Problem 1

a :

0.000000 0.708073 0.826822 0.019915 0.572750

0.025133 0.555284 0.928855 0.087783 0.414230

0.098005 0.396936 0.987775 0.197092 0.264333

0.211290 0.248949 0.997658 0.336853 0.138128

0.353599 0.126201 0.957511 0.493015 0.048302

b ;

1.000000 0.291927 0.173178 0.980085 0.427250

0.974867 0.444716 0.071145 0.912217 0.585770

0.901995 0.603064 0.012225 0.802908 0.735667

0.788710 0.751051 0.002342 0.663147 0.861872

0.646401 0.873799 0.042489 0.506985 0.951698

c :

1.000000 1.000000 1.000000 1.000000 1.000000

1.000000 1.000000 1.000000 1.000000 1.000000

1.000000 1.000000 1.000000 1.000000 1.000000

1.000000 1.000000 1.000000 1.000000 1.000000

1.000000 1.000000 1.000000 1.000000 1.000000

Problem 3

For a vector addition, assume that the vector length is 2000, each thread calculates one output element, the thread block size is 512 threads and the entire computation is carried out by one grid. How many threads will be in the grid? How many warps do you expect to have divergence due to the boundary check on the vector length?

There are 2048 threads in the grid, one warp is totally out of boundary.

Problem 4

Consider matrix addition where each element of the output matrix is the sum of the corresponding elements of the two input matrices. Can one use shared memory to reduce the global memory bandwidth consumption? Hint: analyze the elements accessed by each thread and see if there is any commonality between threads.

No, because each elements is used only once, and there is no input shared by the threads.

Problem 5

Consider the following program that transposes the tiles of a large matrix. The size of the tiles is BLOCK_WIDTH×BLOCK_WIDTH and the dimensions of the matrix A are guaranteed to be multiples of BLOCK_WIDTH, which is known at compile time. BLOCK_WIDTH is in the range from 1 to 20.

```
dim3 blockDim(BLOCK_WIDTH, BLOCK_WIDTH);
dim3 gridDim(A_width/BLOCK_WIDTH, A_height/BLOCK_WIDTH);
BlockTranspose<<<gridDim, blockDim>>>(A, A_width, A_height);
__global__ void BlockTranspose( float* A, int A_width, int A_height)
{
  __shared__ float blockA[BLOCK_WIDTH][BLOCK_WIDTH];
  int baseIdx = blockId.x * BLOCK_WIDTH + threadId.x;
  baseIdx += (blockId.y * BLOCK_WIDTH + threadId.y) * A_width;
  blockA[threadId.y][threadId.x] = A[baseIdx];
  A[baseIdx] = blockA[threadId.x][threadId.y];
}
```

(a) For which values of BLOCK_WIDTH in the range from 1 to 20 will this kernel function execute correctly on the device?

5, because if the BLOCK_WIDTH is 6, the block size will be 36 and it is more than a warp, other wraps may read the invalid data in shared memory before the wrap write correct data into memory.

(b)If the code does not execute correctly for all values of BLOCK_WIDTH, suggest a correction to make the code work for all values of BLOCK_WIDTH.

Add `syncthreads();` *after* `blockA[threadId.y][threadId.x] = A[baseIdx];`