1 Implementation

1.1 How CUDA is used?

Ray-marching parallelization: Each GPU thread renders one complete pixel independently. The kernel render_kernel maps one thread to one pixel using a 1D grid layout:

```
int gid = blockDim.x * blockIdx.x + threadIdx.x;
int i = gid / width;  // row
int j = gid % width;  // column
```

Memory hierarchy
Constant memory (`__constant__`):
All scene parameters (camera basis, render settings, light direction) are copied once to constant cache, which is read-only and broadcast efficiently to all threads.

Registers:
Per-thread state (ray direction rd, accumulated color fcol, distance estimates) lives entirely in registers—no shared memory or global reads during computation.

Coalesced global writes:
Each thread writes a single uchar4 pixel (RGBA) to global memory. Because threads in a warp process consecutive pixels (due to 1D layout), these 4-byte writes are coalesced into 128-byte transactions. No data sharing between threads: Ray marching is embarrassingly parallel—no inter-thread communication, no shared memory, no synchronization barriers within the kernel.

1.2 Task Partitioning (Threads and Blocks)

1D grid of 1D blocks

```
blockDim.x = BLOCK_SIZE = 512 threads per block.
gridDim.x = ceil(width * height / 512) blocks.
Total threads >= total pixels; extra threads exit early.
```

1.3 Optimization Techniques

1. Float precision:
Device math uses `float` (not `double`) for 2–3× throughput on normal GPUs. Host camera setup uses `double` for accuracy, then converts to `float` before copying to the device. This provides the most meaningful speedup.

2. Power-8 specialization:
Since the Mandelbulb power is almost always 8, we precompute and use `use_pow8_d` on the host. In the hot loop (`md`), instead of calling `powf(r, 8.0)` twice per iteration, we compute r^8 via three multiplies.

3. Fast math intrinsics:

- `rsqrtf` (reciprocal square root) instead of `1.0f / sqrtf`.
- `sincosf` computes `sin` and `cos` simultaneously ($\sim 2\times$ faster than separate calls).
- `__fmaf_rn` (fused multiply–add) for `dr = dr_mul * dr + 1.f` (1 op instead of 2).
- `fmaxf`, `fminf` instead of `max`, `min`.

4. Loop unrolling:

- `#pragma unroll 4` in `md`: unrolls four iterations of the Mandelbulb loop to reduce branch overhead and expose instruction-level parallelism (ILP). Unrolling more would increase register pressure and hurt occupancy.
- `#pragma unroll 4` in `softshadow`: balances loop overhead vs. register usage (the shadow kernel is called 9× per pixel).

5. Launch configuration:
`__launch_bounds__(256, 8)` hints the compiler:

- Target 256 threads per block.
- Aim for 8 blocks per SM (2048 threads/SM).
- Compiler limits register usage to about 48 registers per thread to fit 4 blocks, improving occupancy and latency hiding.

6. Fast PNG encoding:
Near the end of this homework, I found out PNG encoding consumes considerablr portions of total mesured runtime, especially for larger file. Replacing the original function from loadpng with new writepng function has provided around 50% speedup for the larger testcase.

2 Analysis

I measure the GPU kernel execution time using **nvprof** and profile the program with **Nsight Compute (ncu)** under different setting. Specifically, I adjust the **block size** among **64, 128, 256, 512, 1024**. I run these profiling on **testcase 0**, which is not the most time-consuming public testcase, but its ncu run much more faster. Part of the profiling results are highlighted beneath:

Table 1: CUDA Kernel Performance Analysis: Block Size Comparison

Block Size	Time (ms)	Occupancy (%)	Reg/Thrd	Spill	SM Busy (%)
64	15.91	2.82	48	No	10.61
128	15.94	5.17	48	No	19.36
256	17.20	9.27	32	Yes	31.47
512	18.68	16.83	49	No	50.92
1024	25.55	31.22	49	No	65.34

Table 2: Metrics Definition

Metric	**Definition**
Time (ms)	Wall-clock kernel execution time
Occupancy (%)	Ratio of active warps to maximum warps per SM
Reg/Thrd	Registers allocated per thread
Spill	Register spills to local memory
SM Busy (%)	Percentage of cycles where SM has active work

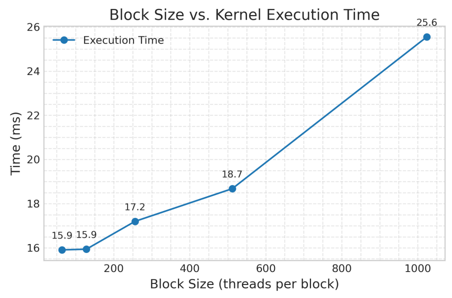

Figure 1: Block size vs. Time for testcase 0

Testing with testcase 0 is not very representative, since its small size severely limits the potential of GPU computing. For larger size testcase, for example, the most time-consuming testcase 8, I found that block size 256 have the best perofrmance. In some further testings, I have noticed that even configurations with register spills can be faster, like when (block size, grid size) = (256, 8). This may be because the kernel is compute-bound, spills enable higher occupancy, and spilled data mostly hits the L1 cache (so the L1 spill cost is much lower than the stalled warp latency).

3 Conclusion

3.1 Difficulties Encountered

One major difficulty in implementation was balancing performance and accuracy. During development, some code modifications led to performance gains, but the accuracy for certain test cases dropped far below acceptable thresholds. When testing different configurations, the actual impact of memory coalescing was not always clear. Additionally, running ncu took considerable time, and interpreting meaningful metrics for analysis and comparison is somehow challenging.

3.2 Feedback and Suggestions

I think the assignment become more interesting when we start working with CUDA programming. I am still learning how to extract meaningful metrics and interpret the profiling logs and utilize them efficiently. Tuning block and grid sizes demonstrated some performance differences, but there is still much to learn, particularly how these metrics interact with different hardware and settings and how they collectively affect overall runtime.