Daniel Pelis
CPE 810
10/09/2020

Lab 1: Matrix-Matrix Multiplication

Objective:

Implement a tiled dense matrix multiplication routine (C=A*B) using shared memory in CUDA. A and B matrix elements can be randomly generated on the host. Your code should be able to handle arbitrary sizes for input rectangular matrices.

Instructions:

Make a new folder for your project. Copy and modify the code of matrixMul in CUDA sample code to include the following key functions:

1. allocate device memory
2. copy host memory to device
3. initialize thread block and kernel grid dimensions
4. invoke CUDA kernel
5. copy results from device to host
6. deallocate device memory
7. implement the matrix-matrix multiplication routine using shared memory and tiling algorithm
8. handle boundary conditions (thread divergence) when dealing with arbitrary matrix sizes

Your final executable can be run using the following command:

```
.TiledMatrixMul -i <rowDimA> <colDimA> <colDimB>
```

About input parameters:
<rowDimA> is the row dimension of the rectangular matrix A.
<colDimA> is the col dimension of the rectangular matrix A.
<colDimB> is the col dimension of the rectangular matrix B.

Implementation:

My matrix multiplication program begins by checking for the '-i' flag to set the matrix dimensions. If the flag is found it will read in the values and set rowDimA, colDimA, and comDimB respectively. If the flag is absent the default matrix size is 2 x 2. The host code then allocated memory for the A and B matrices based on the dimension given. The sizes and memory sizes of these matrices are also calculated at this point. The matrices are then filled with random values based on time. Now the memory space for the resulting C matrix is allocated and the device variables for each matrix is initialized. Matrix A and Matrix B are copied into the device's global memory and the grid and block dimensions are calculated based on the tile width, which is set to 16, and the given matrix dimensions. A timer is then started, and the kernel is then iterated through 150 times. The iterations allow me to get e more precise reading of the kernel's execution time.

My kernel begins my allocating shared memory to fit the necessary A and B values for the thread. Variables for block and thread indices are set and the current row and column of C are calculated based on the current tile. The number of phases is calculated based on the column width of matrix A and the tile width. The thread then loops through its phases at the same time are the other threads in its warp. They will all copy two values from global memory into their shared memory and wait for each other. Once this is done all the values of A and B will be in the shared memory that is needed for calculating the current block of C. Conditional are placed so that if the current tile extends past the C matrix's boundaries, the value copied into the shared memory by that thread will be set to zero. After the threads sync, they all calculated their respective C value and sync before writing this value back into the global memory. This is also covered by a conditional checking is the thread is outside the C matrix's boundaries.

Once all the iterations of the kernel are done, the timer is stopped, and the elapsed time is calculated. This value and the number of iterations is used to calculate the total time for one iteration. The number of flops performed per iteration is also calculated based on the sizes of the matrices. These values are then used to determine the gigaflops per second of the kernel.

Finally, the calculated values for matrix C are copied back into the host's memory and the device memory is freed. The matrix multiplication is re done by the CPU only and the results are compared to check the kernels accuracy. Lastly, the host memory is freed to prevent memory leakage.

Questions:

1. How many floating operations are being performed in your dense matrix multiply kernel if the matrix size is N times N? Explain.

   To calculate the resulting matrix, the kernel must perform two floating point operations for every pair of values from matrices A and B. It must first multiple the corresponding values from matrix A and matrix B and then add the result to the corresponding cell in matrix C. These two operations must be performed for every value along the corresponding row of matrix A and column of matrix B. Since the widths are assumed to be N, we can conclude that these 2 operations are performed N times each for a single cell in the resulting matrix C. If the resulting matrix is N x N then there are N^2 values that must be calculated. Assuming the kernel is using one thread per value we can conclude that the amount of floating-point operations being performed is $2N \times N^2$ = **$2N^3$**.

2. How many global memory reads are being performed by your kernel? Explain.

   Without using a tiling method, my kernel would have made 2 * N global memory reads for every cell of the resulting matrix, assuming the inputs were both N x N matrices. 2N comes from the fact that each thread must read the values for an entire row of matrix A and an entire column for matrix B. Therefore overall, it would have made $2N^3$ global memory reads, causing an enormous delay for large Ns. Since a tiling method was utilized my kernel only needs to make **two** global memory reads for every phase. The number of phases each thread will go through is equivalent to the column width of matrix A (N) divided by the set tile width. By default, I have set my tile width to be 16, so the number of phases for a **N x N** matrix for my kernel would be **N/16**. Finally, the number of global memory reads my kernel performs is $2 * N^2 * N/16$ = **$1/8\ N^3$**. This derivation shows use that the tiling method has reduced the number of global memory reads to 1/8 the original amount.

3. How many global memory writes are being performed by your kernel? Explain.

   My kernel only performs one global memory write for every value in the resulting matrix. Each thread is assigned a cell in the resulting matrix and only after it has finished the computations for that cell will it write the value to the global memory. Assuming the kernel is working on an N x N matrix, the amount of global memory writes will be **N^2**.

4. Describe what further optimizations can be implemented to your kernel to achieve a performance speedup.

   One thing that may cause decreased performance in my kernel could be a large amount of diverging threads. Since my tile size is a set value some matrix sizes could cause my diverging threads than others. To combat this, I could add a generator to my code to set tile width at runtime. This could produce a tile width that better match the input matrix size and current GPU specs. The only limitation would be the tile would have to be 32 or smaller since the warp size is 32 threads for most modern GPUs. Exceeding this would cause issues in the SMs since they are running all the threads in the warp simultaneously with Single Instruction Multiple Data (SIMD)

5. Suppose you have matrices with dimensions bigger than the max thread dimensions allowed in CUDA. Sketch an algorithm that would perform matrix multiplication algorithm that would perform the multiplication in this case.

   Assuming global memory would not have an issue holding the matrices, a method resembling the tiling method could be implemented. In the host code, the resulting matrix can be split into many sub matrices that can be computed separately. The size of the sub matrices would be dependent on the max thread dimensions. The matrix multiplication kernel would be iterated through and given one sub matrix at a time. The host code would the keep track of the resulting matrix C values by pulling them out of global memory after every iteration and putting them in their appropriate position.

6. Suppose you have matrices that would not fit in global memory. Sketch an algorithm that would perform matrix multiplication algorithm that would perform the multiplication out of place.

   Like the last question, this limitation could also be overcome by splitting the matrix into smaller sub matrices. The difference for this algorithm is that the sizes of the sub matrices would correspond to the number of columns and rows that can fit into global memory at the same time. Assuming the matrices are N x N, the number of rows or columns that could be fit in the global memory would be N * 4 /global memory size. This value divided by 2 would then tell us how many rows and how many columns we could load into the global memory, $\frac{2 * \mathrm{N}}{\text{global memory size}}$. Now assuming we can choose block size safely below the maximum we just calculated we would also have room for the resulting matrix in the global memory. The host code would iterate over the kernel, passing in the columns and rows that would fit in global memory based on the derivation above and the device would calculate the values for the sub matrix. The host would also be responsible for taking the resulting values out of the global memory and clearing the global memory for the next iteration.

Experimental Results:

Experimentation was done on my kernel using two different tile widths, first 16 was tested then 32. Overall, there was a general increase in performance for larger matrix sizes. Both the 16 and 32 tile widths had similar performance metrics for the larger matrices of the tests. This is most likely due to the matrices not being large enough for a noticeable difference to occur. The main differences in the performances happened during the computation of the smaller matrices. The performance of the 32-tile kernel was nearly half of the 16-tile kernel. This is likely due to diverging threads during the computation. Since the 32-kernel size is larger it will have more threads that exceed the bounds of the resulting C matrix and diverge. They diverge due to the boundary conditionals in the kernel and when this happens the Stream Multiprocessor must run the whole warp again for these threads. Since the 16-tile kernel is smaller it shows less divergence on the smaller matrix sizes. This is not seen in the larger sized matrices since the amount of diverging threads is smaller compared to the total amount of threads.

Experimentation also showed that my kernel was no where near the peak performance of my gpu (gtx 1070). My GPU's global memory bandwidth is 256 GB/s and multiplying that by 4, for the size of a floating-point value gets a max of 64 GFLOPs. Since I utilized a tiling method, my memory accessed was reduced by a factor of the tile width. For the 16-width tile I increased the compute-to-global-memory ratio increase from 1 to 16 making new max for this kernel 1024 GFLOPS (64 * 16). The theoretical max for my GPU is listed at 5783 GFLOPs, but to reach this I would have to continue increasing my compute-to-global-memory ratio. My kernel is mostly likely not reaching the 1024 GFLOP limit due to unhandled diverging threads. As discussed in question 4 I could probably add code to alter the tile width at run-time to create more suitable tiles for the input matrix.

Tile_Width* = 16:

rowDimA x colDimA x colDimB	Average* GFLOPS	Average* Time (ms)
64 x 64 x 64	14.84	0.036
70 x 70 x 70	14.11	0.049
1000 x 1000 x 1000	60.27	32.95
500 x 1000 x 1500	61.47	24.41

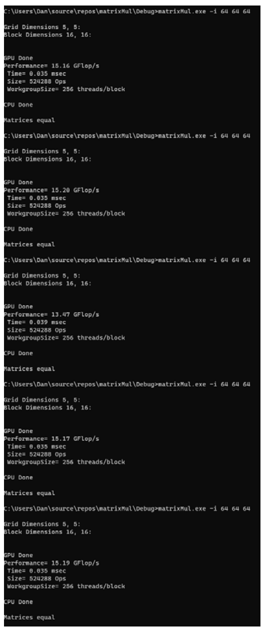
*Thread block size is set to be equal to tile width
*Averages taken over 5 runs

Tile_Width* = 32:

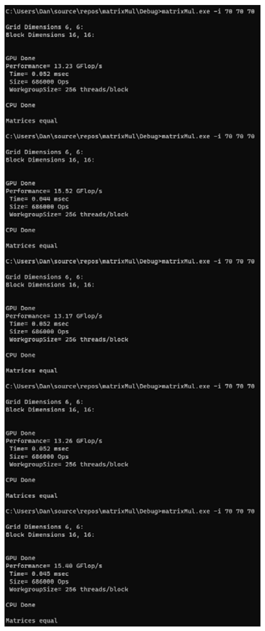
rowDimA x colDimA x colDimB	Average* GFLOPS	Average* Time (ms)
64 x 64 x 64	8.43	0.062
70 x 70 x 70	5.52	0.125
1000 x 1000 x 1000	61.53	32.57
500 x 1000 x 1500	60.39	24.93

*Thread block size is set to be equal to tile width
*Averages taken over 5 runs

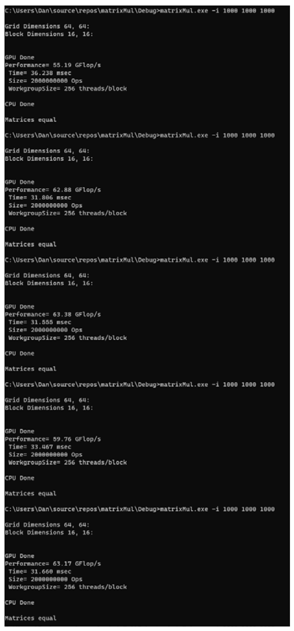
Outputs of 16 tile width runs:



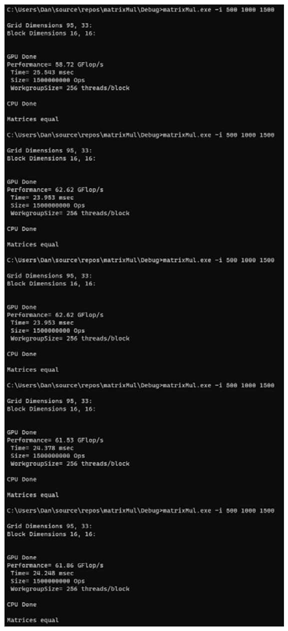
```
C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 64 64 64

Grid Dimensions 5, 5:
Block Dimensions 16, 16:


GPU Done
Performance= 15.16 GFlop/s
 Time= 0.035 msec
 Size= 524288 Ops
 WorkgroupSize= 256 threads/block

CPU Done

Matrices equal

C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 64 64 64

Grid Dimensions 5, 5:
Block Dimensions 16, 16:


GPU Done
Performance= 15.20 GFlop/s
 Time= 0.035 msec
 Size= 524288 Ops
 WorkgroupSize= 256 threads/block

CPU Done

Matrices equal

C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 64 64 64

Grid Dimensions 5, 5:
Block Dimensions 16, 16:


GPU Done
Performance= 13.47 GFlop/s
 Time= 0.039 msec
 Size= 524288 Ops
 WorkgroupSize= 256 threads/block

CPU Done

Matrices equal

C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 64 64 64

Grid Dimensions 5, 5:
Block Dimensions 16, 16:


GPU Done
Performance= 15.17 GFlop/s
 Time= 0.035 msec
 Size= 524288 Ops
 WorkgroupSize= 256 threads/block

CPU Done

Matrices equal

C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 64 64 64

Grid Dimensions 5, 5:
Block Dimensions 16, 16:


GPU Done
Performance= 15.19 GFlop/s
 Time= 0.035 msec
 Size= 524288 Ops
 WorkgroupSize= 256 threads/block

CPU Done

Matrices equal
```



```
C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 70 70 70

Grid Dimensions 6, 6:
Block Dimensions 16, 16:


GPU Done
Performance= 13.23 GFlop/s
 Time= 0.052 msec
 Size= 686000 Ops
 WorkgroupSize= 256 threads/block

CPU Done

Matrices equal

C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 70 70 70

Grid Dimensions 6, 6:
Block Dimensions 16, 16:


GPU Done
Performance= 15.52 GFlop/s
 Time= 0.044 msec
 Size= 686000 Ops
 WorkgroupSize= 256 threads/block

CPU Done

Matrices equal

C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 70 70 70

Grid Dimensions 6, 6:
Block Dimensions 16, 16:


GPU Done
Performance= 13.17 GFlop/s
 Time= 0.052 msec
 Size= 686000 Ops
 WorkgroupSize= 256 threads/block

CPU Done

Matrices equal

C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 70 70 70

Grid Dimensions 6, 6:
Block Dimensions 16, 16:


GPU Done
Performance= 13.26 GFlop/s
 Time= 0.052 msec
 Size= 686000 Ops
 WorkgroupSize= 256 threads/block

CPU Done

Matrices equal

C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 70 70 70

Grid Dimensions 6, 6:
Block Dimensions 16, 16:


GPU Done
Performance= 15.40 GFlop/s
 Time= 0.045 msec
 Size= 686000 Ops
 WorkgroupSize= 256 threads/block

CPU Done

Matrices equal
```

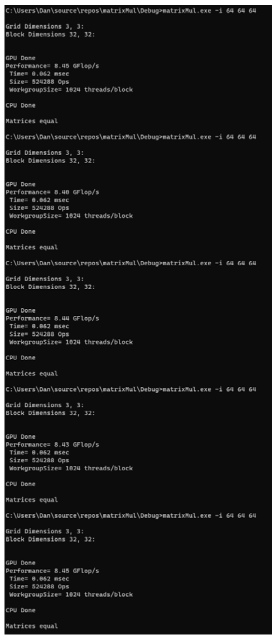


```
C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 1000 1000 1000

Grid Dimensions 64, 64:
Block Dimensions 16, 16:


GPU Done
Performance= 55.19 GFlop/s
 Time= 36.238 msec
 Size= 2000000000 Ops
 WorkgroupSize= 256 threads/block

CPU Done

Matrices equal

C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 1000 1000 1000

Grid Dimensions 64, 64:
Block Dimensions 16, 16:


GPU Done
Performance= 62.88 GFlop/s
 Time= 31.806 msec
 Size= 2000000000 Ops
 WorkgroupSize= 256 threads/block

CPU Done

Matrices equal

C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 1000 1000 1000

Grid Dimensions 64, 64:
Block Dimensions 16, 16:


GPU Done
Performance= 63.38 GFlop/s
 Time= 31.555 msec
 Size= 2000000000 Ops
 WorkgroupSize= 256 threads/block

CPU Done

Matrices equal

C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 1000 1000 1000

Grid Dimensions 64, 64:
Block Dimensions 16, 16:


GPU Done
Performance= 59.76 GFlop/s
 Time= 33.467 msec
 Size= 2000000000 Ops
 WorkgroupSize= 256 threads/block

CPU Done

Matrices equal

C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 1000 1000 1000

Grid Dimensions 64, 64:
Block Dimensions 16, 16:


GPU Done
Performance= 63.17 GFlop/s
 Time= 31.660 msec
 Size= 2000000000 Ops
 WorkgroupSize= 256 threads/block

CPU Done

Matrices equal
```

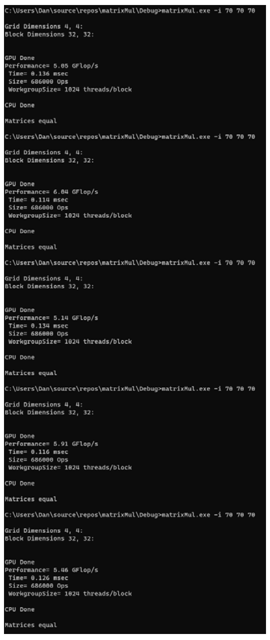


```
C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 500 1000 1500

Grid Dimensions 95, 33:
Block Dimensions 16, 16:


GPU Done
Performance= 58.72 GFlop/s
 Time= 25.543 msec
 Size= 1500000000 Ops
 WorkgroupSize= 256 threads/block

CPU Done

Matrices equal

C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 500 1000 1500

Grid Dimensions 95, 33:
Block Dimensions 16, 16:


GPU Done
Performance= 62.62 GFlop/s
 Time= 23.953 msec
 Size= 1500000000 Ops
 WorkgroupSize= 256 threads/block

CPU Done

Matrices equal

C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 500 1000 1500

Grid Dimensions 95, 33:
Block Dimensions 16, 16:


GPU Done
Performance= 62.62 GFlop/s
 Time= 23.953 msec
 Size= 1500000000 Ops
 WorkgroupSize= 256 threads/block

CPU Done

Matrices equal

C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 500 1000 1500

Grid Dimensions 95, 33:
Block Dimensions 16, 16:


GPU Done
Performance= 61.53 GFlop/s
 Time= 24.378 msec
 Size= 1500000000 Ops
 WorkgroupSize= 256 threads/block

CPU Done

Matrices equal

C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 500 1000 1500

Grid Dimensions 95, 33:
Block Dimensions 16, 16:


GPU Done
Performance= 61.86 GFlop/s
 Time= 24.248 msec
 Size= 1500000000 Ops
 WorkgroupSize= 256 threads/block

CPU Done

Matrices equal
```

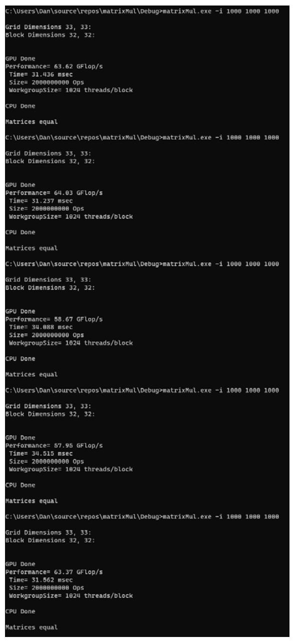
Outputs of 32 tile width runs:



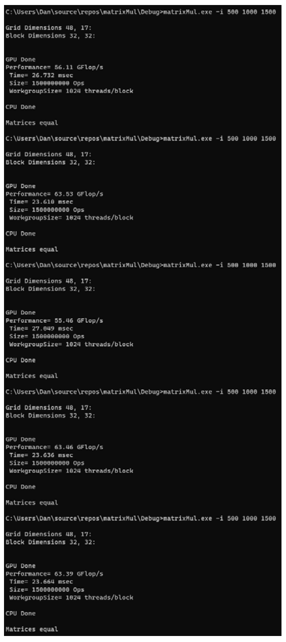
```
C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 64 64 64

Grid Dimensions 3, 3:
Block Dimensions 32, 32:


GPU Done
Performance= 8.45 GFlop/s
 Time= 0.062 msec
 Size= 524288 Ops
 WorkgroupSize= 1024 threads/block

CPU Done

Matrices equal

C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 64 64 64

Grid Dimensions 3, 3:
Block Dimensions 32, 32:


GPU Done
Performance= 8.40 GFlop/s
 Time= 0.062 msec
 Size= 524288 Ops
 WorkgroupSize= 1024 threads/block

CPU Done

Matrices equal

C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 64 64 64

Grid Dimensions 3, 3:
Block Dimensions 32, 32:


GPU Done
Performance= 8.44 GFlop/s
 Time= 0.062 msec
 Size= 524288 Ops
 WorkgroupSize= 1024 threads/block

CPU Done

Matrices equal

C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 64 64 64

Grid Dimensions 3, 3:
Block Dimensions 32, 32:


GPU Done
Performance= 8.43 GFlop/s
 Time= 0.062 msec
 Size= 524288 Ops
 WorkgroupSize= 1024 threads/block

CPU Done

Matrices equal

C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 64 64 64

Grid Dimensions 3, 3:
Block Dimensions 32, 32:


GPU Done
Performance= 8.45 GFlop/s
 Time= 0.062 msec
 Size= 524288 Ops
 WorkgroupSize= 1024 threads/block

CPU Done

Matrices equal
```



```
C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 70 70 70

Grid Dimensions 4, 4:
Block Dimensions 32, 32:


GPU Done
Performance= 5.05 GFlop/s
 Time= 0.136 msec
 Size= 686000 Ops
 WorkgroupSize= 1024 threads/block

CPU Done

Matrices equal

C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 70 70 70

Grid Dimensions 4, 4:
Block Dimensions 32, 32:


GPU Done
Performance= 6.04 GFlop/s
 Time= 0.114 msec
 Size= 686000 Ops
 WorkgroupSize= 1024 threads/block

CPU Done

Matrices equal

C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 70 70 70

Grid Dimensions 4, 4:
Block Dimensions 32, 32:


GPU Done
Performance= 5.14 GFlop/s
 Time= 0.134 msec
 Size= 686000 Ops
 WorkgroupSize= 1024 threads/block

CPU Done

Matrices equal

C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 70 70 70

Grid Dimensions 4, 4:
Block Dimensions 32, 32:


GPU Done
Performance= 5.91 GFlop/s
 Time= 0.116 msec
 Size= 686000 Ops
 WorkgroupSize= 1024 threads/block

CPU Done

Matrices equal

C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 70 70 70

Grid Dimensions 4, 4:
Block Dimensions 32, 32:


GPU Done
Performance= 5.46 GFlop/s
 Time= 0.126 msec
 Size= 686000 Ops
 WorkgroupSize= 1024 threads/block

CPU Done

Matrices equal
```



```
C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 1000 1000 1000

Grid Dimensions 33, 33:
Block Dimensions 32, 32:


GPU Done
Performance= 63.62 GFlop/s
 Time= 31.436 msec
 Size= 2000000000 Ops
 WorkgroupSize= 1024 threads/block

CPU Done

Matrices equal

C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 1000 1000 1000

Grid Dimensions 33, 33:
Block Dimensions 32, 32:


GPU Done
Performance= 64.03 GFlop/s
 Time= 31.237 msec
 Size= 2000000000 Ops
 WorkgroupSize= 1024 threads/block

CPU Done

Matrices equal

C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 1000 1000 1000

Grid Dimensions 33, 33:
Block Dimensions 32, 32:


GPU Done
Performance= 58.67 GFlop/s
 Time= 34.088 msec
 Size= 2000000000 Ops
 WorkgroupSize= 1024 threads/block

CPU Done

Matrices equal

C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 1000 1000 1000

Grid Dimensions 33, 33:
Block Dimensions 32, 32:


GPU Done
Performance= 57.95 GFlop/s
 Time= 34.515 msec
 Size= 2000000000 Ops
 WorkgroupSize= 1024 threads/block

CPU Done

Matrices equal

C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 1000 1000 1000

Grid Dimensions 33, 33:
Block Dimensions 32, 32:


GPU Done
Performance= 63.37 GFlop/s
 Time= 31.562 msec
 Size= 2000000000 Ops
 WorkgroupSize= 1024 threads/block

CPU Done

Matrices equal
```



```
C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 500 1000 1500

Grid Dimensions 48, 17:
Block Dimensions 32, 32:


GPU Done
Performance= 56.11 GFlop/s
 Time= 26.732 msec
 Size= 1500000000 Ops
 WorkgroupSize= 1024 threads/block

CPU Done

Matrices equal

C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 500 1000 1500

Grid Dimensions 48, 17:
Block Dimensions 32, 32:


GPU Done
Performance= 63.53 GFlop/s
 Time= 23.610 msec
 Size= 1500000000 Ops
 WorkgroupSize= 1024 threads/block

CPU Done

Matrices equal

C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 500 1000 1500

Grid Dimensions 48, 17:
Block Dimensions 32, 32:


GPU Done
Performance= 55.46 GFlop/s
 Time= 27.049 msec
 Size= 1500000000 Ops
 WorkgroupSize= 1024 threads/block

CPU Done

Matrices equal

C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 500 1000 1500

Grid Dimensions 48, 17:
Block Dimensions 32, 32:


GPU Done
Performance= 63.46 GFlop/s
 Time= 23.636 msec
 Size= 1500000000 Ops
 WorkgroupSize= 1024 threads/block

CPU Done

Matrices equal

C:\Users\Dan\source\repos\matrixMul\Debug>matrixMul.exe -i 500 1000 1500

Grid Dimensions 48, 17:
Block Dimensions 32, 32:


GPU Done
Performance= 63.39 GFlop/s
 Time= 23.664 msec
 Size= 1500000000 Ops
 WorkgroupSize= 1024 threads/block

CPU Done

Matrices equal
```