QUESTION 1:

Let us consider the 2D convolution operation for a 2D Matrix of floating point numbers. It is a neighborhood operation where each output element in the matrix is the weighted sum of a collection of neighboring elements of the input matrix. The weights used in the weighted sum are typically stored in an array called the convolution mask. This mask is applied across each element of the input to yield a convolved output. Let us focus on a specific convolution operation, namely average convolution. This operation considers a 3x3 mask and for every element A[i][j], considers the nearest 8 neighbors contained in the 3x3 mask centered at the element A[i][j], takes the average and stores in the output matrix B[i][j].

Implement a CUDA program which takes as input i) the number of test cases and for each test case i) the integer n, iii) the values of each element of an n x n matrix; apply the average convolution operation and produce an n x n output matrix. Refer to the following input output example.

```
Input
1
3
3.0 3.0 3.0
3.0 3.0 3.0
3.0 3.0 3.0
 Output
 1.3  2.0  1.3
 2.0  3.0  2.0
 1.3  2.0  1.3
```

Note for boundary elements, assume a padding of 0.0. In the above example, for element input[0][0], placing the 3x3 mask with its center at input[0][0], produces
output[0][0] = 0.0/9 +0.0/9 +0.0/9 + 0.0/9 + input[0][0]/9 + input[0][1]/9 + 0.0/9 + input[1][0]/9 + input[1][1]/9
=0.0/9 +0.0/9 +0.0/9 + 0.0/9 + 3.0/9 + 3.0/9 + 0.0/9 + 3.0/9 + 3.0/9
= 1.3.

Further note that the above input-output was a representative example depicting what the convolution operation is. Expect sample input matrices to be of the order NxN where N =512,1024,2048 etc. Your task would be to generate random floating point matrices of this order and test your implementation while adhering to the input format discussed above. The code should be general enough to handle large matrices. Accordingly, your implementation should be able to automatically select suitable grid and block dimensions for processing the matrix. Ensure that you print your output matrices using %.2f format as taught in the tutorial.

QUESTION 2:

Recall the shared memory implementations of matrix transpose operation taught in the lecture videos. The lectures focused primarily on optimizations involving shared memory with square tiles. Consider a more general scenario where tiles are rectangular in nature. Implement a transpose kernel which performs the following memory operations:

A. Write to a shared memory row with each warp to avoid bank conflicts.
B. Read from a shared memory column with each warp to perform a matrix transpose.
C. Write to a global memory row from each warp with coalesced access.

Implement a CUDA program which takes as input i)the number of test cases ii) for each test case the value of N and iii) N lines of floating point values where each line contains N space separated floating point numbers. This represents the NxN input matrix. Assume that N is divisible by 32. Your program should print the transposed matrix as N lines of N space separated floating point values for each test case.

Again expect N to be of the order 512,1024,2048,etc. Your task would be to generate random floating point matrices of this order and test your implementation while adhering to the input format discussed above. The code should be general enough to handle large matrices. Your implementation should be generic enough to identify suitable grid block dimensions and shared memory dimensions. The shared memory should be rectangular in nature and of the dimensions W x (2*W). Depending on what GPU you are working on, identify first the maximum shared memory size per SM. Accordingly decide for your given input matrix what the value of W should be. Ensure that you print your output matrices using %.2f format again.

QUESTION 3:

Consider the transposeCoalesced implementation taught in the lecture videos. Implement a version of the code that uses dynamic shared memory instead of static shared memory. For designing a generic implementation, it makes sense to use dynamic shared memory rather than static shared memory where you have to redefine the macros for the shared memory size for every GPU architecture family. Dynamic shared memory can be specified as the third argument while launching the kernel from the host program. For example, the launch parameters can be as follows.

```
transposeCoalesced<<<grid,block,sharedMemElements*sizeof(float)>>>
```

You can query the maximum size of the GPU shared memory and accordingly set up shared memory dynamically for any GPU architecture. Note for this case, the shared memory inside the kernel must be declared as an unsized 1D array as follows.

```
extern __shared__ float tile[];
```

Accordingly implement your transpose operation using 1D Shared Memory Tiles with appropriate padding instead of standard 2D Shared Tiles with padding to avoid bank conflicts. Think carefully on what the dimensions of the shared memory should be and how it should be accessed.

Implement a CUDA program which takes as input i)the number of test cases ii) for each test case the value of N and iii) N lines of floating point values where each line contains N floating point numbers separated by spaces. This represents the NxN input matrix. Your task would be to generate random floating point matrices of this order and test your implementation while adhering to the input format discussed above. The code should be general enough to handle large matrices. Assume that N is divisible by 32. Your program should print the transposed matrix as N lines of N space separated floating point values for each test case. Again expect N to be of the order 512,1024 ,2048,etc. Your implementation should be generic enough to identify suitable grid block dimensions and shared memory dimensions as discussed in the previous question.

<u>QUESTION 4:</u>

Let us consider the dot product operation of two vectors A and B, each of dimension N. Implement a CUDA program which uses the 1D CUDA kernel dotproduct() launched with parameters <<<m,k>>> such that
i) m * k = N,
ii) each block of k threads of dotproduct() is responsible for computing the pairwise products of k successive elements in A and B and finally returning the partial sum of those k products. Use shared memory while computing partial sums.

Implement and call the reduction kernel repeatedly on these k partial sums until you are left with p partial sums where p <1024. For summing up elements less than 1024, it makes sense to execute reduction on the CPU, since the GPU is highly underutilized in this case. Once the p partial sums have been calculated, implement a sequential CPU function that will reduce these p elements to yield the final scalar value of the dot product. Note for the reduction GPU kernel, implement a version that -i) avoids branch divergence, ii) avoids shared memory bank conflicts, iii) performs first add during load and iv) uses loop unrolling.

Implement a complete CUDA program which takes as input i) the number of test cases ii) for each test case the value of N and iii) 2 lines of N space separated values representing the 2 vectors A and B. Assume that N is a power of 2 and is very large (of the order in the range 2^18 -- 2^23.). Your task would be to generate random floating point arrays of this order and test your implementation while adhering to the input format discussed above. The code should be general enough to handle large matrices. Your program should print the required dot product i.e. a scalar value for each test case.