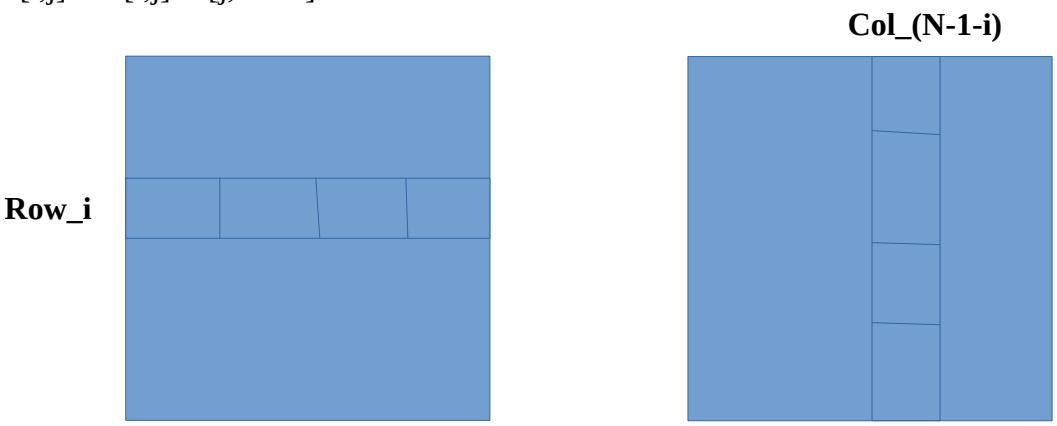
HPCA PART-B ASSIGNMENT REPORT

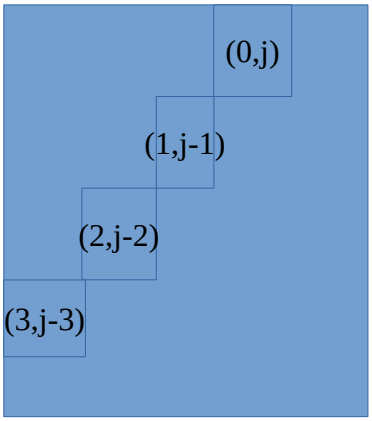
The given matrix multiplication can be divided into two steps:

1) First elementwise multiplication of a row (say row i) of matrix A , with that of column N-1-i of matrix B.(Here N is the dimension of the square matrices A and B) and storing the result in $A_{NxN.}$ A[i,j] = A[i,j]*B[j,N-1-i].



2) Summing up the elements of matrix A diagonally, and storing the values in the **output** array. There are 2*N-1 diagonals of an NxN matrix.

A single GPU thread computes the sum of the marked elements.



This implementation is followed in this assignment as a backbone algorithm.
The reason for using this algorithm is because it allows high parallelisation of data processing which can be leveraged through GPU threads quite effectively.
The above mentioned steps are implemented separately by calling two different kernels sequentially. Details provided below.

Specifications of the Grid and Blocks for step 1:

1. Block dimension = 16 by 16. Grid dimension = (N/16) by (N/16).
2. One single GPU thread is assigned the job of multiplying two numbers, one each from matrix A and matrix B. A[i,j] = A[i,j] * B[j, N-1-i]. (as shown in step 1 above).
3. Total number of GPU threads equals the number of elements in matrix A(or B), i.e. no. of threads = N*N.

4. Copies of matrices **matA** and **matB** are sent to Device before this step.

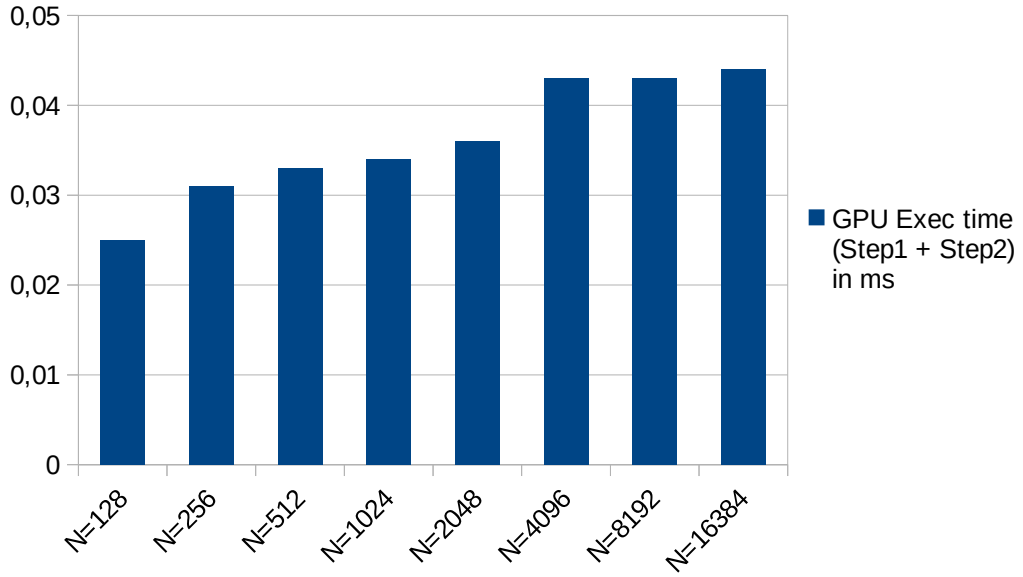
Specifications of the Grid and Blocks for step 2:

1. Dimension of Blocks = (x,y) = (16,1).
2. Demension of Grid = (x,y) = (2*N/16 , 1).
3. Total number of threads = 2*N.
4. Each thread calculates the sum of elements of a particular diagonal of the matrix A (obtained from step 1) and writes its value in **output** array at a particular index. Since there are 2*N-1 diagonals for any NxN matrix and we have 2*N threads, 1 thread is redundant and does not compute anything.
5. Rest of the threads store their results in the array **output** of size 2*N-1.
6. **Output** array is copied back from Device to Host after computaion.

Table showing the time consumption of GPU implementation with different input sizes and SpeedUps achieved wrt the reference implementaion:

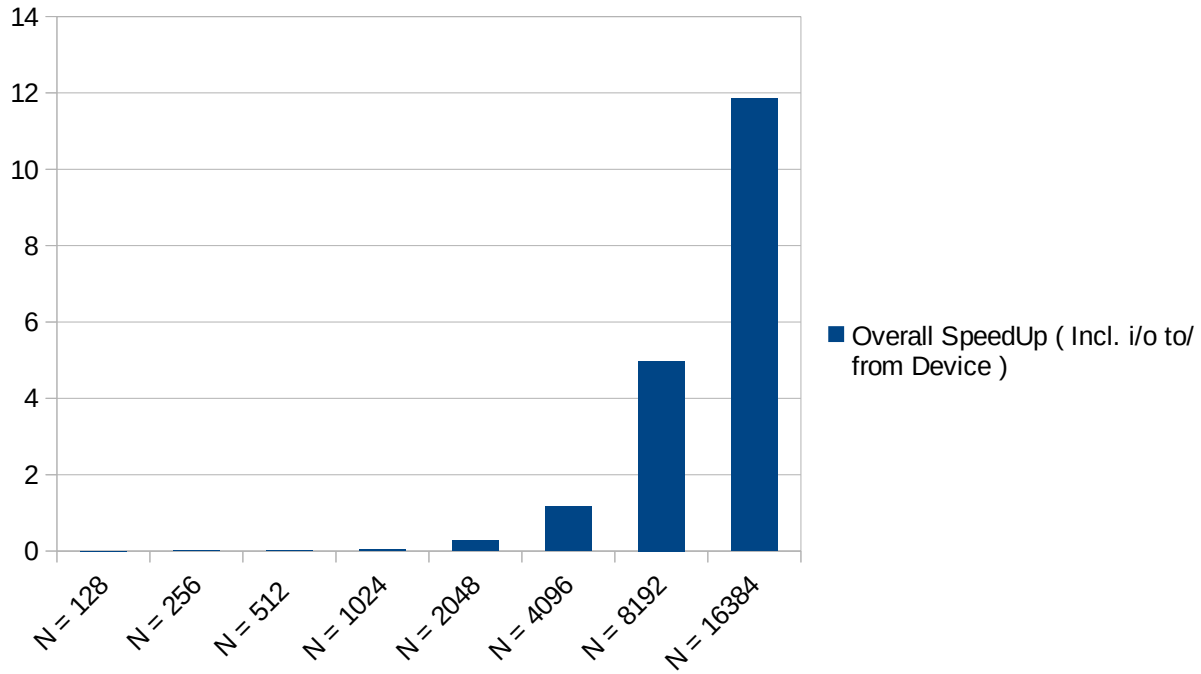
Input Size (N)	**Reference Time (ms)**	**Time for calculating matrix C in GPU(step 1) (ms)**	**Time for producing the *output* matrix in step 2 (ms)**	**Total Execution time (incl. Movement of data to and from the Device) (ms)**	**Speed Up (negating the movement of data to and from the Device)**
128	0.275	0.022	0.003	270.027	14.47
256	0.795	0.027	0.004	194.169	25.64
512	1.832	0.030	0.003	184.268	55.51
1024	10.28	0.031	0.003	216.264	302.35
2048	61.15	0.032	0.004	213.842	1698.61
4096	299.02	0.039	0.004	256.098	6953.95
8192	1973	0.039	0.004	395.905	45883.72
16384	7958.48	0.039	0.005	671.782	180874.55

One interesting observation is that the **GPU execution time** (does not include i/o to/from Device mem) remains approximately the same for the entire range of input sizes starting from N=128 to N=16K. This can be attributed to the parallel execution of threads over the domain of input data.



Below Table showing the overall speed up of the GPU implementation:

	N=128	N=256	N=512	N=1024	N=2048	N=4096	N=8192	N=16384
SpeedUp (Overall)	0.001	0.004	0.01	0.047	0.286	1.17	4.98	11.85



Other details:

Jobs were submitted at cl-gpusrv1. All executions were done there.