Welcome to homework #4! This assignment is about less parallel applications in CUDA. You'll implement three of them, and do a bit of analysis and critical thinking about them. I've once again ordered them *roughly* from easiest to hardest (again, in my opinion!), so do with that as you will. Each problem is worth a total of 5 points (please let me know if I messed up the math), divided as described below.

Our goal with this assignment is to get into some more complex CUDA programs. Note that most of these problems will require you to write multiple kernels, or possibly do interesting things with a single kernel.

Instructions for turning this in (aka, how to make my life easy 😉):

- Make sure you have all of these ready to go, with a separate directory for each program:
    - A file with a list of modules to load
    - Your makefile (ideally, all I need to do is load the modules and run make and be good to go)
    - Any sbatch scripts you want to send as examples for how to run your program
    - Any instructions or notes about your code (what your makefile targets do, etc.)
    - Your code!
- Compress all that (zip or tar) into a single file with directories for each program, name the file <your name or username>.<tar/zip/gz>
- Upload to Canvas

1. Implement a 2-dimensional AXPY (A times X Plus Y – usually a vector operation, not a matrix operation, but I couldn't come up with a better name) with CUDA.
    a. [1 point for doing this with CUDA] Create two arrays of floats, x and y, of size NxN (where N is a parameter), and fill them with 1.0 and 2.0, respectively.
        i. [1 point for accepting input] Your program should accept input for the size of x and y like this (don't worry about making this robust): `./axpy -n <N>`
    b. [1 point for doing this with CUDA] Compute the sum `y[i][j] = a*x[i][j]+y[i][j]`, where `a` is a scalar of your choice.
    c. [1 point for doing this] Copy the results back to the CPU and verify your results.
    d. [1 point for doing this] Run your program with Nsight Compute and look at the statistics it shows. Pick one set of statistics and write a sentence or three on what that says about your implementation.
    e. [1 BONUS point for doing this] Pick a second set of Nsight Compute statistics and write a sentence or three about what those say as well.

2. Implement a heat conduction stencil with CUDA.
    a. Initialize two arrays of floats, A and Anew, of size NxN (where N is a parameter), to all zeroes, except for the first row of A and Anew, which should be all 100.

i. [1 point for taking input correctly] Your program should accept input for the size of A and Anew (N), the error tolerance (tol), and maximum number of iterations (max_iter) like this (don't worry about making it robust):

```
./heat -n <N> -tol <t> -max_iter <iter>
```

b. [2 points for doing this correctly with CUDA] Calculate a Laplacian heat transfer over Anew using a "four-point stencil". Note that the loop bounds go from 1 to N-1, so you don't need to worry about edge cases. (Think of row 0 of A, which will always be 100, as your heat source, and row N-1 (and the sides) of A, which will always be 0, as your heat sink.) Pseudocode:

```
while (err > tol && iter < max_iter) {
    err = 0.0;
    for (int i = 1; i < N-1; i++) {
        for (int j = 1; j < N-1; j++) {
            Anew[i][j] = (A[i+1][j] + A[i-1][j] +
                A[i][j-1] + A[i][j+1])/4;
            err = max(err, abs(Anew[i][j] - A[i][j]));
        }
    }
    // yes you do have to do it like this, think about
why
    for (int i = 1; i < N-1; i++) {
        for (int j = 1; j < N-1; j++) {
            A[i][j] = Anew[i][j];
        }
    }
    iter++;
}
```

c. [1 point for producing CSVs] Every 1000 iterations, output the contents of Anew to a CSV file. This file should have one row of Anew per line, with each number followed by a comma. It should be named `heat_<iter count>.csv`.

d. [1 point for producing *correct* CSVs] Like we did with Mandelbrot, I'll provide a python script to visualize the heat transfer. The script will be called `heat.py`, and will produce images named `heat_<iter count>.png`. You can either scp all these to your local machine for viewing, or set up X11 forwarding so you can use the `display` command on Talapas.

3. Implement matrix multiplication with CUDA.

a. Create an NxM matrix of floats, A, and an MxN matrix of floats, B, where N and M are both parameters, and fill them both with 1.0. Create an NxN matrix C, where we'll store the results.

    i. [1 point for taking input correctly] Your program should take input for N and M like this (don't worry about making this robust):

       ```
       ./matmult -n <N> -m <M>
       ```

b. [1 point for doing this with CUDA] Compute the product of A*B and store it in C. Note that

   ```
   C[i][j] = sum over all k (0->M) of A[i][k]*B[k][j]
   ```

c. [1 point for doing this with CUDA] Check your results. (Since A and B are both full of 1s, C should be full of Ms.)
d. [1 point for doing this with CUDA] Repeat (b), but with a different memory layout for the matrix B. Transpose B in memory (so that, instead of looping over columns, we're looping over rows), so that the math for computing the product A*B is now

   ```
   C[i][j] = sum over all k (0->M) of A[i][k]*B[j][k]
   ```

   Look at the performance of (b) and (d) (both including and not including the code that does the transpostion). Are there any performance differences (again, just timing results is enough)? Write a sentence or three about why or why not.
e. [1 point for doing this with CUDA] Repeat (b), but with a different memory layout for the matrix A. Transpose A in memory (so that, instead of looping over rows, we're looping over columns), so that the math for computing the product A*B is now

   ```
   C[i][j] = sum over all k (0->M) of A[k][i]*B[k][j]
   ```

   Look at the performance of (b) and (e) (both including and not including the code that does the transposition). Are there any performance differences (again, just timing results is good)? Write a sentence or three about why or why not, and what you might conclude about the general layout of GPU memory from these experiments. (I.e., is it row major, or column major?)