Parallel Programming HW3 Report

B10705009 資工四 邱一新

My Implementation

1. How did I implement my program using cuda?

My CUDA implementation follows the classic Host/Device model, transforming the original sequential CPU loop that processed each pixel individually into a parallel GPU computation that handles all pixels simultaneously.

On the Host side (the `main` function), the CPU is responsible for parsing command-line arguments, allocating memory on both the Host (`raw_image`) and Device (`d_image`), and packaging all rendering parameters such as camera position, field of view, and iteration count into a `RenderParams` structure. This structure is then copied to the GPU's `__constant__` memory for fast access. The Host also configures the grid and block dimensions for the kernel launch, waits for execution to finish, and finally copies the rendered result from the Device back to the Host.

On the Device side, the main kernel `render_kernel` (marked `__global__`) performs the per-pixel rendering tasks in parallel—essentially replacing the original CPU `for` loop. Supporting functions such as `md`, `map`, `trace`, `calcNor`, and `softshadow` are all marked with `__device__`, meaning they are executed directly on the GPU and called from within the kernel.

In place of the GLM library used in the CPU version, the CUDA implementation relies on built-in vector types like `float3` and `float2`. I reimplemented all necessary vector operations and mathematical functions such as addition, subtraction, dot product, normalization, and cross product, each annotated with `__device__ __forceinline__` to ensure efficient inlining and execution performance.

2. How did I partition the tasks among GPU threads and blocks?

I adopted a 2D data-parallel strategy, mapping each pixel of the image directly to a single CUDA thread.

The task of each thread is to compute the final color of one pixel. Inside the kernel, I use `blockIdx`, `blockDim`, and `threadIdx` to determine the global pixel coordinates `(j, i)` corresponding to each thread.

Threads are organized into 2D blocks. In the `main` function, I set `dim3 blockDim(16, 16);`, meaning each block contains $(16 \times 16 = 256)$ threads. This configuration helps

effectively hide memory latency and achieve good occupancy.

For the grid configuration, I use the "ceiling division" technique `(width + blockDim.x - 1) / blockDim.x` (and similarly for height) to ensure that enough blocks are launched even when the image dimensions are not multiples of 16.

Because this method may launch extra threads at the image boundaries, I include a boundary check `if (j >= width || i >= height) { return; }` inside the kernel. This ensures that out-of-range threads terminate early, avoiding unnecessary computation and memory access.

3. Other optimization skills in my program.

My CUDA implementation incorporates several key optimization techniques that significantly improve performance. First, I use `__constant__` memory to store all read-only parameters shared across threads (such as camera position, FOV, and iteration count) in a `RenderParams` structure. This leverages the GPU's constant cache, allowing warp-wide broadcasts when threads access the same address, resulting in much faster reads than from global memory. Second, I replaced all double-precision (`double`, `glm::dvec3`) computations from the CPU version with single-precision (`float`, `float3`) in CUDA. This reduces memory usage per value from 64 to 32 bits, improves register efficiency, and provides a practical 2–3× speedup without noticeable loss of accuracy. Third, I optimized mathematical computations by eliminating expensive `pow()` calls in the `md` function. Since the exponent was fixed (`power = 8.0`), I expanded it using simple multiplications (`r2 = r*r`, `r4 = r2*r2`, `r8 = r4*r4`, `r7 = r8 / r`), which is far more efficient for integer powers. In addition, I compiled the program with the `-use_fast_math` flag, enabling the use of faster, hardware-optimized approximations for common mathematical functions, which further accelerates floating-point operations. Finally, I used the CUDA intrinsic `sincos()` function to compute both sine and cosine values simultaneously, reducing redundant trigonometric calls and improving performance in angle-related calculations.

Analysis

1. Measure the GPU kernel execution time using nvprof

Using `nvprof`, I measured the GPU kernel execution time as the duration reported for my `render_kernel`. The results show that under heavy workloads, the total execution time is almost entirely dominated by the render kernel, indicating a compute-bound behavior where the CPU mainly waits for GPU completion. Under lighter workloads, the kernel time represents only a portion of the total runtime, as fixed overheads such as memory allocation and launch costs become relatively significant. In other words, the `nvprof` measurements reveal that the kernel execution time dominates performance at high workloads but is diluted by fixed costs when the workload is small.

- The result of testcase01 (light workload)

```
[b10705009@un-ln01 hw3]$ srun -N 1 -n 1 --gpus-per-node 1 -A ACD114118 -t 5 nvprof ./hw3 4.152 2.398 -2.601 0 0 0 512 512 output_gpu/01.
png
WARNING: When login node is DOWN, Your Job will FAIL!
WARNING: Please use `sbatch` instead :)
srun: INFO: Try to avoid using --ntasks
srun: INFO: It is recommended to specify `--nodes` and `--ntasks-per-node` together
==3379533== NVPROF is profiling process 3379533, command: ./hw3 4.152 2.398 -2.601 0 0 0 512 512 output_gpu/01.png
==3379533== Profiling application: ./hw3 4.152 2.398 -2.601 0 0 0 512 512 output_gpu/01.png
==3379533== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   99.95%  160.09ms         1  160.09ms  160.09ms  160.09ms  render_kernel(unsigned char*, int, int)
                    0.05%  82.367us         1  82.367us  82.367us  82.367us  [CUDA memcpy DtoH]
                    0.00%  1.3760us         1  1.3760us  1.3760us  1.3760us  [CUDA memcpy HtoD]
      API calls:   60.69%  160.08ms         1  160.08ms  160.08ms  160.08ms  cudaEventSynchronize
                   38.52%  101.60ms         1  101.60ms  101.60ms  101.60ms  cudaMalloc
                    0.32%  837.37us         1  837.37us  837.37us  837.37us  cudaMemcpy
                    0.18%  463.59us       114  4.0660us      71ns  255.25us  cuDeviceGetAttribute
                    0.14%  363.80us         1  363.80us  363.80us  363.80us  cudaMemcpyToSymbol
                    0.10%  262.89us         1  262.89us  262.89us  262.89us  cudaFree
                    0.03%  67.607us         1  67.607us  67.607us  67.607us  cudaLaunchKernel
                    0.01%  35.109us         2  17.554us  7.2350us  27.874us  cudaEventRecord
                    0.01%  13.317us         2  6.6580us     531ns  12.786us  cudaEventCreate
                    0.01%  13.196us         1  13.196us  13.196us  13.196us  cuDeviceGetName
                    0.00%  7.3880us         2  3.6940us     442ns  6.9460us  cudaEventDestroy
                    0.00%  6.7850us         1  6.7850us  6.7850us  6.7850us  cuDeviceGetPCIBusId
                    0.00%  4.3150us         1  4.3150us  4.3150us  4.3150us  cudaEventElapsedTime
                    0.00%  3.1200us         1  3.1200us  3.1200us  3.1200us  cudaGetLastError
                    0.00%  1.5200us         3     506ns      78ns  1.3200us  cuDeviceGetCount
                    0.00%     447ns         2     223ns      86ns     361ns  cuDeviceGet
                    0.00%     214ns         1     214ns     214ns     214ns  cuModuleGetLoadingMode
                    0.00%     212ns         1     212ns     212ns     212ns  cuDeviceTotalMem
                    0.00%     172ns         1     172ns     172ns     172ns  cuDeviceGetUuid
[b10705009@un-ln01 hw3]$
```

截圖 2025-10-30 晚上10.30.32.png

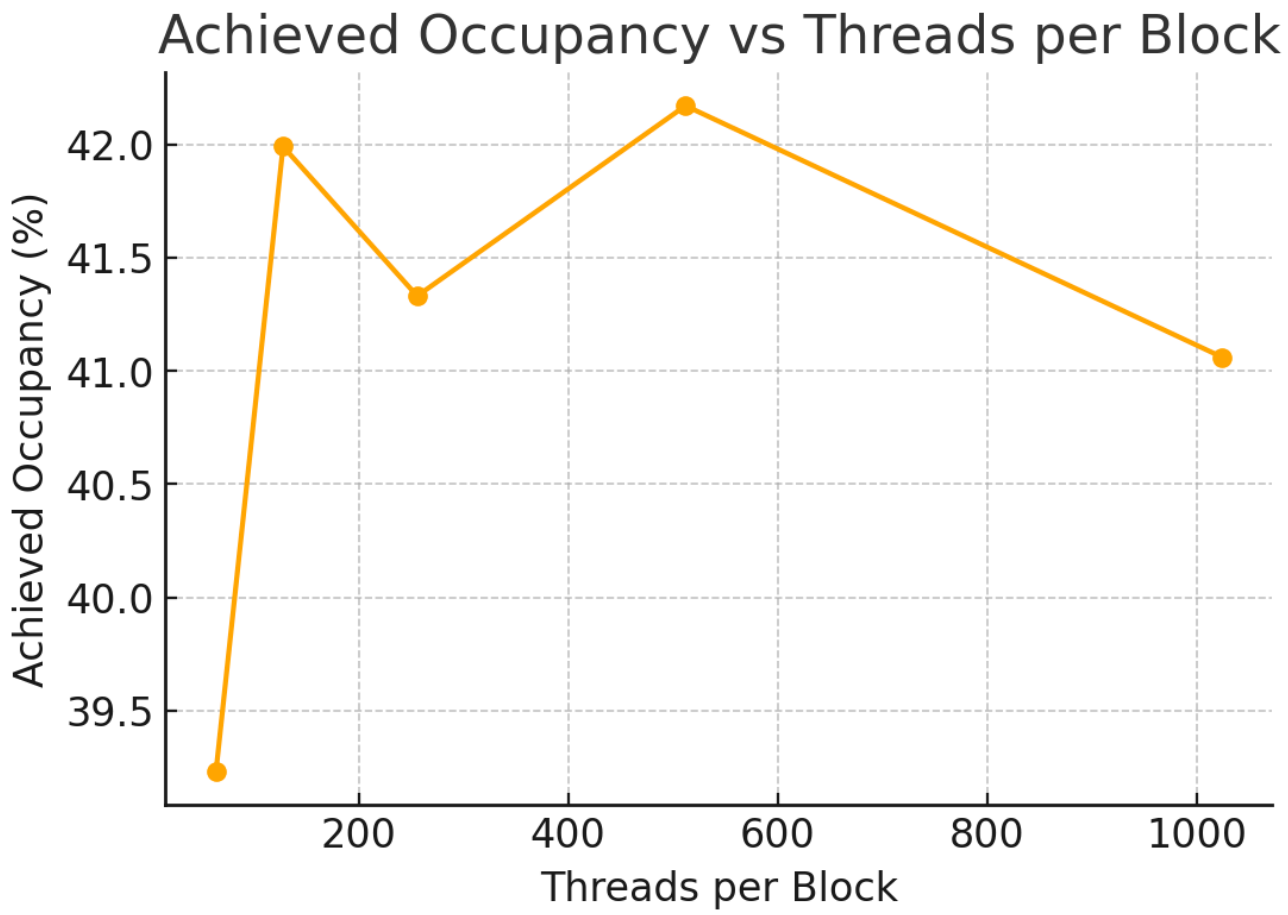
- The result of testcase08 (heavy workload)

```
[b10705009@un-ln01 hw3]$ srun -N 1 -n 1 --gpus-per-node 1 -A ACD114118 -t 60 nvprof ./hw3 -1.2 -0.51 -0.8 -0.271 -0.299 -0.379 4096 4096
 output_gpu/08.png
WARNING: When login node is DOWN, Your Job will FAIL!
WARNING: Please use `sbatch` instead :)
srun: INFO: Try to avoid using --ntasks
srun: INFO: It is recommended to specify `--nodes` and `--ntasks-per-node` together
==3344468== NVPROF is profiling process 3344468, command: ./hw3 -1.2 -0.51 -0.8 -0.271 -0.299 -0.379 4096 4096 output_gpu/08.png
==3344468== Profiling application: ./hw3 -1.2 -0.51 -0.8 -0.271 -0.299 -0.379 4096 4096 output_gpu/08.png
==3344468== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   99.22%  2.87795s         1  2.87795s  2.87795s  2.87795s  render_kernel(unsigned char*, int, int)
                    0.78%  22.629ms         1  22.629ms  22.629ms  22.629ms  [CUDA memcpy DtoH]
                    0.00%  1.3760us         1  1.3760us  1.3760us  1.3760us  [CUDA memcpy HtoD]
      API calls:   95.78%  2.87794s         1  2.87794s  2.87794s  2.87794s  cudaEventSynchronize
                    3.39%  101.74ms         1  101.74ms  101.74ms  101.74ms  cudaMalloc
                    0.79%  23.619ms         1  23.619ms  23.619ms  23.619ms  cudaMemcpy
                    0.01%  439.82us         1  439.82us  439.82us  439.82us  cudaMemcpyToSymbol
                    0.01%  390.84us       114  3.4280us      68ns  184.13us  cuDeviceGetAttribute
                    0.01%  364.17us         1  364.17us  364.17us  364.17us  cudaFree
                    0.00%  78.025us         1  78.025us  78.025us  78.025us  cudaLaunchKernel
                    0.00%  49.690us         2  24.845us  7.4370us  42.253us  cudaEventRecord
                    0.00%  31.901us         2  15.950us  2.6270us  29.274us  cudaEventCreate
                    0.00%  14.756us         1  14.756us  14.756us  14.756us  cuDeviceGetName
                    0.00%  10.144us         1  10.144us  10.144us  10.144us  cuDeviceGetPCIBusId
                    0.00%  10.053us         2  5.0260us     539ns  9.5140us  cudaEventDestroy
                    0.00%  9.8800us         1  9.8800us  9.8800us  9.8800us  cudaGetLastError
                    0.00%  7.3530us         1  7.3530us  7.3530us  7.3530us  cudaEventElapsedTime
                    0.00%  1.0640us         3     354ns      69ns     795ns  cuDeviceGetCount
                    0.00%     449ns         2     224ns      76ns     373ns  cuDeviceGet
                    0.00%     323ns         1     323ns     323ns     323ns  cuDeviceTotalMem
                    0.00%     293ns         1     293ns     293ns     293ns  cuModuleGetLoadingMode
                    0.00%     285ns         1     285ns     285ns     285ns  cuDeviceGetUuid
[b10705009@un-ln01 hw3]$
```

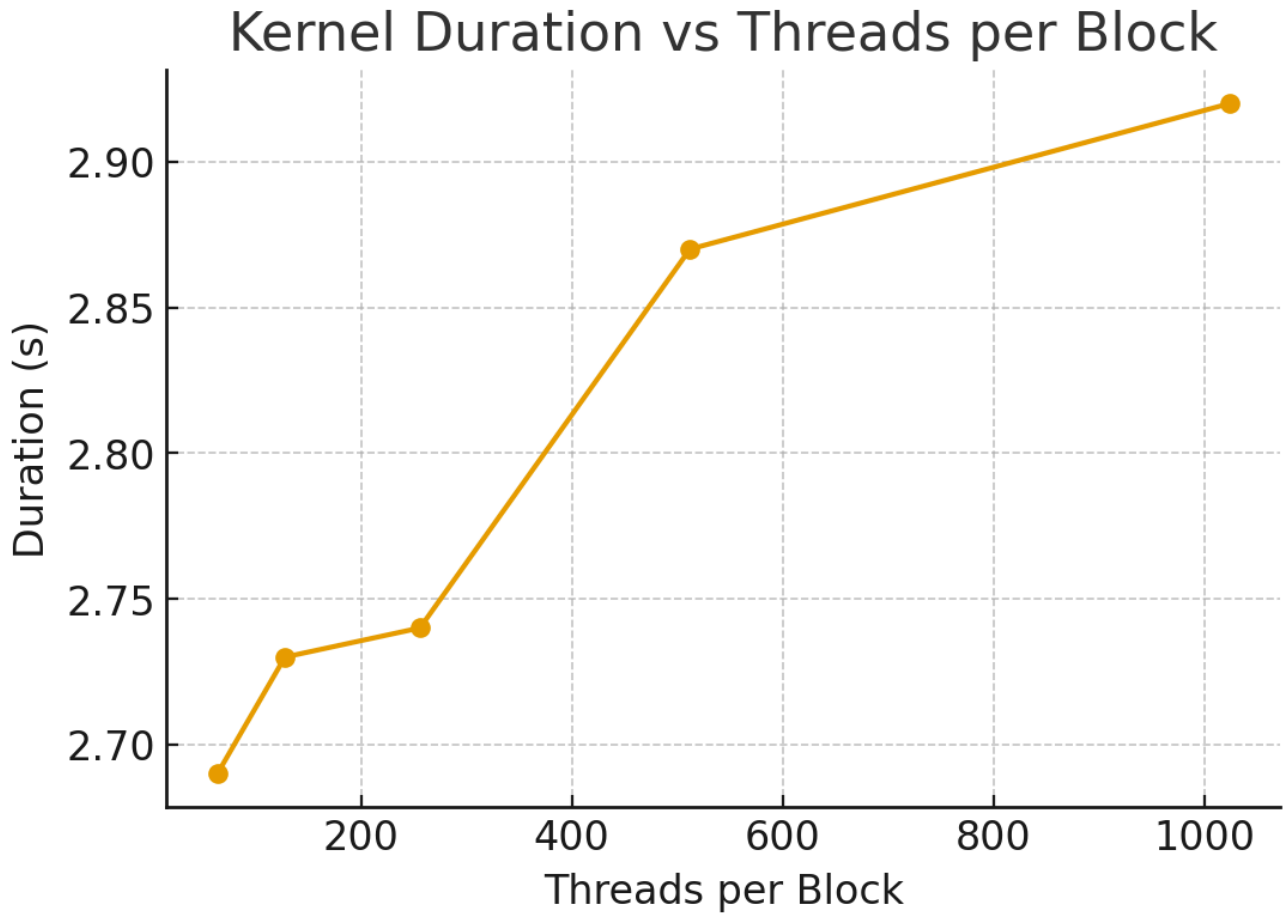
截圖 2025-10-30 晚上10.21.43.png

2. Performance difference under different GPU kernel configurations

I experimented with several block dimension configurations, and the results are shown in the figures below. Each configuration corresponds to a different number of threads per block ((8, 8), (16, 8), (16, 16), (32, 16), (32, 32)).



output (1) 1.png



output 1.png

This experiment shows that performance differences across various block configurations are minimal, with the primary bottleneck coming from the uniform 50% theoretical occupancy limit. Each thread uses 64 registers, and this register pressure fixes the GPU's maximum parallelism at 50%, causing all configurations to achieve only around 39–43% occupancy. Although total runtime differs by only about 8.5%, some trends are observable: smaller blocks (e.g., 8×8, 16×8) perform slightly better due to finer scheduling granularity and better load balancing; medium-sized blocks (16×16) strike a good balance between cache locality and scheduling flexibility; and larger blocks (32×16, 32×32) suffer from intra-block workload imbalance, where threads become idle waiting for the slowest ones to finish, leading to slightly worse performance. An attempt was made to increase occupancy by switching to 32-bit registers and reducing register usage, but this caused register spilling into local memory, introducing additional memory access latency and degrading overall performance. As a result, the configuration using 64 registers per thread —yielding a 50% theoretical occupancy—remained the most efficient overall.

Conclusion

The main difficulty I encountered was that performance optimization involves not only simplifying computations but also understanding how many resources the GPU actually has, which greatly affects the results. This way of thinking about performance tuning was quite new to me. Other than that, the assignment itself wasn't particularly tricky—I think it was very well designed. Completing it made me much more familiar with CUDA programming and gave me a concrete sense of how parallelizing computations can significantly improve performance. At the same time, it didn't include too many unnecessary or overly difficult parts, allowing me to focus on the core objectives of the assignment. I think that was great.