Welcome to homework #3! This assignment is about "embarassingly parallel" applications in CUDA. You'll implement the same three programs as you did in homework #1. Feel free to use that as a starting place. Each problem is worth a total of 5 points (if I've messed up the math, please let me know), divided as described below.

Our goal with this assignment is give you an intro to CUDA and its environs.

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 Monte Carlo pi calculation using CUDA.
    a. [1 point for generating N random points using cuRAND] Generate N random 2-dimensional points (where N is a parameter) in the range [0, 1] in both the x and y dimensions.
        i. [1 point for taking input correctly] Your program should accept input for the number of points like this (don't worry about making this robust, it's fine if you don't set default values or if your program breaks without it, that's not our goal here):
        ```
        ./montecarlopi -numpoints <N>
        ```
    b. [0.5 points for doing this with CUDA] For each point (x,y), calculate whether $x^2+y^2<=1$.
    c. [0.5 points for doing this with CUDA] Calculate the fraction of points for which the above is true.
    d. pi = 4*(# of points <=1)/(total # of points), approximately.
    e. [1 point for outputting something, 1 more point for outputting something *correct*] Your program should output its approximation of pi, to whatever level of precision you see fit (at least 4 decimal places though, please).

2. Implement the STREAM benchmark in CUDA.
    a. Initialize three arrays of floats (not doubles) a, b, and c to 1.0, 2.0, and 0.0, respectively. Make the size of these arrays a parameter.

        i. [1 point for taking input correctly] Your program should accept input for the size of the arrays like this (again, it's fine if this isn't robust):

```
./stream -size <n>
```

b. [1 point for outputting and writing up something, 1 more point for outputting and writing up something correct] For each of the benchmarks, time how long it takes the benchmark to run (using your favorite timer library – you may need to use a very high resolution clock to get a non-zero time, even with large arrays) and output the time. Do this again, but with a, b, and c as arrays of doubles. Do you notice any differences? Why or why not?
c. [0.5 points for doing this with CUDA] Run the "copy" benchmark: copy each element of array a into array c. (`c[i] = a[i];`)
d. [0.5 points for doing this with CUDA] Run the "scale" benchmark: multiply each element of array c by a scalar (your choice) and put the result in array b. (`b[i] = scalar*c[i];`)
e. [0.5 points for doing this with CUDA] Run the "add" benchmark: add arrays a and b and put the result in array c. (`c[i] = a[i]+b[i];`)
f. [0.5 points for doing this with CUDA] Run the "triad" benchmark: add array b to array c times some scalar (your choice again) and put the result in array a. (`a[i] = b[i]+scalar*c[i];`)
g. [1 BONUS point] Use Nvidia's Nsight Compute tool to find the memory bandwidth of each kernel. (Hint: `module load cuda/11.5.1; ncu --clock-control none ./stream -size <n>`.) Does the memory bandwidth differ depending on the data type used? Why or why not?

3. Implement the Mandelbrot set with CUDA.
    a. [2 points for doing this calculation using CUDA] For a grid from [-2.0,1.0] in the x direction and [-2.0,2.0] in the y direction, with a parameterized number of points in the x and y directions, calculate whether or not each grid point is in the Mandelbrot set, with the maximum number of iterations set to 1000. Pseudocode:

```
for each grid point P do
    x0 := x coordinate of P
    y0 := y coordinate of P
    x := 0.0
    y := 0.0
    iteration := 0
    while (x*x + y*y ≤ 2*2 AND iteration < max_iteration) do
        xtemp := x*x - y*y + x0
        y := 2*x*y + y0
        x := xtemp
        iteration := iteration + 1
```

        i. [1 point for taking input correctly] Your program should accept input for the number of points like this (don't worry about robustness here either):

```
./mandelbrot -numx <n> -numy <m>
```

b. [1 point for outputting an "image" file] Visualize the results by outputting the number of iterations required for each point to a CSV file. Your file should have one row of iteration counts per line, with each number followed by a comma. This file should be named `mandelbrot.csv`.
c. [1 point for outputting a *correct* "image" file] I'll provide a python script that will take in a file of numbers and make an image for you. The script is called mandel.py, and you can run it by doing `module load python3` followed by `python3 mandel.py`. It'll output an image called mandel.png, which you can scp to your local machine for viewing.