CS6023: GPU Programming
Assignment 2

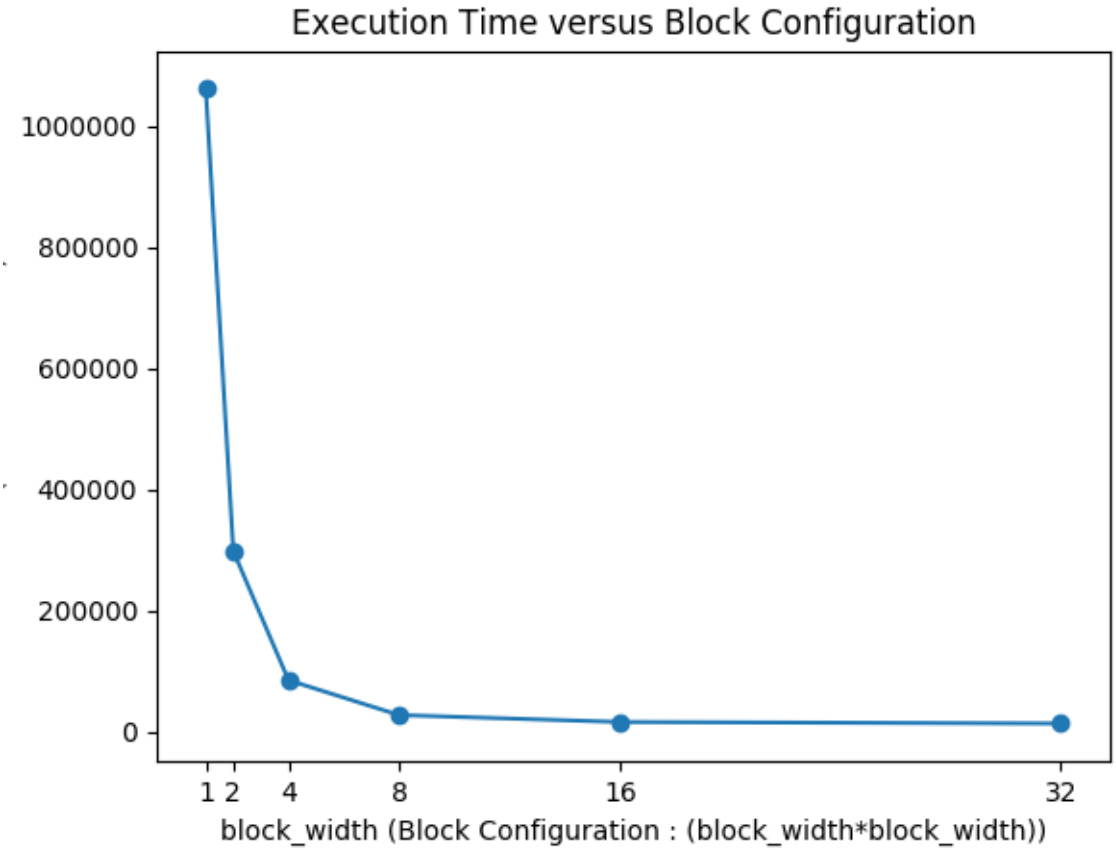
Abhishek Nair
EE16B060

Question 1

In this question we implement matrix multiplication using only global memory of the GPU. The two input matrices are loaded into the GPU and their product is calculated. The configuration of threads per block is 16*16. The number of blocks are chosen such that each thread computes exactly one element of the output matrix. Therefore each thread multiplies one row of the first input matrix with one column of the second input matrix. The work is divided among the threads in two different ways. In the first configuration, the fastest varying index is .y while in the second case the fastest varying index is .x . There is a significant variation in performance between the two configurations. The average execution times of the two configurations are as follows:
Configuration 1: 85389.12344 milliseconds
Configuration 2: 15069.998 milliseconds
The second configuration performs more than 5.5 times better than the first configuration. This is because, in the second configuration, threads in the same warp are accessing the same row in A and elements belonging to the same row in B i.e. they access elements belonging to the same burst section. On the other hand, threads in the same warp in the first configuration are accessing the same column of B and elements in the same column of A, which need not belong to the same burst section. Therefore the second configuration has a lower execution time than the first configuration. All kernels run in the following questions use the second configuration for dividing work among the threads.

Question 2

In this question we vary the number of threads per block and blocks per grid in the code written for question-1 and then try to analyse the run times to identify the optimal configuration. The number of threads per block and block per grid are varied in such a way that each thread computes only one element of the output matrix. The plot of the execution time of the kernel versus the threads per block is shown below. The plot is obtained for only 6 data-points due to resource constraints on the GPU.



It appears that the 32*32 configuration is optimal.

Question 3

In this question we use shared memory to speed up the matrix multiplication. Each thread computes one specific value of the product matrix. If the size

of the block is n*n then it requires n rows of the first input matrix and n columns of the second input matrix to calculate all the entries in the product matrix. All the threads co-operatively load the required data into the shared memory. Let the number of columns in A and the number of rows in B be n and the block size be B*B. then thread (i, j) in the block loads the j^{th}, $j + B^{th}$, $j + 2 * B^{th}$, etc elements of the $(blockIdx.y * B + i)^{th}$ row in matrix A and the i^{th}, $i + B^{th}$, $i + 2 * B^{th}$, etc elements of the $(blockIdx.x * B + j)^{th}$ column in matrix B.
When run with input matrix sizes of 32*32, the execution time is 0.156 milliseconds . However when run with input matrix size of 8192*8192 the compiler throws an error. The error arises due to the fact that we are loading entire rows and columns into the shared memory and the shared memory does not have enough space to store all the data. The size of the shared memory is 48 KB. For block sizes of 16*16, we are loading the shared memory with 16 rows and 16 columns. This amounts to a total of $2 * 16 * 8192 * 8$ bytes of memory which is 2 MB of data.

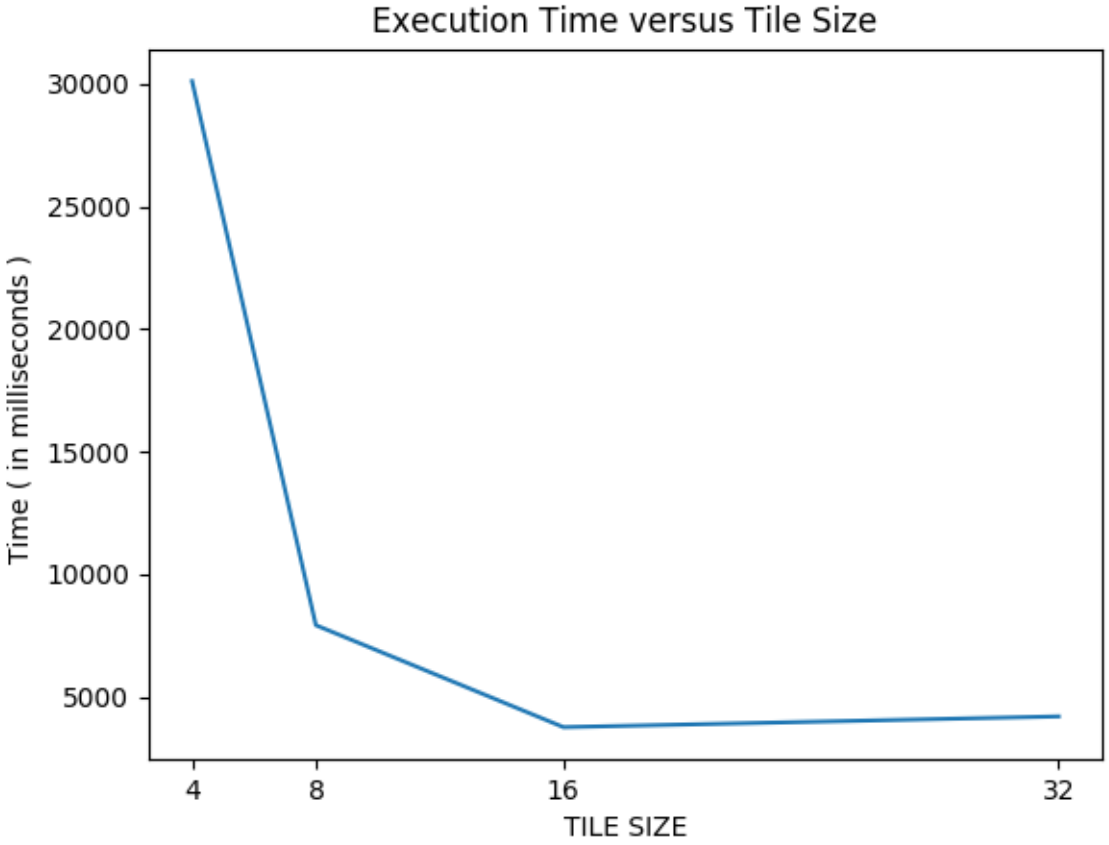
Question 4

Here we implement tiling to use shared memory efficiently. Tiling consists of loading parts of a matrix into shared memory, using the loaded elements to compute the partial sums of the elements of the product matrix and then loading a new set of elements into shared memory. Tiling allows us to use shared memory efficiently even if the input matrices are very large because the amount of data loaded into shared memory remains constant irrespective of the data size. When matrix multiplication is carried out using tiling, the average run time is 3772.175 milliseconds. Therefore it can be seen that a speed-up of upto 4 times can be achieved over the case in which shared memory is not used (question-1). The main bottleneck to performance in matrix multiplication is the time taken to access data. By using shared memory, we can make memory access orders of magnitude faster and therefore reduce the execution time drastically.

Question 5

In this question we vary the tile width used in the previous example to identify the optimal tile width to be used. The average execution time of the kernel for each tile width is given below along with the plot of the execution times.

Execution Time (in milliseconds) for tile size 4*4 : 30108.75
Execution Time (in milliseconds) for tile size 8*8: 7928.65
Execution Time (in milliseconds) for tile size 16*16 : 3772.17
Execution Time (in milliseconds) for tile size 32*32 : 4209.56



Therefore it can be inferred that the optimal tile size is 16*16.

Question 6

Two rectangular matrices with dimensions 4096*8192 and 8192*16384 are multiplied together using only global memory. The average execution time is 18275.019 milliseconds.