ASSIGNMENT 3:

The assignment is aimed at applying tiling to optimize CUDA programs, and corresponds to implementing and using matrix transposition and dense matrix-matrix multiplication.

Please consult the lecture notes L5-LoopparI.pdf (Block Tiling and Coalesced Accesses).
For all programming tasks, please provide a Makefile with targets for compiling and running each programming task. Each programming task should generate the input data (e.g., filled with random numbers) and should validate the CUDA execution against a sequential implementation that runs on CPU on one core. For each subtask your program should output whether the execution is valid or not and the CUDA kernel runtime (and the corresponding runtime for the sequential program).

Task 1) Assuming a matrix A with M rows and N columns, the pseudo-code for computing the matrix transpose of A into matrix B (with N rows and M columns) is:

```
1. for i from 0 to M-1   // parallel
2.   for j from 0 to N-1 // parallel
3.     B[j, i] = A[i, j]
```

Task 1.a) Implement a sequential version of transpose that runs on the CPU, i.e., create a matrix filled with random floats and re-write the pseudo-code above in C.

Task 1.b) Bonus: write an OpenMP version of the program that executes the outer loop (of count M) in parallel. You will have to compile this version with `gcc -fopenmp` rather than with `nvcc`.

Task 1.c) Implement a “naive” transpose in CUDA, i.e., write a two-dimensional CUDA kernel that exploits both N and M dimensions of parallelism and which performs the transposition much in the way shown in the pseudo-code.

Task 1.d) Implement a CUDA optimized transposition that uses tiling in shared memory in order to achieve memory-coalesced accesses for both the reads from the input array A and for the writes to the result array B.

Task 2) *This task aims to demonstrate the usefulness of matrix transposition.*
Assume that both A and B are matrices with N rows and 64 columns. Consider the pseudo-code below:

```
1. for i from 0 to N-1          // outer loop
2.   accum = A[i,0] * A[i,0];
3.   B[i,0] = accum;
4.   for j from 1 to 63         // inner loop
5.     tmpA   = A[i, j];
6.     accum  = sqrt(accum) + tmpA*tmpA;
7.     B[i,j] = accum;
```

Task 2.a) Reason about the loop-level parallelism of the code above:

- Assume that `tmpA` and `accum` are declared (and initialized) before the loop. Why is the outer loop not parallel? What technique can be used to make it parallel? Re-write the code such that the outer loop is parallel, i.e., no loop-carried dependencies exist.
- Is the inner loop parallel? Explain why or why not.
- Can the inner loop be re-written as a composition of parallel operators? Explain why.
- If line `6` is re-written as `accum = accum + tmpA*tmpA,` would it be possible now to re-write the inner loop as a composition of bulk operators?

Task 2.b) Bonus: write an OpenMP version of the program that executes the outer loop (of count N) in parallel. You will have to compile this version with `gcc -fopenmp` rather than with `nvcc`.

Task 2.c) Implement quickly a straightforward CUDA version of the program above, in which the first loop of index `i` and count `N` is executed in parallel, i.e., corresponds to a one-dimensional CUDA kernel, and the second one is executed sequentially, i.e., it is part of the kernel code.

Task 2.d) Rewrite quickly the CUDA program such that all accesses to global memory are coalesced, i.e., the new program reads from the transpose of A, and computes the transpose of B:

- transpose A in A', using the optimized CUDA implementation of Task I.1.
- write a CUDA kernel that implements a modified version of the pseudo-code above that uses A' instead of A and computes B' (the transpose of B), instead of B.
- finally, after the execution of the CUDA kernel, transpose B' to obtain the original result B.

The modified program (CUDA transpositions included) has about two times the number of global memory accesses of the original program. ***Does it run faster or slower than the original, and by how much (for a suitably large N)?***

Task 3) Refers to dense matrix-matrix multiplication (see lecture notes); see pseudo-code below that assumes that matrices A, B and C have sizes MxU, UxN and MxN respectively:

```
1. for i from 0 to M-1        // first outer parallel loop
2.   for j from 0 to N-1      // second outer parallel loop
3.     float tmp = 0.0;
4.     for k from 0 to U-1    // innermost loop executes sequentially
5.       tmp = tmp + A[i,k]*B[k,j];
6.     C[i,j] = tmp;
```

Task 3.a) Implement a sequential version of the pseudo-code above.

Task 2.b) Bonus: write an OpenMP version of the program that executes the outer loop (of count M) in parallel. You will have to compile this version with `gcc -fopenmp` rather than with `nvcc`.

Task 3.c) Implement a naïve CUDA version that straightforwardly implements the pseudo-code above. (Uses a two-dimensional kernel/grid corresponding to the two parallel outer loops.)

Task 3.d) Implement a CUDA optimized version that uses tiling in shared memory in order to reduce the number of global-memory accesses by a factor of TILE-size – see lecture notes. (Uses a two-dimensional kernel/grid corresponding to the two parallel outer loops.)

Measure and compare the various running times. How many GFlops does the naïve and optimized CUDA versions achieve?