PA2 Report

Section 1: Development Flow

1a: Program Workflow

Our code completes matrix-multiplication with the following steps:

a. src_todo_T4/OPTIONS.txt file defines the bx and by variables which represent the x and y dimensions of a thread block respectively.

b. src/mmpy.cu file invokes setGrid() function defined in the src_todo_T4/setGrid.cu file to set number of blocks along x dimension to be equal to ceil(N/TILEDIM_N) and number of blocks along y dimensions to be equal to ceil(N/TILEDIM_M) where N is the matrix size.
   i. The TILEDIM_M and TILEDIM_N variable values are defined in the src_todo_T4/mytypes.h file.
   ii. In our code, we have set bx=32 by=8 and we want a given thread to do 2 operations along x-direction and 16 such operations along y-direction. Hence, we have chosen the value of TILEDIM_N=64 and TILEDIM_M=128.
   iii. Considering our thread block geometry, and since we want a thread to do 2 and 16 operations in the x-direction and y-directions respectively, we need to set TILEDIM_K=64.

c. Code in src/mmpy.cu copies matrices fromhost memory to device (GPU) memory and issues matMul() kernel having 64KB shared memory per thread block.

d. Matrix multiplication kernel defined in src/mmpy_kernel.cu is invoked.
   i. The code in matMul() kernel is responsible for the computation of elements in the TILEDIM_M x TILEDIM_N tile of the resultant C matrix. One thread block is responsible for one TILEDIM_M x TILEDIM_N tile computation, and multiple such thread blocks work together in parallel to produce the final C matrix.
   ii. The matrix multiplication kernel has an outer for loop that chooses a row of matrix A and a column of matrix B. Iterations of this outer for loop perform multiplication of TILEDIM_M x TILEDIM_K tile of A and TILEDIM_K x TILEDIM_N tile of B along the selected row and column.
   iii. In each iteration, the elements of TILEDIM_M x TILEDIM_K tile of A and TILEDIM_K x TILEDIM_N tile of B are brought from the global memory of GPU to the shared memory within a thread block. As and Bs represent tiles of A and B respectively residing in the shared memory.
   iv. Each iteration consists of an inner for loop to multiply each row of As with the corresponding column of Bs. Threads within a block cooperatively do this computation.
   v. Each thread is responsible for computing 32 resultant elements of the C matrix. It computes two elements along x-direction at an offset of 32 and performs these two operations 16 times along y-direction at an offset of 8.
   vi. At the end of the aforementioned two nested for loops, resultant elements of TILEDIM_M x TILEDIM_N tile of C are written to global memory.

e. Finally, src/mmpy.cu copies the resultant C matrix from the device memory to the host memory.

f. TILEDIM_M and TILEDIM_N are chosen such that they are divisible by block dimensions by and bx respectively. To handle the case where matrix size N is not divisible by TILEDIM_M and TILEDIM_N:
   i. Before writing to As or Bs, we ensure that the index that we are trying to access in matrix A or B is less than N*N and that the index is less than the first element in the next row. We write the value for an index in As or Bs to be zero if any one of these two conditions are not satisfied.
   ii. Before writing the final result to an element in matrix C, we ensure that the index that we are trying to write to in matrix C is less than N*N and that the index is less than the first element in the next row. We do not perform a write operation to matrix C if any one of these conditions are not satisfied.

1b and 1c: Development Process

We have tried four optimization ideas to achieve maximum possible performance enhancement. The first three optimization ideas gave desired results while the last optimization idea did not give desired results. We have described all of our optimization ideas below.

Optimization 1: Tiling matrices to use shared memory and maximising computations performed per thread

Methodology and Reasoning:

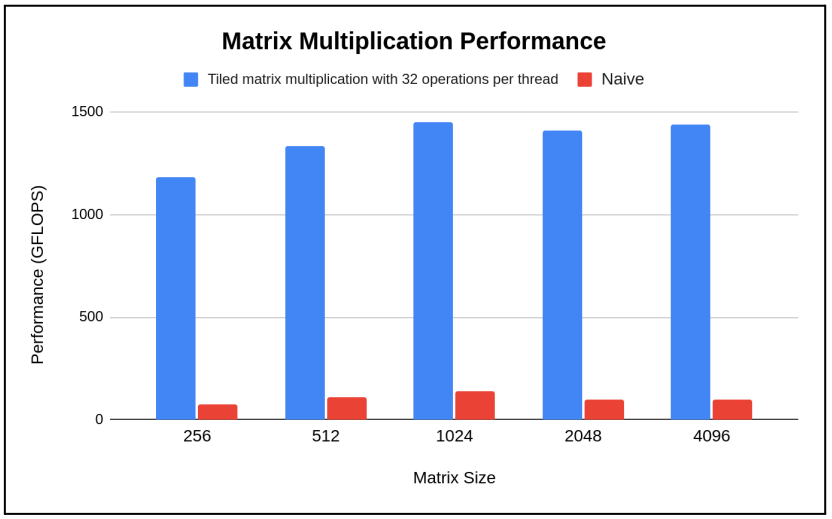
In the naive approach, to compute a single element of C, a thread reads the entire row and column of A and B respectively from global memory. Each thread is responsible for computing one element of C and all threads redundantly perform this memory access operation. Thus, there are 2N (reading of row and column of A and B respectively) reads done to compute one element of C, and to compute N^2 elements of C, there will be $2N^3$ reads from global memory.

The total number of global memory reads are reduced by tiling matrices. Threads in a block cooperatively bring TILEDIM_M x TILEDIM_K sized tile of A and TILEDIM_K x TILEDIM_N sized tile of B into shared memory and then compute TILEDIM_M x TILEDIM_N elements of C. Thus, elements required by a thread might have already been brought to shared memory by another thread. Through this tiling approach, instead of 2N reads in the case of the naive approach, there will be approximately 2N/TILEDIM_K reads from global memory to compute one element of C. Moreover, the total reads from global memory to compute entire matrix C also reduces by a factor TILEDIM_M and TILEDIM_N respectively. Since shared memory access latency is less than global memory's, the reduction in global memory reads helps achieve performance enhancement.

In addition to tiling, we let each thread be responsible for computation of 32 elements of matrix C. This is inspired from the results of Volkov's studies [1]. The study reveals that with small occupancy we can achieve close to peak performance by increasing the instruction level parallelism. This helps hide the arithmetic and memory access latency. With this reasoning, we choose to do 32 outputs per thread. We tried to do more outputs per thread but were limited by the amount of shared memory available as described later in the report.

Result and Conclusion:

The plot below shows the matrix multiplication performance achieved with the following configuration: bx=8,by=8,TILEDIM_M=64,TILEDIM_N=32,TILEDIM_K=32.
Tiling while using shared memory and performing 32 outputs per thread resulted in 9x to 15x improvement in matrix multiplication performance for different matrix sizes. We applied the same optimization to different thread block geometries described in Section 2.



Optimization 2: Taking advantage of memory coalescing

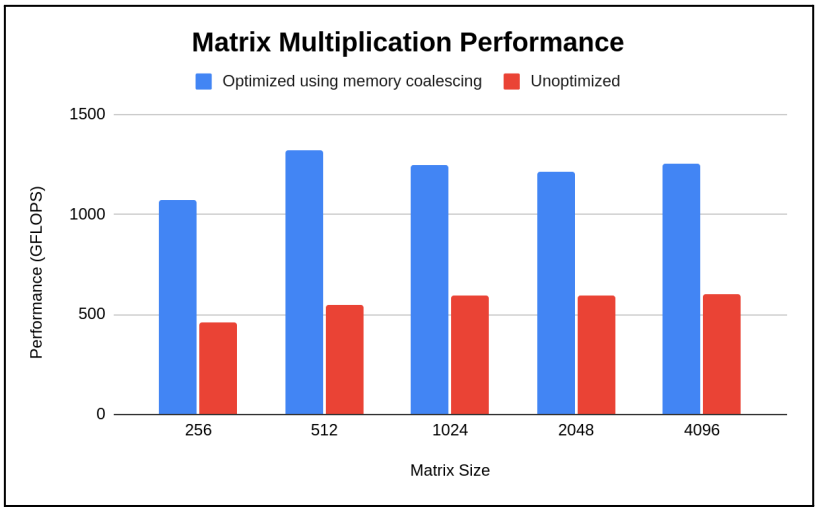
Methodology and Reasoning:

Every successive 128 byte chunk of GPU global memory could be accessed in a single memory transaction [2] by a warp. Hence, while loading values from global memory to As and Bs in shared memory, if a warp

accesses successive elements of A and B matrix, then it can result in significant reduction of global memory accesses. In our implementation, threads in a warp access sequentially stored elements of A and B while bringing data from global memory to shared memory. This helped us take advantage of memory coalescing and thereby improve performance.

Result and Conclusion:
The plot below shows the matrix multiplication performance achieved using the following configurations with and without taking advantage of memory coalescing:
bx=8,by=8,TILEDIM_M=64,TILEDIM_N=32,TILEDIM_K=32.

Memory coalescing helped us achieve **10 to 14 percent higher performance** over implementation that didn't take advantage of memory coalescing. We applied the same optimization to different thread block geometries described in Section 2.



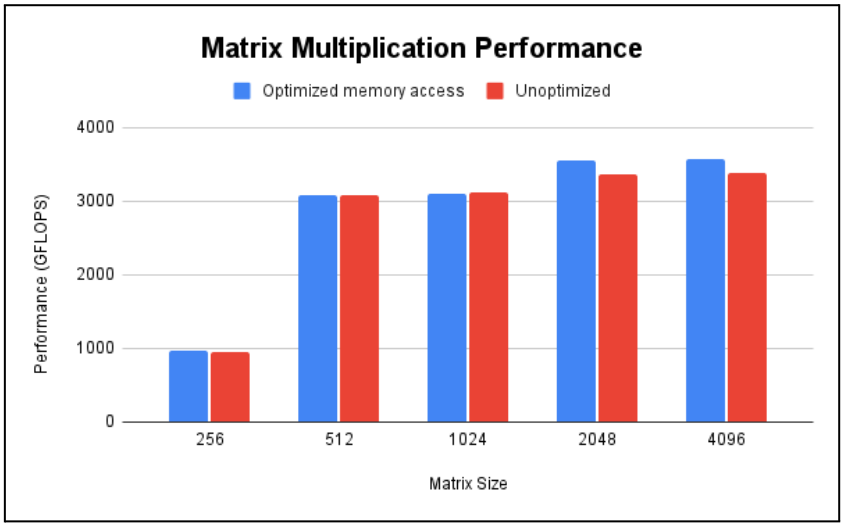
Optimization 3: Optimising shared memory access pattern and using registers to reduce the number of shared memory accesses
Methodology and Reasoning:
In our implementation with configuration bx=32,by=8, TILEDIM_M=128, TILEDIM_N=64, TILEDIM_K=64, we compute 32 outputs per thread. We observed that we are accessing a single element of matrix B from shared memory 16 times. We store this element in a single register and reuse this register whenever the stored value is required for future multiplication operations. Since register access latency is smaller than shared memory access latency [1], a reduction in the number of shared memory accesses will result in an improvement in performance. In addition, we order the global memory accesses such that consecutive accesses have elements that are as close as possible in memory. This could help us get a performance advantage if a prefetcher has brought subsequent cache lines to the L1 cache.

Example of optimised memory access,

Optimised Memory Access	Unoptimized
As[ty*TILEDIM_K + tx] = A[a + ty*N + tx]; As[ty*TILEDIM_K + (tx+32)] = A[a + ty*N + (tx+32)]; As[(ty+8)*TILEDIM_K + tx] = A[a + (ty+8)*N + tx]; As[(ty+8)*TILEDIM_K + (tx+32)] = A[a + (ty+8)*N + (tx+32)];	As[ty*TILEDIM_K + tx] = A[a + ty*N + tx]; As[(ty+8)*TILEDIM_K + tx] = A[a + (ty+8)*N + tx]; As[ty*TILEDIM_K + (tx+32)] = A[a + ty*N + (tx+32)]; As[(ty+8)*TILEDIM_K + (tx+32)] = A[a + (ty+8)*N + (tx+32)];

Result and Conclusion:
The plot below shows the matrix multiplication performance achieved using the following configurations with and without optimising shared memory access:
bx=32,by=8, TILEDIM_M=128, TILEDIM_N=64, TILEDIM_K=64.

Optimising the shared memory access helped us achieve a **2 to 5 percent improvement** in performance for different matrix sizes.

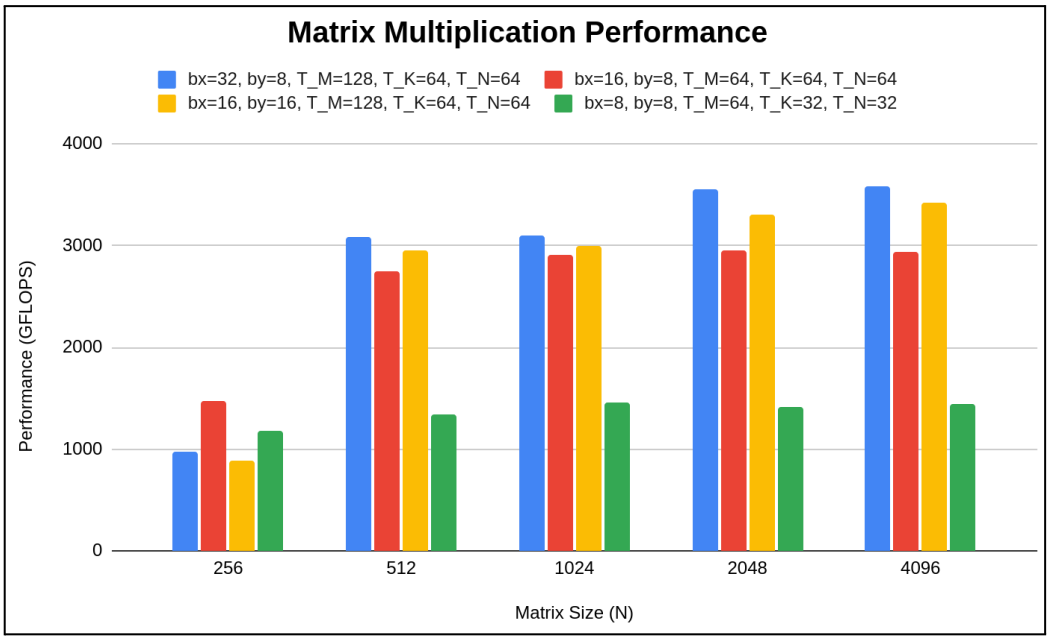


Optimization 4: Increasing the tile to compute 64 outputs per thread

To get a further boost in performance, we tried to increase the number of outputs per thread to 64. This required us to have bigger tiles (TILEDIM_M=TILEDIM_N=TILEDIM_N=128). The shared memory requirement with a bigger tile size is 128KB (128*128*2*4) which is more than the 64KB shared memory available per block in GPUs. Hence, our bigger tile implementation resulted in CudaError and was not successful.

Section 2: Result

2a: Performance Plot



T_M: TILEDIM_M, T_K: TILEDIM_K, T_N: TILEDIM_N

2b: Optimal Threadblock Size

Note: All of the thread blocks shown above perform 32 operations per thread. Hence, all have equal instruction level parallelism.

For small matrix size (N=256), optimal thread block size is [bx=16, by=8].

Analysis: The large-size thread blocks [bx=16, by=16] and [bx=32, by=8] do not work well for this matrix size because there are not enough computations available to take advantage of the block-level (tile level) parallelism. Moreover, warps within a large-size thread block face more interference from other warps to obtain execution resources and also incur a longer stall on syncthreads() call. On the other hand, medium-sized

blocks [bx=16, by=8] have fewer threads per block and thus won't have as high interference overheads and stalls as the large-size threadblocks. Moreover, medium-sized thread blocks have enough block-level parallelism to give us high performance.

For matrix size N=512, N=1024, N=2048, and N=4096 optimal thread block size is [bx=32, by=8].
Analysis: For large N, there are more computations to be done and hence there is sufficient block level parallelism. Large tile sizes help us significantly reduce the total number of global memory accesses and thereby enhance performance. The performance gained through reduction in global memory accesses outweigh the overheads due to the large number of threads in a block. Due to this reason, the large-size thread block [bx=32, by=8] performs better than small-size thread blocks [bx=8, by=8] and [bx=16, by=8].

[bx=32, by=8] thread block performs slightly better than [bx=16, by=16] thread block because the former has twice the number of threads along x-direction than the latter. This gives it more memory coalescing advantage as per our code implementation.

2c: Peak Performance

N	Peak GF	Thread Block Size
256	1467.73	bx=16, by=8, T_M=64, T_K=64, T_N=64
512	3087.60	bx=32, by=8, T_M=128, T_K=64, T_N=64
1024	3105.88	bx=32, by=8, T_M=128, T_K=64, T_N=64
2048	3559.34	bx=32, by=8, T_M=128, T_K=64, T_N=64
4096	3575.40	bx=32, by=8, T_M=128, T_K=64, T_N=64

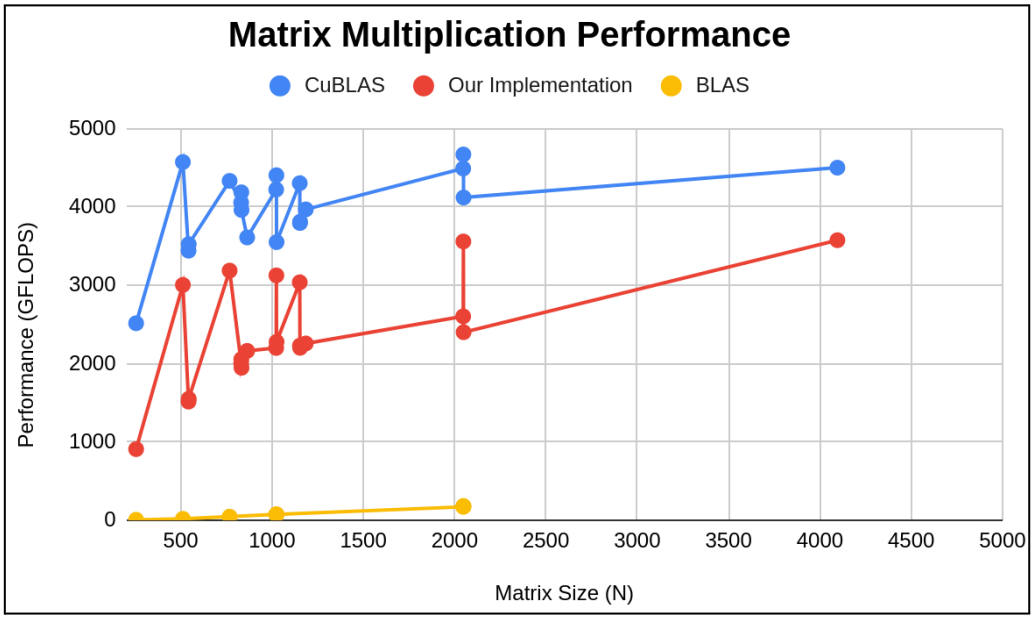
Section 3

3a:

N	Naive Performance (GFLOPS)	Our Implementation Performance (GFLOPS)	Quantitative Comparison
256	87.41	1467.73	~16.8x more performance than naive
512	157.09	3087.60	~19.6x more performance than naive
1024	246.77	3105.88	~12.6x more performance than naive
2048	277.39	3559.34	~12.8x more performance than naive

Section 4 Analysis

4a:



4b:

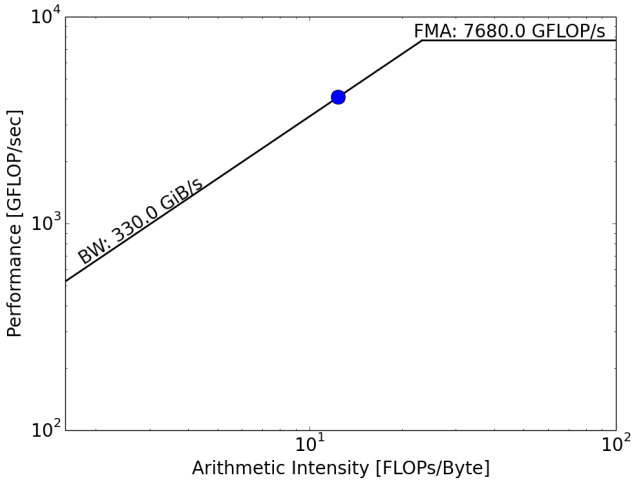
The curve for BLAS is roughly increasing (minuscule peaks and dips) as matrix size increases while the curve for our implementation faces significant dips and peaks in performance as matrix size increases.
This is because in the case of BLAS, cacheline (64KB) worth of data is fetched from the packed block and the computations are performed while reading operands from registers (least latency). Due to this, the overhead of memory accesses and computation for padded elements is not as significant as in the case of our GPU-based implementation.
On the other hand, in our GPU-based implementation, 128 bytes worth of data is brought into shared memory in a single operation and the computations are performed while reading operands from shared memory (slower than register access). Moreover, GPU-based implementation has massive thread-level parallelism and thus a large number of threads will face the shared memory access and computation overhead of padded elements. All these factors contribute to the dips and peaks in performance observed in the plot shown in 4a.

4c:

In the plot shown above in 4a, peaks are observed when the matrix size N is divisible by TILEDIM_M, TILEDIM_N, and TILEDIM_K and dips in performance are observed when the matrix size N is not divisible by TILEDIM_M, TILEDIM_N, and TILEDIM_K. This is because when matrix size is not divisible by tile dimensions then this would result in multiple threads in multiple blocks doing shared memory accesses and computation for the padded elements. Due to this overhead of shared memory accesses and computations for padded elements, the performance dip is seen.

Section 5

5a: Roofline plot for 320GiB/s

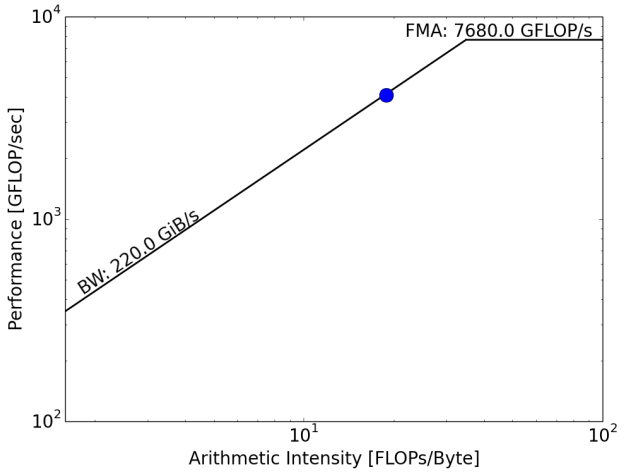


Peak Performance was computed by multiplying together the number of SMs, FP cores per SM, #ops/cycle per core, and GPU frequency. This resulted in the following arithmetic:

40(SM) x 64 (FPcore/SM) x 2 (ops/cycle) * 1.5GHz = 7.68 TFLOPS/s

The q value (Arithmetic Intensity) of our performance for 4065.7 GFLOPS/s (computed over 10-15 reps) was extrapolated to be 11.474 FLOPS/byte or 22.948 FLOPS/word since q = (achieved performance) / (# of memory ops) and 1 word is 2 bytes.

5b:Roofline plot for 220GiB/s



The new q value in ops/word is equal to (4065.7 GFLOPS/s) / ((220GiB/s) / (2 bytes/word)) = 34.42 FLOPS/word. Thus, the q value has increased, and this shows that there is more arithmetic intensity as the memory bandwidth gets smaller.

Section 6 Potential Future Work

In the future, we want to try implementing the CUTLASS method for faster performance [3]. In our current implementation, we only consider the use of thread blocks and do not consider threads at the warp level. In order to implement CUTLASS, we can extend our current work and divide the thread blocks into finer warp tiles which will enable us to get more performance boost.

Additionally, we would like to try implementing matrix multiplication with the CUDA Warp Matrix Multiply-Accumulate API (WMMA). It would be interesting to see whether using this API would help reduce the complexity of our code's logic and help boost performance.

Section 7 Extra Credit and References

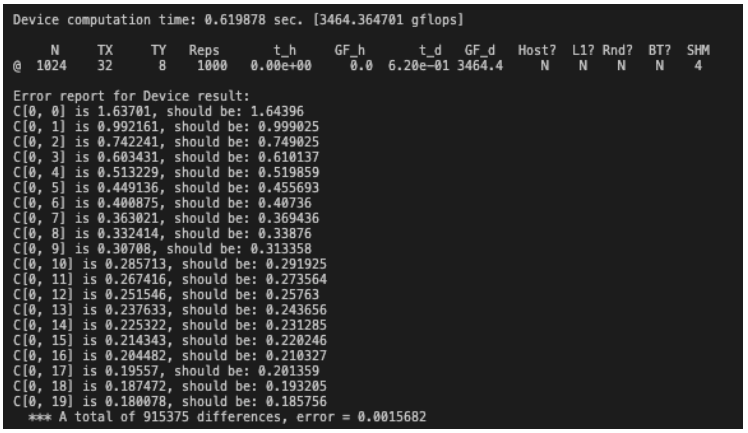
Extra Credits

Part 1 (6 points): Achieved the desired performance for matrix size N=512, 1024, and 2048 to get extra credits.

Part 2 (2 points)

Created the GitHub branch "Non_Square_Extra_Credit" for this work.

For this task, we attempted to implement matrix multiplication for two rectangular matrices A (2048 x 1024) and B (1024 x 2048) by hardcoding the dimensions into the makefile to see how we can optimize rectangular matrix multiplication. The below screenshot shows the result of our attempt that resulted in some error. Due to lack of time, we could not debug the solution. However, this should certainly be possible as long as we carefully replace the respective bounds of A and B shared memory to match the rectangular dimensions.



Part 3 (2 points)

In order to improve performance of non-square matrix multiplication, we can implement a kernel having rectangular threadblocks and rectangular tiles. This would help us handle relatively more cases where the matrix dimensions are divisible by the tile dimensions and thereby improve performance (eliminating overheads due to shared memory access and computation of padded elements).

Secondly, we can attempt to have bx dimension value greater than by dimension value in the thread block. This will help us take more advantage of memory coalescing and thereby help us improve performance.

References

[1] *Better performance at Lower Occupancy - Nvidia*. (n.d.). Retrieved November 10, 2022, from https://www.nvidia.com/content/GTC-2010/pdfs/2238_GTC2010.pdf

[2] *Virtual workshop*. Cornell Virtual Workshop: Memory Coalescing. (n.d.). Retrieved November 9, 2022, from https://cvw.cac.cornell.edu/gpu/coalesced

[3] *Cutlass: Fast linear algebra in Cuda C++*. NVIDIA Technical Blog. (2022, August 21). Retrieved November 9, 2022, from https://developer.nvidia.com/blog/cutlass-linear-algebra-cuda/

[4] *Tuning CUDA Applications for Turing* - https://docs.nvidia.com/cuda/pdf/Turing_Tuning_Guide.pdf