Clarifications for Ex2/Hw1.
Due date extended to SUN, 31 Jan, 11:59 PM.

For part 2 - Implement kernel functions:
a. matrix-add

The maximum size of matrices which can be added will depend on the specific GPU device you have.
However, even an older lower-end consumer-grade device should be able to handle adding float matrices of 4096x4096.
Try to go that large, or larger.
HINT: test your row-wise and col-wise thread access patterns on a much smaller test matrix to ensure it is behaving correctly first.

b. dot-product

Implement the method of computing the partial dot-product sums in each kernel thread
Do both the multiplies and additions for a partial section of the full vectors in each kernel thread, and then return the "partial dot-product sums" back to the CPU which only does the final summation of the partial dot-products, using associative property. This will increase the amount of work being done in each kernel thread, which is desireable.
For example, for vector dot-product a.b
= a1*b1 + a2*b2 + a3*b3 + a4*b4 +a5*b5 + a6*b6 + a7*b7 + a8*b8 + a9*b9 + ...
= (a1*b1 + a2*b2 + a3*b3) + (a4*b4 +a5*b5 + a6*b6) + (a7*b7 + a8*b8 + a9*b9) + (... grouped by 3s, or Ns in general)

(You can adjusted the number of elements computed in each kernel thread partial dot-product sum)

You should be able to handle vectors of length 2^25 or larger (depending on your device's available memory).
[This is not intended to be an optimal solution for parallelizing the dot-product, but it will serve as a reference for improved versions we'll be implementing soon.]

c. D/S-GEMV

For this routine, have each kernel thread compute the value of one output element in y, which includes the dot-product of a matrix row and a vector for the Mv term.
y = aMv +bw, where M is MxN, v is Nx1, and w is Mx1, and a, b are scalar floats, and result vector is Mx1
thus use M threads total.

try this with M=64,128,256,1024,4096.

COMMENTS: None of these are expected to be optimal or highly-performant kernels at this point! These are some of the first GPU kernels you've ever written (probably) and the only goal for this assignment is to get the kernels to run and produce correct answers. You do not need to spend much time thinking/worrying about how to optimized the kernels, since we still have not covered the necessary theory and methods you need to do that.
We will learn how to improve the implementations as the class progresses! And we'll compare the performance of future improved kernel variants to these "naive" versions.