Michael Gibson
5/7/2023
ECE 356

Estimating Pie using GPU/Cuda

This program uses cuda in order to calculate many data variables at once within a square, and uses distance to find which of the points in the square are within the circle. Using the area of the circle from the square, we can determine pi using the circle area equation using the radius, or half the square size.

First, we have the number of threads per block and number of blocks in the grid, as well as the total number of points. This will determine what is used within the gpu to calculate data, changing the speed, and the total, changing the speed and accuracy..

```
threads_per_block = 512
num_of_blocks = 128
total = 1000000
```

These variables can be changed without ruining the code.
We also have rng, to set the rng for the variables for the gpu later, as well as the my_var, which helps us calculate pi by being the shared variable between all the threads.

```
rng_ = create_xoroshiro128p_states(threads_per_block * num_of_blocks, seed=1)
my_var = np.zeros(threads_per_block * num_of_blocks, dtype=np.float32)
```

Next we have the function pi_circle. This function is processed in the gpu using cuda, and will randomly generate a location for a point for as many times as the total value is, giving us a controlled amount of points. It then finds out how many of these points are within a circle by determining what points are within half a square's side from the center, allowing us to find the area of the circle.

```
@cuda.jit
def pi_circle(rng_, iterations, my_var):
    thread_id = cuda.grid(1)

    inside = 0

    for i in range(iterations):
        x = xoroshiro128p_uniform_float32(rng_, thread_id)
        y = xoroshiro128p_uniform_float32(rng_, thread_id)


        if x**2 + y**2 <= 1.0:
            inside += 1

    my_var[thread_id] = 4.0 * inside / iterations
```

We then calculate pi with the my var variable and, we pi_circle completes, print it out. We also get the time and print that out as well.

```
pi_circle[num_of_blocks, threads_per_block](rng_, total, my_var)
t1 = time.time()
total_time = t1-t0

print('pi:', my_var.mean())
print('time:', total_time, 'seconds')
```

Here are some different variables and answers from the code

num_threads/num_blocks	total	Computation time (in seconds)	Accuracy of result
16/16	100	0.157 seconds	%99.98100
16/16	100000	0.181 seconds	%99.99435
16/16	100000000	11.269 seconds	%99.99994
64/16	100	0.155 seconds	%99.91391
64/16	100000	0.280 seconds	%99.99521
64/16	100000000	14.991 seconds	%99.99990
16/64	100	0.244 seconds	%99.91391
16/64	100000	0.194 seconds	%99.99521

16/64	100000000	15.035 seconds	%99.99990
64/64	100	0.155 seconds	%99.99287
64/64	100000	0.221 seconds	%99.99957
64/64	100000000	26.449 seconds	%99.99997

From these results, it doesn't appear that increasing the threads or blocks is helping all that much, and in fact are being detrimental to the performance. Whether this is google collabs fault for having confusing and possibly premium service filters, or if it is just natural, it does appear that the larger the total the more accurate the results, which makes much more sense.