Implementation

1. Structure

The program implement rendering in cuda by converting `render` to a kernel, which renders one pixel per threads. In addition, the `3x3` anti-aliasing loops' first layer is unrolled by specifying `m` to the kernel and launching it 3 times. Then, another kernel `_convert_pixel` is used to convert floating point colors to uint8 pixel values.

2. Parrtition

Each block has 2D `32x8=256` threads, corresponding to a `32x8` pixels region. And the whole image is splitted to `ceil(w/32) x ceil(h/8)` blocks.

3. Optimization

The main optimization is to turn on `-use_fast_math` flag in the compiler and use float instead of double. Then, I also converted many operations to their fused version. For example:

- `A * B + C` -> `__fmaf(A, B, C)`
- `s = sin(T), c = cos(T)` -> `__sincosf(T, &s, &c)`

Another small trick is to turn off PNG compression for reduced sequential runtime.

Analysis

1. GPU kernel execution

```
Kernel                                       Time(%)      Time     Calls
Avg        Min        Max
render_pixel                                  98.98%  2.33286s         3
777.62ms   763.78ms   791.09ms
Copy device buffer to host                     0.98%  23.175ms         1
23.175ms   23.175ms   23.175ms
convert_pixel                                  0.02%  456.09us         1
456.09us   456.09us   456.09us
Initialize float device buffer                 0.01%  310.24us         1
310.24us   310.24us   310.24us
Copy global varaibles to device                0.00%  8.3520us         9
928ns      896ns      960ns
```

2. Nsight Compute

Under different settings of threads (`32x4`, `32x8`, `32x16`), the throughput of SM all reach ~80%, so the computation is bounded by ALU and adjusting the partition won't help.

3. Additional analysis

IO can help a lot.

Conclusion

1. It's hard to optimize the arithmetic operations.

2. You are amazing.