Project 2 Final Report

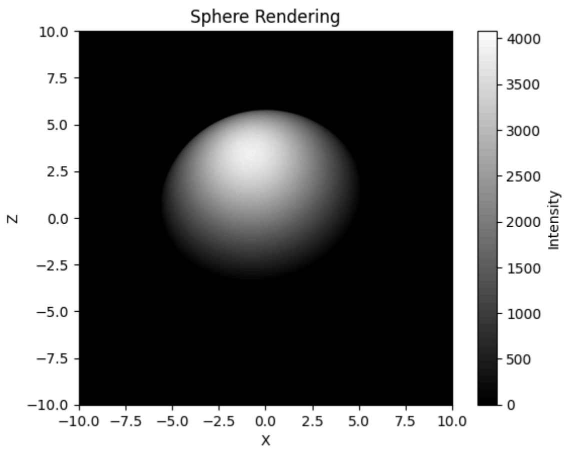
Name - Vedant Kodagi
Email address - vkodagi@uchicago.edu
Professor - A. Seigel

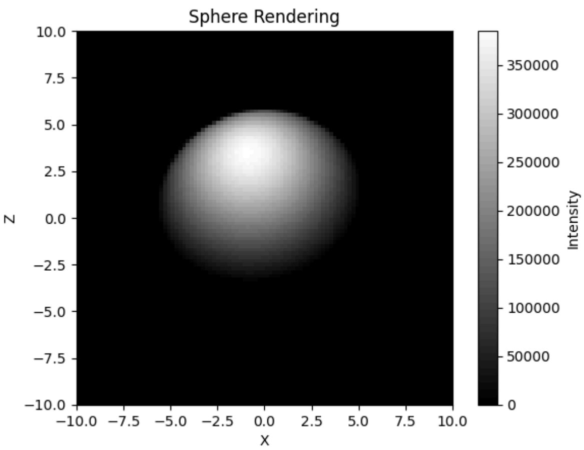
Introduction

I did not alter my algorithmic approach as discussed in the last report however, I improved my kernel runtimes by nearly 4.7 times this week. I will discuss the details of this in the optimizations section.

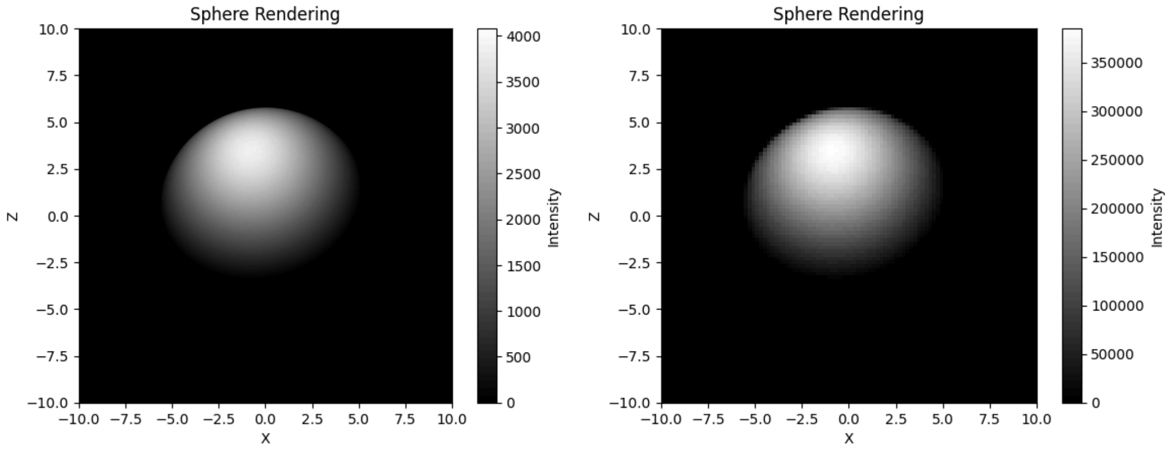
Images by different processes-

1. V100 images for 1000^2 and 100^2 grids.

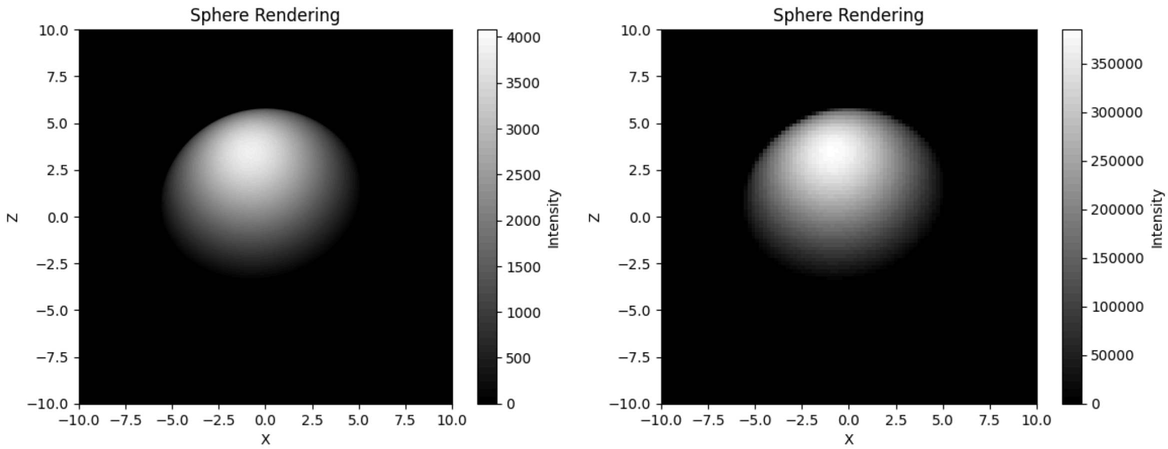




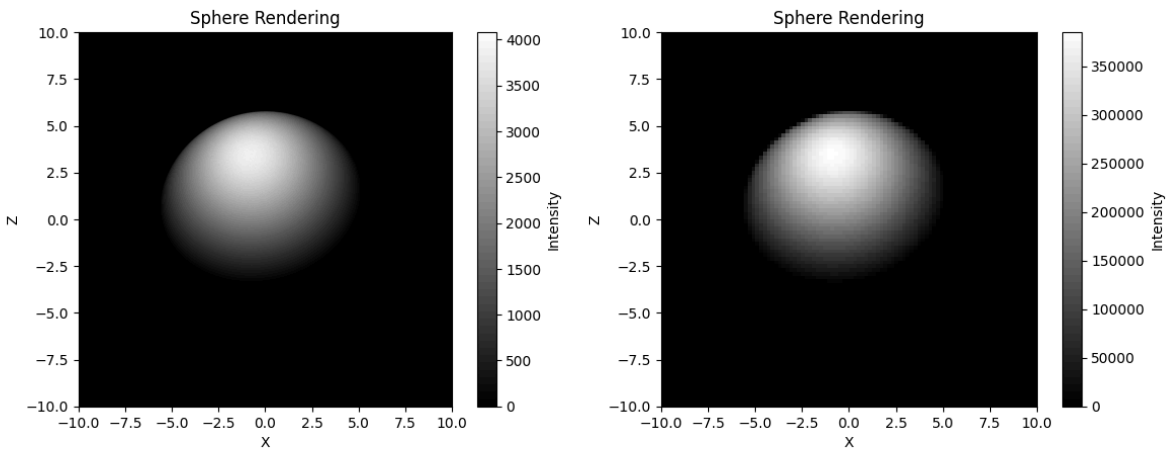
2. RTX6000 images for 1000^2 and 100^2 grids.



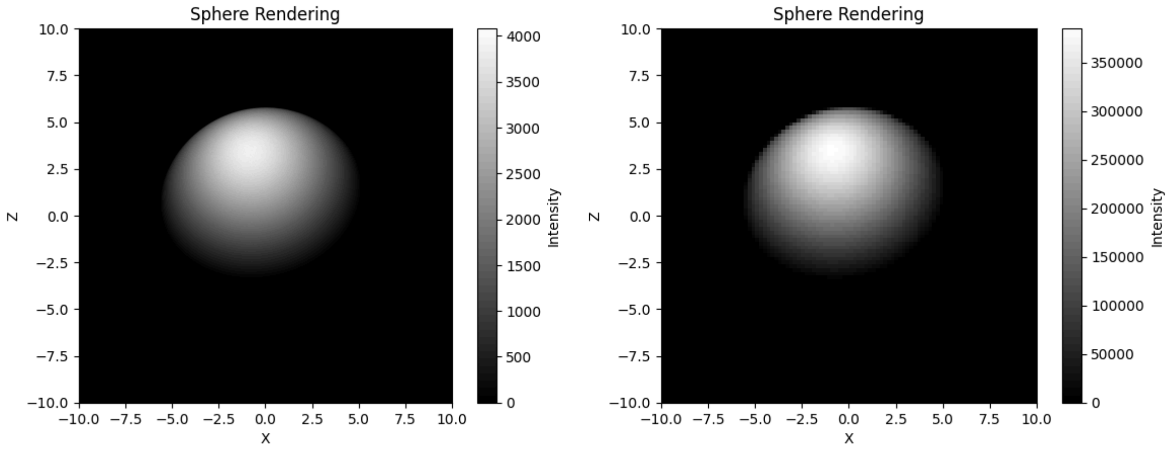
3. CPU Serial images for 1000^2 and 100^2 grids.



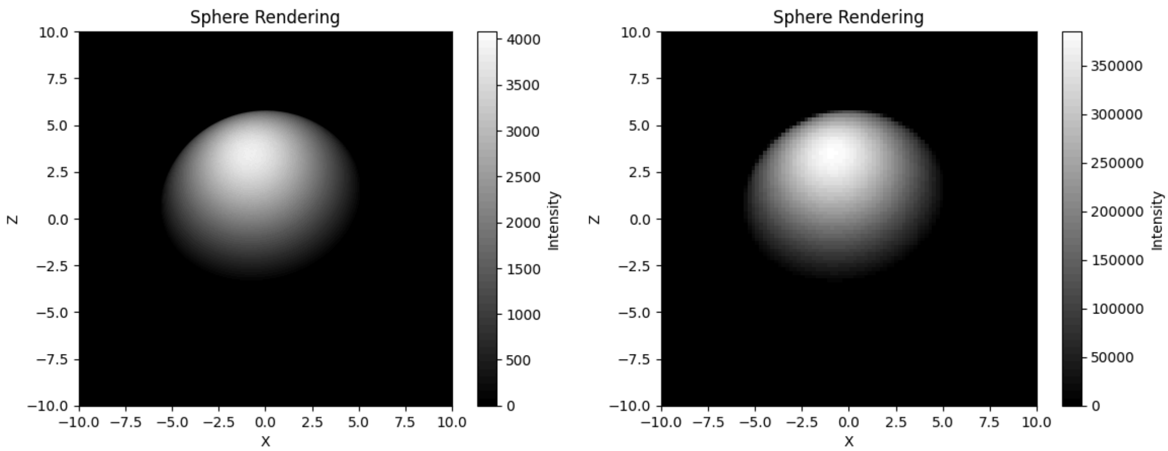
4. CPU OMP images for 1000^2 and 100^2 grids.



5. 2 V100 GPUs images for 1000^2 and 100^2 grids.



6. 4 V100 GPUs images for 1000^2 and 100^2 grids.



Performance

The fastest double-precision solution was 228.88 ms on a single A100 with a total wall time of 613 ms. Adding multiple GPUs, we got the expected speedups with 2 GPUs giving us x1.98 speedup and 4 GPUs giving us x3.85 speedup. However, it must be mentioned that there was a lot of variance in the runs. In 10 runs around 6 of them would give the runtimes mentioned in the table for the Multi-GPU cases. The number of blocks was always chosen to be equal to the number of streaming processors the GPU has. For example the V100 has 80 SMs, RTX6000 - 72 and A100 - 108. Threads per block were kind of a hit-and-trial method, I would check the average of say 5 runs and take whichever gave the fastest times.
Multi-GPU despite being considerably better than a single GPU, the total runtime was slower because I was using MPI_Reduce() to get the results from the GPUs which I think is the reason for the high total runtime. I am not sure whether just letting each of the GPUs write to a different file and then reducing the results would be faster.

Proc	Grid	Time (SP)	K Time (SP)	Time (DP)	K Time (DP)	Blk/TPB	Cores	Samples
A100	1000^2	503 ms	132.75 ms	613 ms	228.88 ms	108/512	-	14.83 Billion
A100	100^2	277 ms	132.75 ms	391 ms	228.84 ms	108/512	-	14.83 Billion
V100	1000^2	574 ms	178.01 ms	708 ms	312.04 ms	80/512	-	14.83 Billion
V100	100^2	341 ms	177.71 ms	475 ms	311.61 ms	80/512	-	14.83 Billion
RTX 6000	1000^2	851 ms	488.88 ms	5.05 s	4.68 s	72/256	-	14.91 Billion
RTX 6000	100^2	615 ms	487.66 ms	4.78 s	4.64 s	72/256	-	14.91 Billion
CPU Serial	1000^2	333.78 s	-	324.61 s	-	-	1	14.92 Billion
CPU Serial	100^2	333.55 s	-	317.52 s	-	-	1	14.92 Billion
CPU OMP	1000^2	26.54 s	-	25.37 s	-	-	16	14.92 Billion
CPU OMP	100^2	26.81 s	-	25.63 s	-	-	16	14.92 Billion
2 GPUs (V100)	1000^2	1.08 s	90.33 ms	1.20 s	157.43 ms	80/512	2	14.78 Billion
2 GPUs (V100)	100^2	0.81 s	90.31 ms	0.95 s	157.50 ms	80/512	2	14.78 Billion
4 GPUs (V100)	1000^2	1.75 s	47.23 ms	1.85 s	81.38 ms	80/512	4	14.87 Billion
4 GPUs (V100)	100^2	1.42 s	46.90 ms	1.50 s	83.10 ms	80/512	4	14.87 Billion

The performance tests were assumed with at least one billion rays (I mention at least because the way I divide work needed the number of rays to be divisible by the product of thread per block andthe number of blocks, so I am calculating slightly more than a billion rays (a few thousand) for GPU problems), xorwow RNG in curand, with problem parameters set in Milestone 1 and 2. For both V100 and RTX 6000, I used -arch=sm_70. Anything other than this and the RTX 6000 would not launch the kernel. For A100 -arch=sm_80 was used.

Optimizations

The majority of my speedup came by simply changing the way I wrote my loops.

I went from this -

```
for(){
        while(1){
        // Get a ray that meets our conditions then break
        }
        // do other processing
}
```

To-

```
while(count<number of rays){
        if(a ray meets our conditions){
        count++;
        // do other processing
        }
}
```

This did not affect my serial runtimes however. But gives considerable speedup on a GPU. The next optimization I did was to reduce the number of times I call trigonometric functions. Using perf on my serial code I got the following output initially -

```
Samples: 1M of event 'cycles:u', Event count (approx.): 1307437720624
Overhead  Command  Shared Object        Symbol
  39.03%  code     code                 [.] main
  34.47%  code     libm-2.28.so         [.] sincosf32x
  11.87%  code     libc-2.28.so         [.] __random
   8.85%  code     libc-2.28.so         [.] __random_r
   3.51%  code     libc-2.28.so         [.] rand
   1.18%  code     code                 [.] rand@plt
   0.57%  code     code                 [.] sincos@plt
   .
   .
```

As you can see, the trigonometric functions were nearly 35 percent of the overhead. I was initially calling a sin function and a cos function for each sample. I reduced this to just one cos function and calculated the sin function value from the formula. This lead to the following perf report -

```
Samples: 1M of event 'cycles:u', Event count (approx.): 1152927769019
Overhead  Command  Shared Object        Symbol
  48.36%  code     code                 [.] main
  13.55%  code     libc-2.28.so         [.] __random
```

```
10.18%  code          libc-2.28.so    [.] __random_r
 4.03%  code          libc-2.28.so    [.] rand
 2.52%  code          libm-2.28.so    [.] __sqrt_finite
 1.32%  code          code            [.] rand@plt
 0.71%  code          libm-2.28.so    [.] sqrtf32x
 0.70%  code          libm-2.28.so    [.] 0x000000000007ef2e
 0.67%  code          libm-2.28.so    [.] 0x000000000007ee2f
 0.66%  code          libm-2.28.so    [.] 0x000000000007ee21
 0.66%  code          libm-2.28.so    [.] 0x000000000007ee13
 0.65%  code          code            [.] cos@plt
```

Reduced the overhead of trigonometric function calls to just 0.65 percent. This did lead to an increase in the square root function call, however the net overhead was reduced considerably.

I also changed from an array of pointers to a single contiguous array but I did not see any speedup with this. Maybe because we had added one more calculation to get the index of the array.