平行程式設計 hw3 report

資工四 B10502136 邱垣盛

1 Implementation

1.1 CUDA Implementation

The program's implementation revolves around migrating the per-pixel rendering task, originally handled by nested `for` loops on the CPU, to a massively parallel CUDA kernel named `renderKernel_tiled`. All core computational logic, including the Mandelbulb distance function (`md`), scene mapping (`map`), ray tracing (`trace`), and shading functions (`calcNor`, `softshadow`), were converted into `__device__` functions. The host-side `main` function is responsible for setup and teardown: it allocates device memory for the output image (`d_image`), copies the final result back to the host (`h_image`), and manages the kernel launch.

A key part of the implementation is the efficient handling of global scene data. Instead of passing many parameters to the kernel, scene-wide, read-only values like the camera position (`d_camera_pos`), target (`d_target_pos`), resolution (`d_iResolution`), and fractal parameters (`d_power`, `d_md_iter`) are copied from the host to `__constant__` device memory before the kernel launch. This data is then broadcast to all threads with low latency via a dedicated cache, which is much faster than repeated global memory reads. The kernel itself is launched with a grid of blocks, and its primary arguments are the pointer to the device image buffer (`d_image`) and the image dimensions.

1.2 Task Partitioning

Task partitioning is achieved using a dynamic load-balancing model built on spatial tiles. The image is divided into `16x16` pixel tiles. The kernel is launched with a 1D grid of thread blocks, where each block contains `16x16` (256) threads. A global `unsigned int` in device memory, `g_tileCounter`, acts as a work queue. Inside a `while(true)` loop, the first thread in each block (`threadIdx.x == 0 && threadIdx.y == 0`) atomically increments this counter to claim a unique `tileId`. All threads in that block then use this `tileId` to calculate the corresponding pixel coordinates they are responsible for rendering.

This dynamic task stealing model is highly effective for this workload. When a block finishes rendering an easy tile (like empty sky), it simply loops and claims the next available `tileId` from the counter. This prevents blocks from becoming idle, maximizing GPU utilization. The loop terminates when a block claims a `tileId` that is greater than or equal to the `totalTiles`.

This exit condition is checked by the first thread and broadcast to the rest of the block using `__shared__` memory (`shouldQuit`) and synchronization barriers (`__syncthreads()`). This approach is superior to a static 2D grid,one block per tile, where workload imbalance would cause the entire kernel to wait for the single block rendering the most complex part of the fractal.

1.3 Optimization Skills

Occupancy tuning launches much more blocks than the number of streaming multiprocessors available. This ensures the GPU always has work available when blocks stall on memory operations, hiding latency and maintaining high throughput.

Shared memory reduces atomic contention by having only one thread per block perform atomic operations and broadcast results to all threads. This reduces atomic memory operations by a factor of 256 compared to having every thread access the global counter independently.

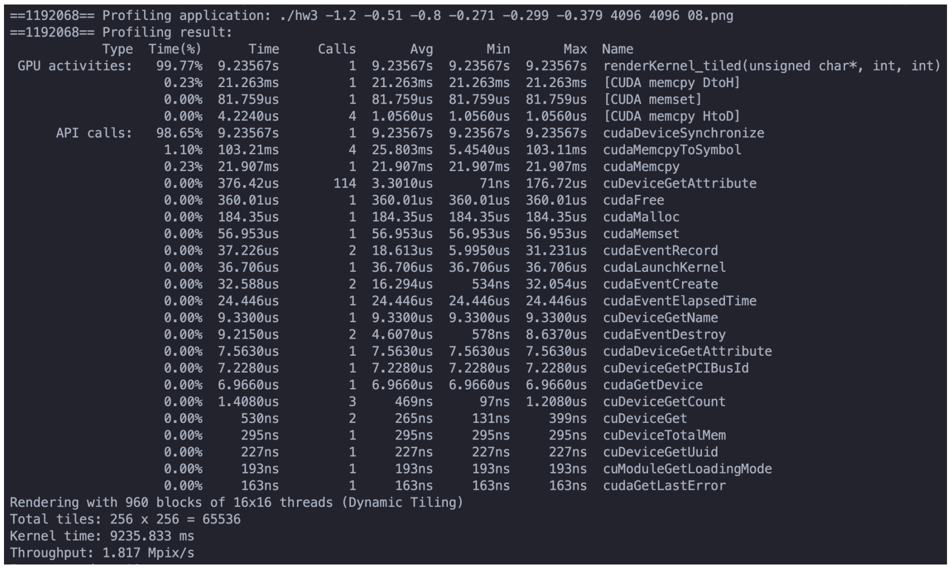
The implementation reduces precision from double (64-bit) to float (32-bit) throughout, as Mandelbulb rendering doesn't require double precision. Float operations execute approximately twice as fast on GPU architectures and consume half the memory bandwidth. Memory is preinitialized with `cudaMemset()` to ensure clean output without requiring conditional writes in the kernel.

The packed RGBA write pattern ensures threads in a warp access consecutive memory addresses, maximizing memory bandwidth utilization to up to 128 bytes per transaction. This coalesced access pattern is critical for achieving high memory throughput on GPU architectures.

2 Analysis

2.1 nvprof

Based on the profiling results, the optimal GPU kernel execution time was 10650.00 ms, achieved with the 8x8 block size configuration in test case 8.

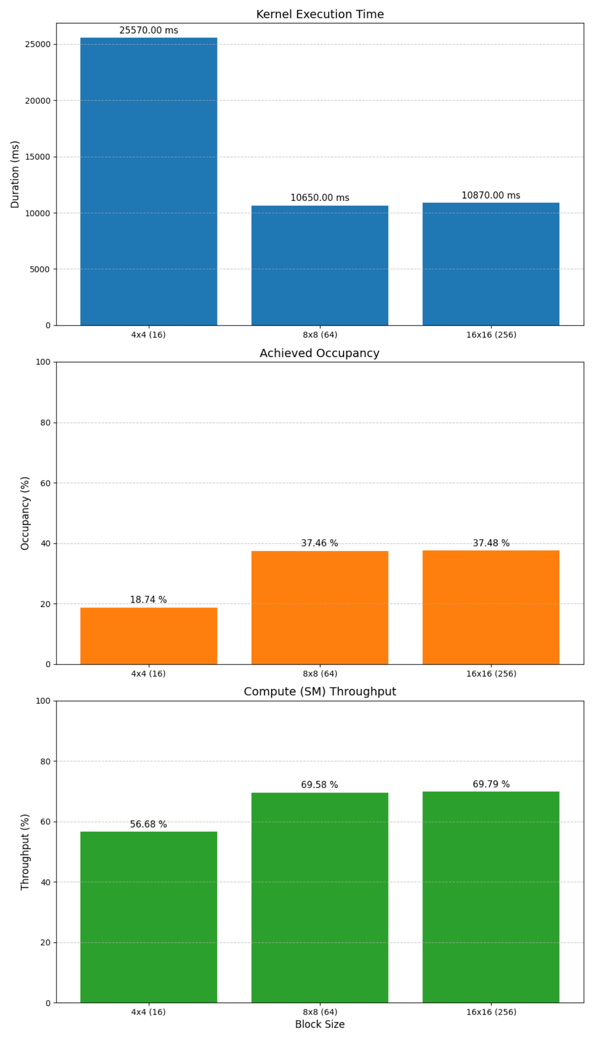


```
==1192068== Profiling application: ./hw3 -1.2 -0.51 -0.8 -0.271 -0.299 -0.379 4096 4096 08.png
==1192068== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   99.77%  9.23567s         1  9.23567s  9.23567s  9.23567s  renderKernel_tiled(unsigned char*, int, int)
                    0.23%  21.263ms         1  21.263ms  21.263ms  21.263ms  [CUDA memcpy DtoH]
                    0.00%  81.759us         1  81.759us  81.759us  81.759us  [CUDA memset]
                    0.00%  4.2240us         4  1.0560us  1.0560us  1.0560us  [CUDA memcpy HtoD]
      API calls:   98.65%  9.23567s         1  9.23567s  9.23567s  9.23567s  cudaDeviceSynchronize
                    1.10%  103.21ms         4  25.803ms  5.4540us  103.11ms  cudaMemcpyToSymbol
                    0.23%  21.907ms         1  21.907ms  21.907ms  21.907ms  cudaMemcpy
                    0.00%  376.42us       114  3.3010us      71ns  176.72us  cuDeviceGetAttribute
                    0.00%  360.01us         1  360.01us  360.01us  360.01us  cudaFree
                    0.00%  184.35us         1  184.35us  184.35us  184.35us  cudaMalloc
                    0.00%  56.953us         1  56.953us  56.953us  56.953us  cudaMemset
                    0.00%  37.226us         2  18.613us  5.9950us  31.231us  cudaEventRecord
                    0.00%  36.706us         1  36.706us  36.706us  36.706us  cudaLaunchKernel
                    0.00%  32.588us         2  16.294us     534ns  32.054us  cudaEventCreate
                    0.00%  24.446us         1  24.446us  24.446us  24.446us  cudaEventElapsedTime
                    0.00%  9.3300us         1  9.3300us  9.3300us  9.3300us  cuDeviceGetName
                    0.00%  9.2150us         2  4.6070us     578ns  8.6370us  cudaEventDestroy
                    0.00%  7.5630us         1  7.5630us  7.5630us  7.5630us  cudaDeviceGetAttribute
                    0.00%  7.2280us         1  7.2280us  7.2280us  7.2280us  cuDeviceGetPCIBusId
                    0.00%  6.9660us         1  6.9660us  6.9660us  6.9660us  cudaGetDevice
                    0.00%  1.4080us         3     469ns      97ns  1.2080us  cuDeviceGetCount
                    0.00%     530ns         2     265ns     131ns     399ns  cuDeviceGet
                    0.00%     295ns         1     295ns     295ns     295ns  cuDeviceTotalMem
                    0.00%     227ns         1     227ns     227ns     227ns  cuDeviceGetUuid
                    0.00%     193ns         1     193ns     193ns     193ns  cuModuleGetLoadingMode
                    0.00%     163ns         1     163ns     163ns     163ns  cudaGetLastError
Rendering with 960 blocks of 16x16 threads (Dynamic Tiling)
Total tiles: 256 x 256 = 65536
Kernel time: 9235.833 ms
Throughput: 1.817 Mpix/s
```

2.2 ncu

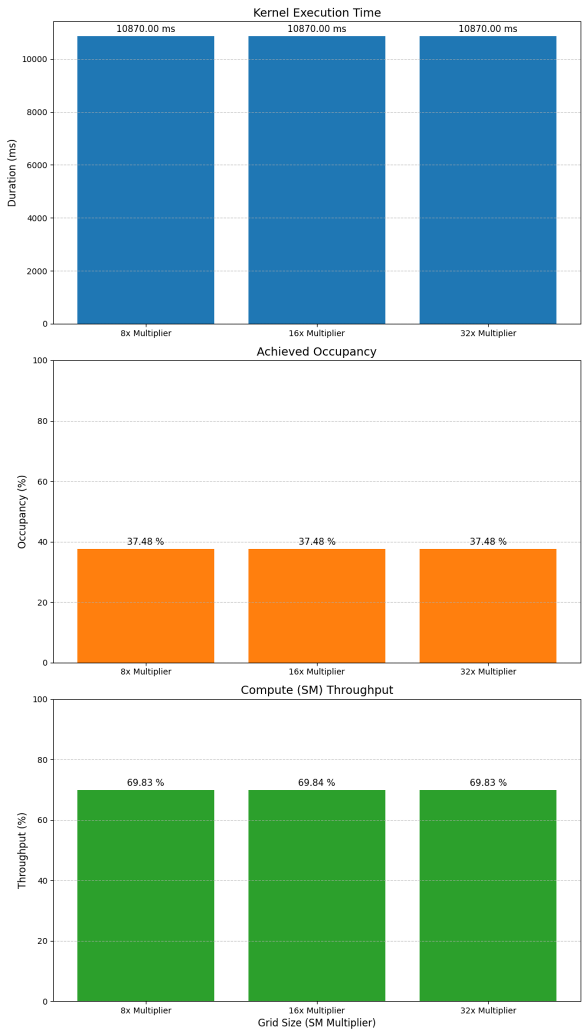
Different block size Based on the profiling results, the optimal kernel execution time was 10650.00 ms, achieved with an 8x8 block size.

The Nsight Compute plots show this performance difference clearly. The `4x4` block was inefficient, with low Achieved Occupancy (18.74%). Switching to `8x8` doubled occupancy to 37.5% and boosted SM Throughput from 56.68% to 69.58%, resulting in the fastest execution time. The `16x16` block offered no significant further improvement.



Different grid size Based on profiling, the kernel execution time was 10870.00 ms.

The Nsight Compute plots show that performance saturated with the 8x multiplier. Increasing the grid size to 16x or 32x provided no change in execution time, occupancy (37.48%), or SM throughput (69.8%), as the GPU was already fully utilized.



2.3 Additions

The dynamic load-balancing strategy is highly effective. The grid size analysis proves that performance saturates at the 8x multiplier; increasing the grid size further has no impact. This design successfully manages the unbalanced workload and removes the need for precise grid size tuning.

This workload is clearly compute-bound, not memory-bound, as SM Throughput reached 70% while DRAM Throughput was near 0%. The `8x8` block size was optimal, providing the fastest execution time (10650.00 ms) by doubling the achieved occupancy (37.5%) compared to the inefficient `4x4` block.

3 Conclusion

3.1 Difficulty

The primary difficulty encountered was managing the highly unbalanced workload inherent in rendering a fractal. The computational cost varies dramatically per pixel; regions of empty space are computed quickly, while pixels near the complex fractal boundary require extensive iteration. A simple, static partitioning of the image among thread blocks would result in severe inefficiency, with most blocks finishing early and sitting idle while a few blocks handled the complex areas. This was overcome by implementing a dynamic tiling system. Using a global atomic counter (`g_tileCounter`), thread blocks could ”steal” the next available tile of work, ensuring that GPU resources remained highly utilized throughout the rendering process.

A secondary challenge involved the low-level performance optimization of the core distance-estimation function (`md`) on the GPU. Porting the mathematical operations from the CPU required careful optimization to maximize throughput. This involved replacing expensive `pow` operations with faster, explicit multiplications (e.g., `r2`, `r4`, `r8`) and using the `sincosf` intrinsic to calculate sine and cosine in a single instruction. Additionally, optimizing memory access patterns, such as using `__constant__` memory for global parameters and packing final pixel data into a `uint32_t` for coalesced writes, was essential to prevent memory bottlenecks.

3.2 Feedback

This assignment was my first experience with CUDA, and my key takeaway is that GPU acceleration is not automatic. I learned that effective parallel programming requires a deep understanding of the hardware’s constraints, such as occupancy, register limits, and shared memory, rather than just moving code to the GPU. This fractal’s unbalanced workload also taught me the importance of implementing a dynamic load-balancing strategy, like the tiling system used. Finally, I realized that even after a correct parallel architecture is designed, performance is not guaranteed. It requires significant fine-tuning of launch parameters, such as the block size and grid number, to truly maximize throughput.