Heterogeneous Computing - HOMEWORK 1 - ECGR-6090

Observed and Documented by: Sumukh Raghuram Bhat – 801131997

The testing platform information is as follows:

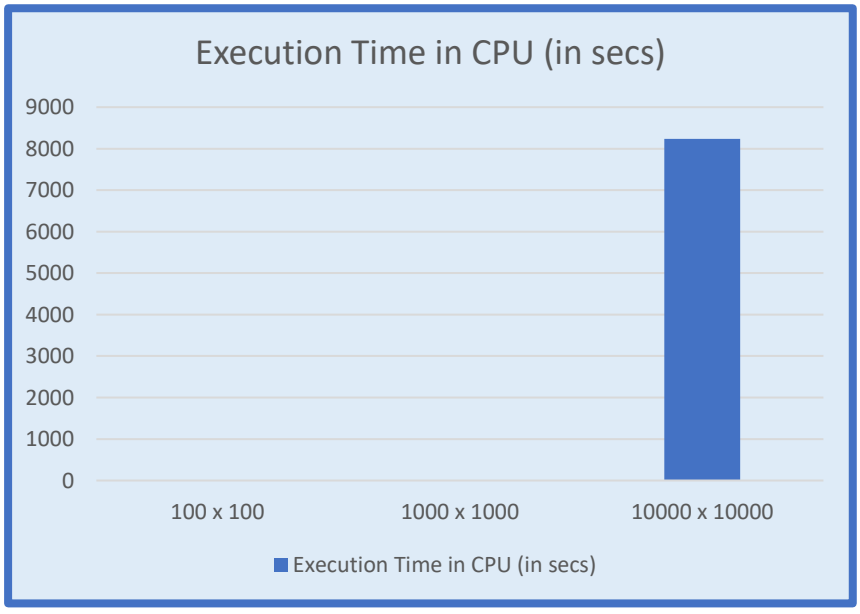
CPU: Dual core Intel Core i5-4200U
Software environment: Linux (64-bit), Ubuntu 18.04
Development Language: CUDA C and C.
CUDA Version: 9.2 using MAMBA cluster.

1. **A.** The matrix multiplication using the naïve method on the CPU is implemented and is shown.

Matrix size is taken as 100, 1000 and 10000.

The table and the graph for the execution time for various matrix sizes is:

Matrix Size	Execution Time in CPU (in secs)
100 x 100	0.00973
1000 x 1000	5.841174
10000 x 10000	8235.71667



1. **B.** The matrix multiplication using the naïve method on the GPU is implemented and is shown.

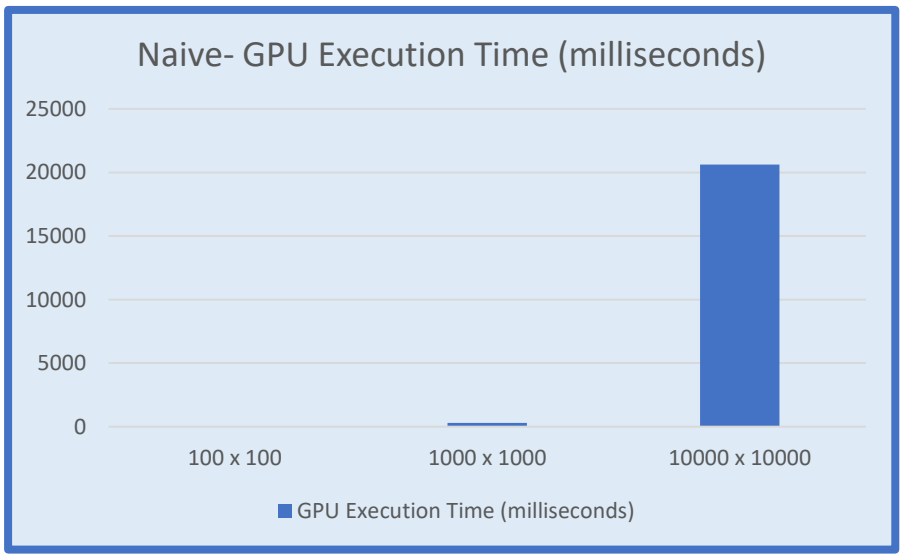
Each thread loads one row of matrix A and one column of matrix B from the global memory, performs the necessary calculations and stores the result back to matrix C in the global memory.

In our computation, we keep the thread block dimension as (16, 16) while varying the size of the matrix.

Matrix size is taken as 100, 1000 and 10000.

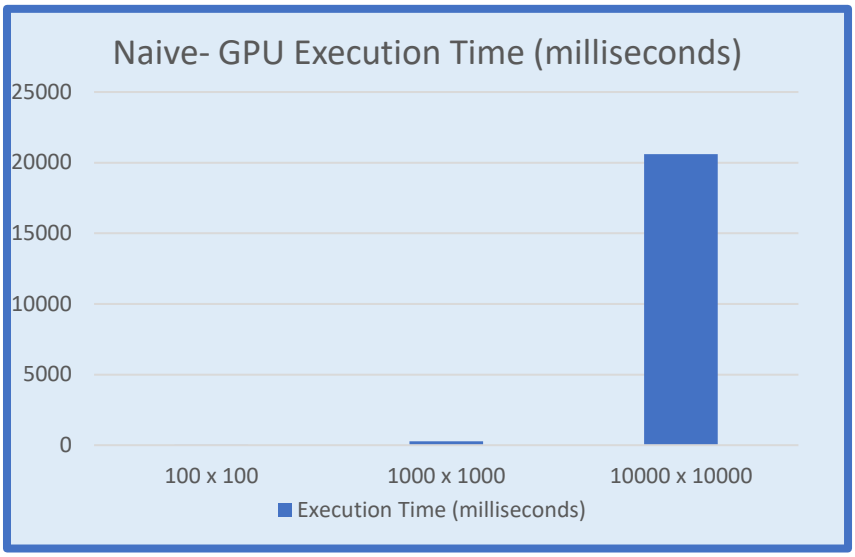
The table and the graph for the execution time for various matrix sizes considering memory transfer is:

Matrix Size	GPU Execution Time (milliseconds)
100 x 100	3.292256
1000 x 1000	288.394348
10000 x 10000	20626.226562



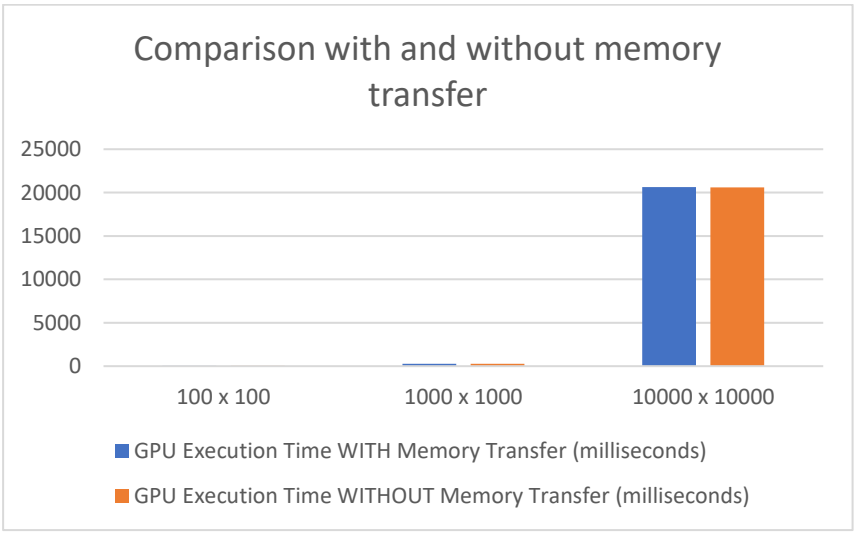
The table and the graph for the execution time for various matrix sizes without considering memory transfer is:

Matrix Size	GPU Execution Time (milliseconds)
100 x 100	3.236608
1000 x 1000	285.605042
10000 x 10000	20604.472656



The below graph compares the execution time with and without memory transfer.

We can see that there is **negligible change** when with and without memory graph is plotted.



When we do not consider the time of moving the data while porting the code to CUDA is lesser than that when we consider the transfer time. **The execution time is less without memory transfer.**

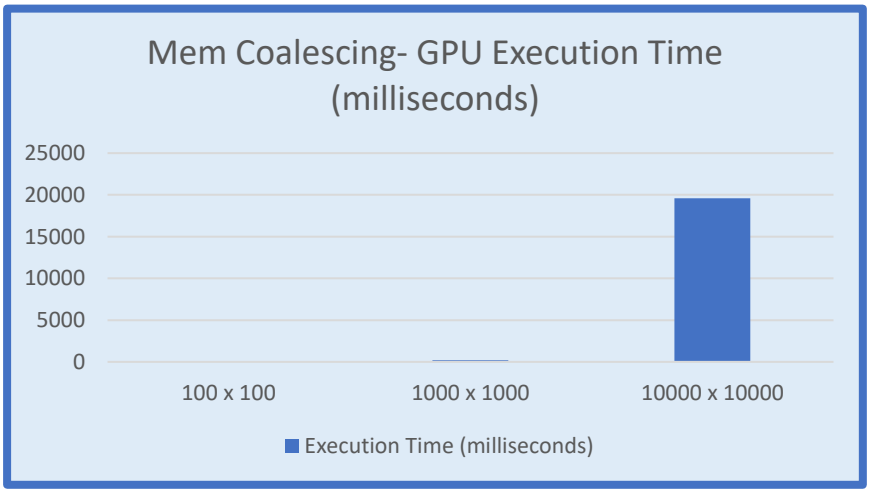
2. This time we use memory coalescing to enhance the memory access efficiency and reduce memory divergence.

To take full advantage of the high memory bandwidth of the GPU, the reading from global memory must run in parallel.

In order to use spatial locality, we transpose matrix B. By doing so, we improve the memory access efficiency.

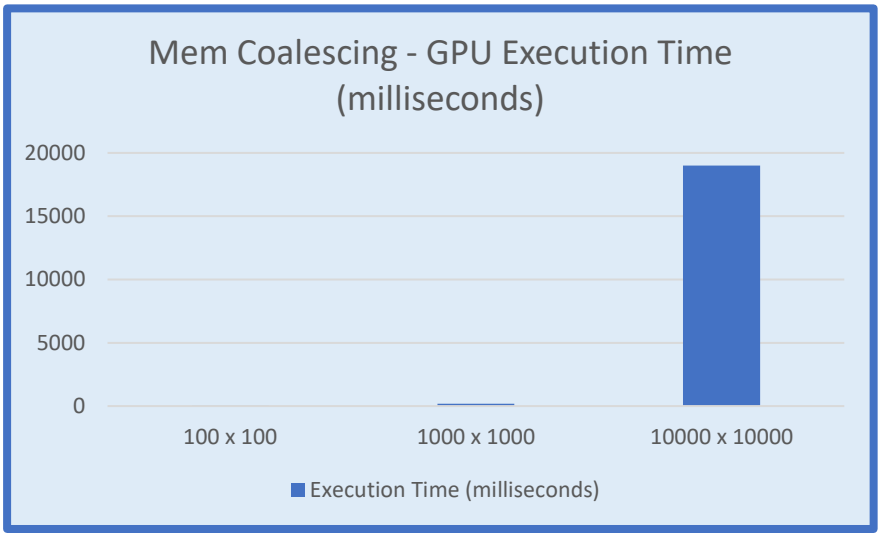
The table and the graph for the execution time for various matrix sizes considering memory transfer is:

Matrix Size	GPU Execution Time (milliseconds)
100 x 100	2.161984
1000 x 1000	183.847427
10000 x 10000	19611.178564

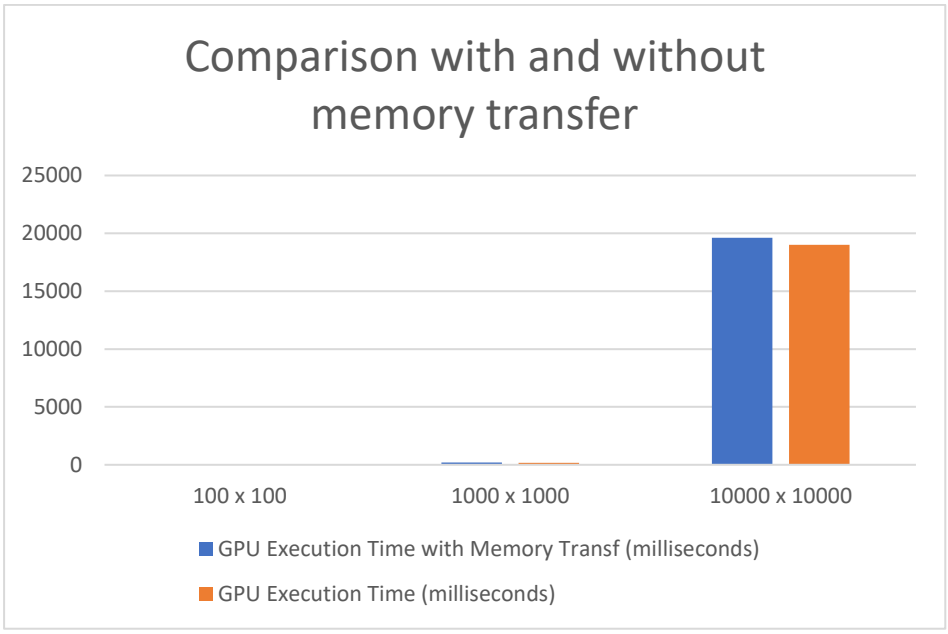


The table and the graph for the execution time for various matrix sizes without considering memory transfer is:

Matrix Size	GPU Execution Time (milliseconds)
100 x 100	2.114208
1000 x 1000	180.441751
10000 x 10000	19005.457368



The below graph compares the execution time with and without memory transfer.



There is a reduction in the execution time when we use the transpose of a matrix B by using spatial locality.

The execution time is less without memory transfer.

3. Tiling with shared memory can be seen implemented on increasing Matrix Sizes (100*100, 1000*1000, 10000*10000), with thread block dimension of (16,16).

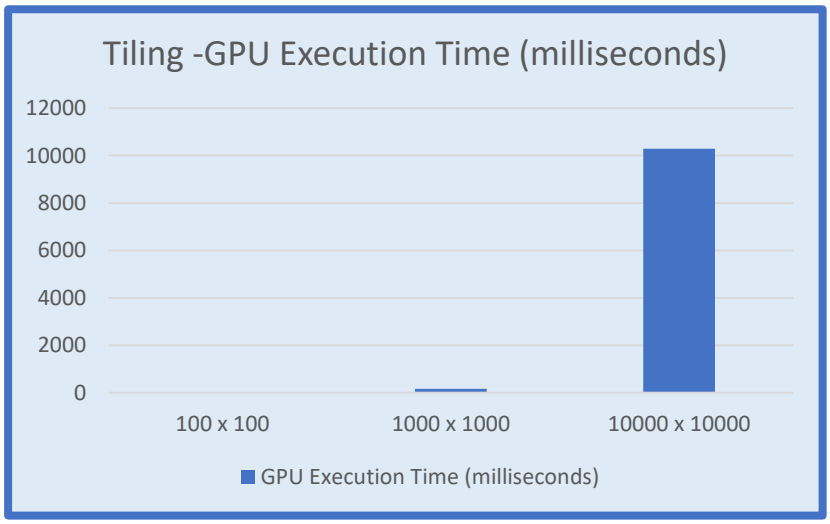
To improve the performance of the GPU, the tiled matrix multiplication can be applied.

One thread will compute one tile of matrix C. One thread in the thread block will compute one element of the tile. The kernel computes matrix C in multiple iterations.

In each iteration, one thread block loads one tile of A and one tile of B from global memory to shared memory, performs the calculations and stores the result temporary in a register. These repeats and the final value of C is transferred from the register to global memory.

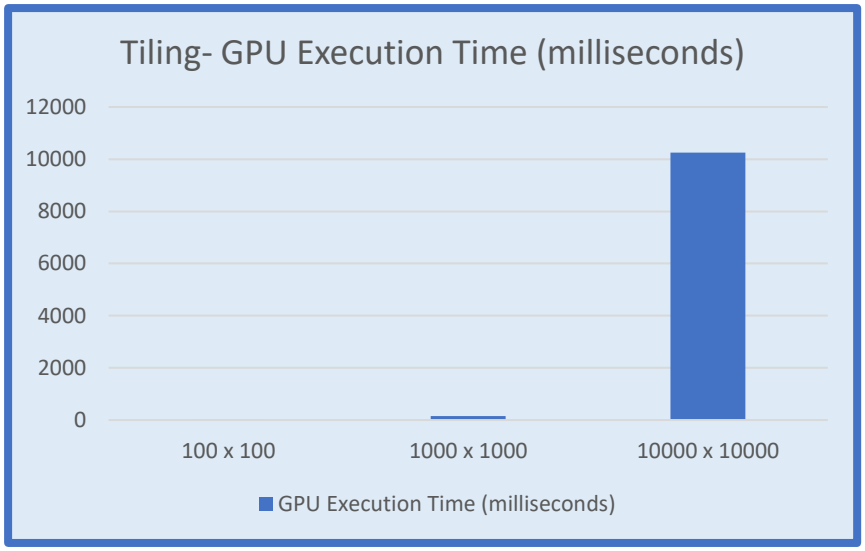
The table and the graph for the execution time for various matrix sizes considering memory transfer is:

Matrix Size	GPU Execution Time (milliseconds)
100 x 100	1.867744
1000 x 1000	155.755325
10000 x 10000	10290.892578

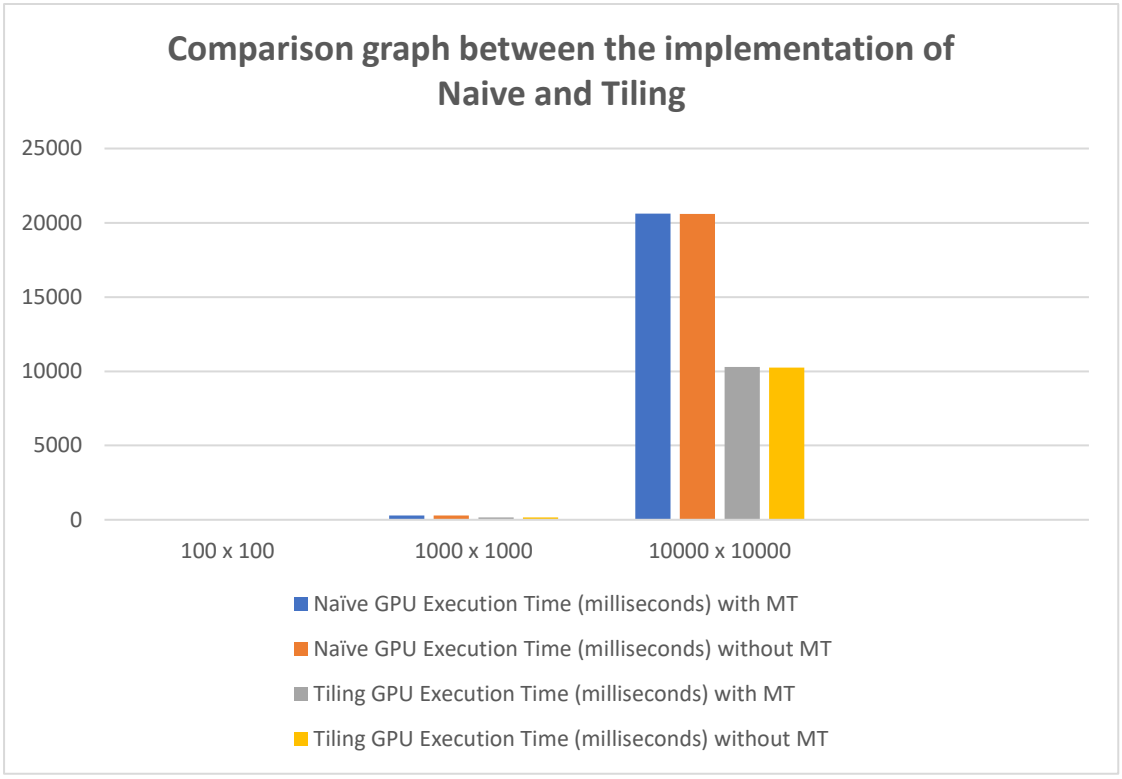


The table and the graph for the execution time for various matrix sizes without considering memory transfer is:

Matrix Size	GPU Execution Time (milliseconds)
100 x 100	1.855488
1000 x 1000	152.347839
10000 x 10000	10254.843750



Below is the comparison graph between the implementation of Naive and Tiling.



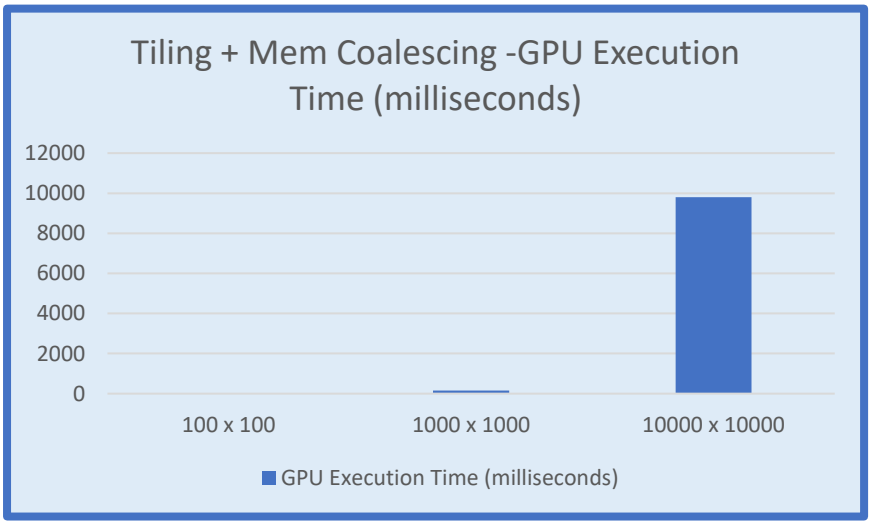
From the graph we can see the execution time reduces by almost 2 times by using tiling and shared memory.

4. Implement memory coalescing to the tiling implementation.

Here, the matrix B is transposed in order to make use of the spatial locality. Hence memory coalescing is used to enhance the memory access efficiency and reduce memory divergence.

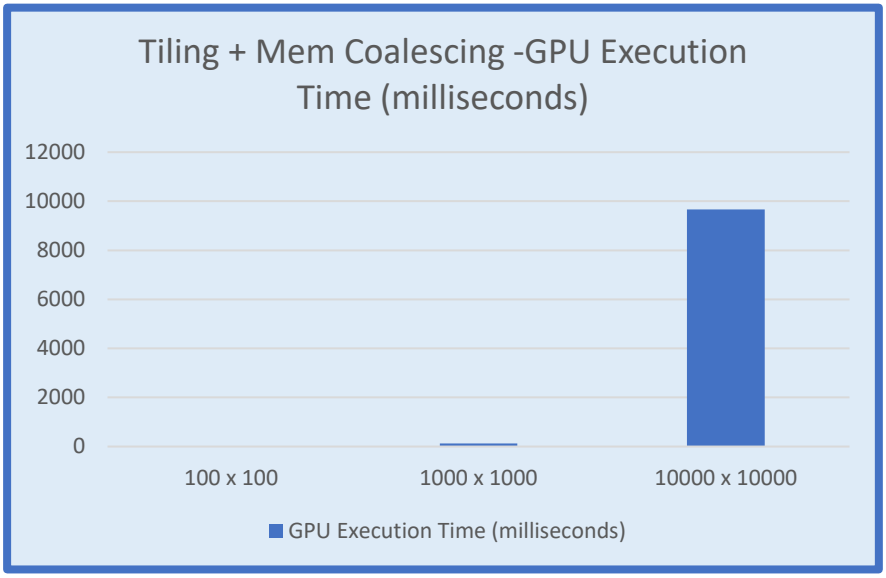
The table and the graph for the execution time for various matrix sizes considering memory transfer is:

Matrix Size	GPU Execution Time (milliseconds)
100 x 100	1.65238
1000 x 1000	147.75318
10000 x 10000	9801.83375

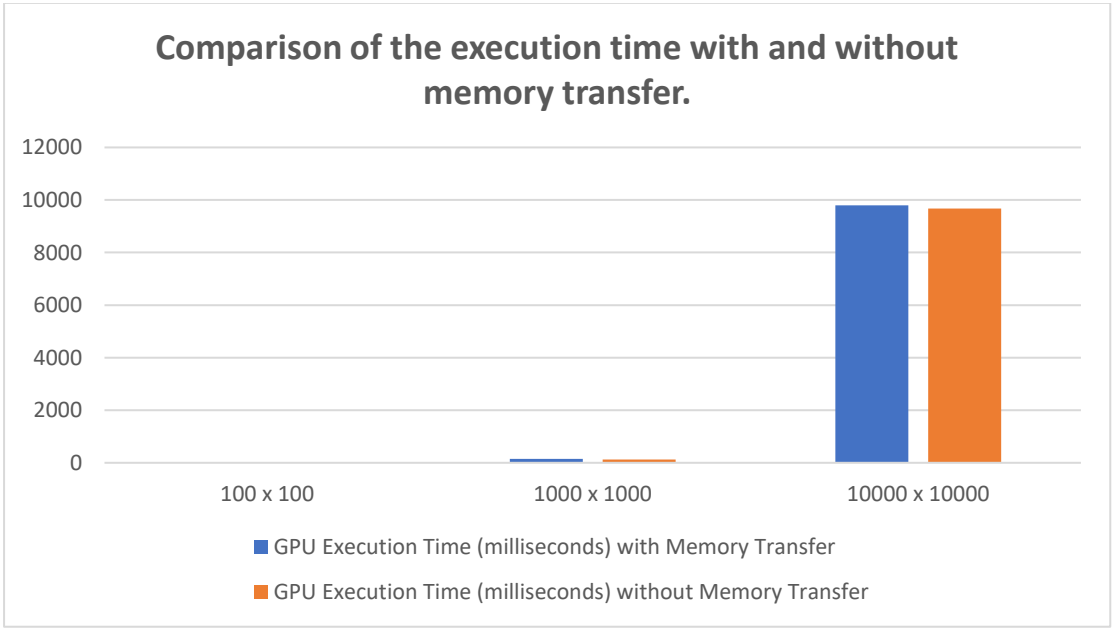


The table and the graph for the execution time for various matrix sizes without considering memory transfer is:

Matrix Size	GPU Execution Time (milliseconds)
100 x 100	1.43942
1000 x 1000	123.81347
10000 x 10000	9670.655273

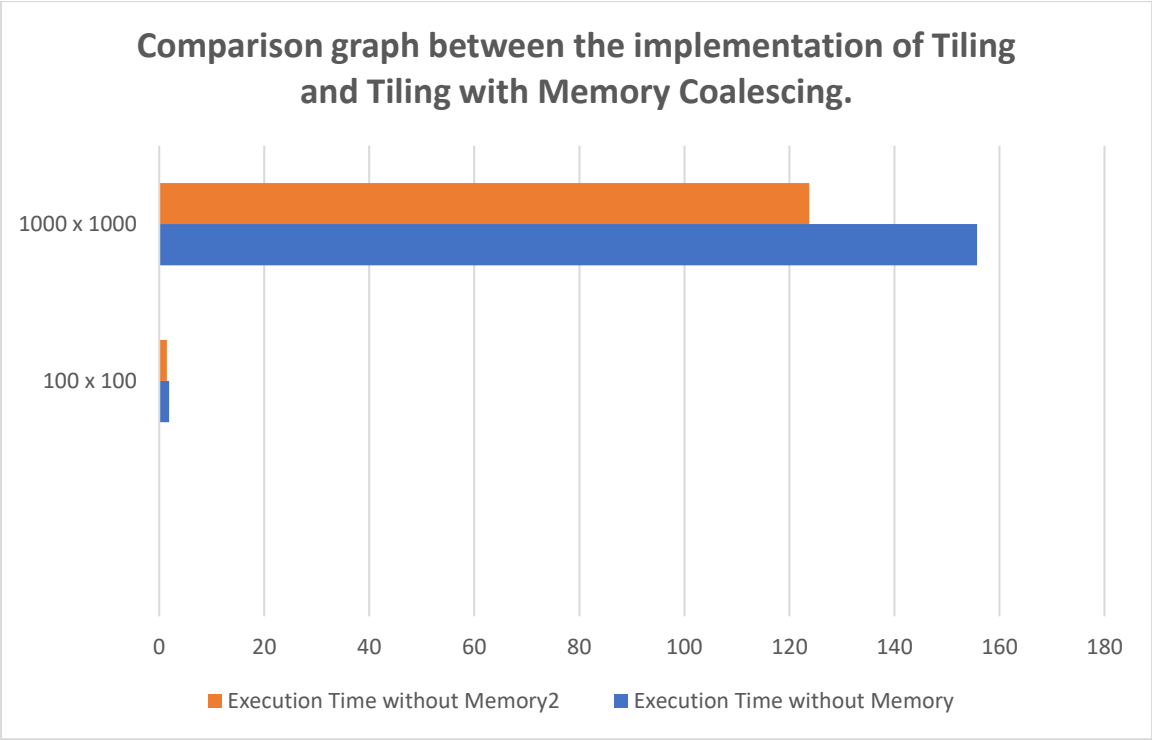


The below graph compares the execution time with and without memory transfer.



Below is the table comparison graph between the implementation of Tiling and Tiling with Memory Coalescing.

Matrix Size	Execution Time without Memory Transfer (milliseconds) in Tiling implementation	Execution Time without Memory Transfer (milliseconds) in Tiling implementation with Memory Coalescing
100 x 100	1.867744	1.43942
1000 x 1000	155.755325	123.81347
10000 x 10000	10290.892578	9670.655273



Below is the comparison graph for matrix size 1000 x 1000 for all the four questions.