycle (FPMAD: 1 multiply + 1 add)
- Clock frequency: 1.5 GHz
- 320 GiB/sec = 85.9 GWord/sec

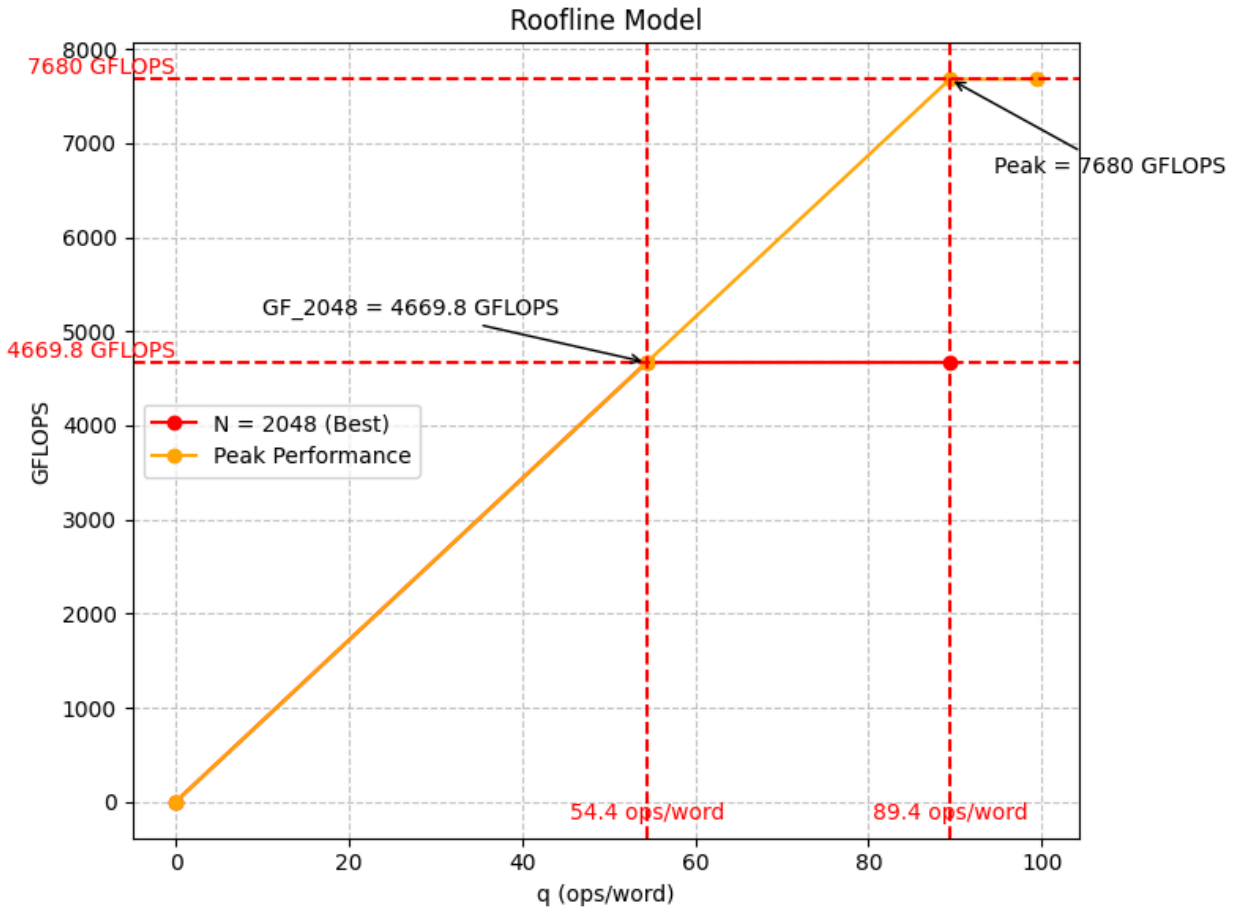
1. Total cores = 40 × 64 = 2560 cores
2. Flops/core = 2 × 1.5 GHz = 3 GFlops/core
3. Peak = 2560 × 3 = **7680 GFlops**
4. GF_2048 = 4669.8 / 85.9 = 54.4 ops/word

Q5.b) Estimate the value of q in ops/word (single precision float). Consider that the actual BW is less than 320GB/sec - Jia, etal say it is 220 GiB/sec. , Using this smaller BW, plot this roofline and estimate the new "q" value. How has the value of q been affected by the change in BW?

320 GiB/sec = 85.9 GWord/sec

q = 7680/85.9 = **89.4 ops/word**

Plot for 5a and 5b:



Section(6) - Potential Future work

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

If we have more time, we would like to try a more specially designed program for specific input matrix size to get a better idea of the maximum performance. Also, the memory access pattern is something that could be optimized further. Since we didn't dig into the memory hardware design and more CUDA features too much. There must be room for improvement.

Section(7) - Extra credits

Create a non-square matrix multiplication kernel that should be able to take 2 matrices A and B of dimensions MxK and KxN respectively and compute C of MxN. (4 points). What optimizations can you make with this kernel that would improve performance as compared to a simple square matrix multiplication? (2 points)

Section (7) - References (cite all references used)

- Course Lecture Materials
- Slides presented by TAs in discussion sessions