

# HW3: Mandelbulb Ray Marching

## CUDA Optimization Report

Parallel Programming Assignment 3

**Date:** October 28, 2025

**Name:** 黃泓諺 R14922156

Course: Parallel Programming

# Contents

<b>1 Implementation</b>	<b>3</b>
1.1 How did you implement your program using CUDA? . . . . .	3
1.1.1 Core Architecture . . . . .	3
1.1.2 Memory Management Strategy . . . . .	3
1.1.3 Key Data Structures . . . . .	4
1.2 How did you partition the tasks among GPU threads and blocks? . . . . .	4
1.2.1 Thread and Block Configuration . . . . .	4
1.2.2 Task Allocation Strategy . . . . .	5
1.2.3 Per-Thread Workload Breakdown . . . . .	5
1.2.4 Workload Balance Considerations . . . . .	6
1.3 Other optimization skills in your program . . . . .	7
1.3.1 A. Kernel Configuration Optimization . . . . .	7
1.3.2 B. Algorithmic Optimization . . . . .	8
1.3.3 C. Compiler Optimization Directives . . . . .	8
1.3.4 D. Instruction-Level Optimization . . . . .	9
1.3.5 E. Memory Optimization . . . . .	10
1.3.6 F. Divergence Reduction . . . . .	11
1.3.7 Performance Results Summary . . . . .	12
<b>2 Analysis</b>	<b>13</b>
2.1 GPU Kernel Execution Time Measurement using nvprof . . . . .	13
2.1.1 nvprof Command . . . . .	13
2.1.2 nvprof Results (Case 06: $4096 \times 4096$ ) . . . . .	13
2.1.3 Key Findings from nvprof . . . . .	13
2.2 Nsight Compute (ncu) Detailed Profiling . . . . .	14
2.2.1 ncu Command . . . . .	14
2.2.2 GPU Speed Of Light Analysis . . . . .	14
2.2.3 Compute Workload Analysis . . . . .	15
2.2.4 Memory Workload Analysis . . . . .	15
2.2.5 Scheduler Statistics . . . . .	16
2.2.6 Warp State Statistics . . . . .	16
2.2.7 Instruction Statistics . . . . .	17
2.2.8 Launch Configuration . . . . .	17
2.2.9 Occupancy Analysis . . . . .	18
2.2.10 Performance Metrics Summary . . . . .	19
2.3 Block Size and Grid Size Configuration Analysis . . . . .	19
2.3.1 Experimental Setup . . . . .	19
2.3.2 Performance Results . . . . .	20
2.3.3 Why $8 \times 16$ Performs Best? . . . . .	21
2.3.4 Grid Size Impact . . . . .	23
2.3.5 Block Size Recommendation . . . . .	24
2.4 Additional Performance Observations . . . . .	24
2.4.1 1. Divergence vs Computation Trade-off . . . . .	24
2.4.2 2. L1 vs L2 Cache Behavior . . . . .	24
2.4.3 3. Register Usage Optimization Opportunities . . . . .	25
2.4.4 4. Instruction Mix Analysis . . . . .	25

2.4.5	5. Cache Hierarchy Performance . . . . .	26
2.4.6	6. Performance Bottleneck Summary . . . . .	26
2.4.7	Conclusions and Recommendations . . . . .	27
<b>3</b>	<b>Conclusion</b>	<b>30</b>
3.1	Difficulties Encountered . . . . .	30
3.1.1	1. Balancing Divergence vs Computation . . . . .	30
3.1.2	2. Register Pressure and Occupancy . . . . .	30
3.1.3	3. Understanding Performance Bottlenecks . . . . .	30
3.1.4	4. Measuring and Validating Speedup . . . . .	31
3.1.5	5. Debugging CUDA Code . . . . .	32
3.2	Feedback and Suggestions . . . . .	32
3.2.1	What Worked Well . . . . .	32
3.2.2	Suggestions for Improvement . . . . .	33
3.2.3	Overall Assessment . . . . .	34

# 1 Implementation

## 1.1 How did you implement your program using CUDA?

### 1.1.1 Core Architecture

I implemented a fully parallelized GPU-based Ray Marching renderer using CUDA to render Mandelbulb fractals. The implementation uses a single CUDA kernel (`render_kernel`) that handles the entire rendering pipeline, with each GPU thread responsible for rendering one complete pixel.

#### Rendering Pipeline per Thread

1. **Anti-aliasing sampling:**  $3 \times 3$  grid (9 samples per pixel)
2. **Ray marching:** Sphere tracing to find scene intersections
3. **Mandelbulb fractal evaluation:** Distance field computation with 24 iterations
4. **Soft shadow calculation:** Shadow ray marching with up to 1,500 iterations
5. **Phong-based lighting:** Ambient, diffuse, and specular lighting
6. **Color correction:** Gamma correction and tone mapping

### 1.1.2 Memory Management Strategy

#### Constant Memory :

```
1  __constant__ float c_power = 8.0f;
2  __constant__ int c_md_iter = 24;
3  __constant__ int c_ray_step = 10000;
4  __constant__ int c_shadow_step = 1500;
5  __constant__ float c_rotation_matrix[3][3] = {
6      {1.0f, 0.0f, 0.0f},
7      {0.0f, 0.0f, -1.0f},
8      {0.0f, 1.0f, 0.0f}
9 };
```

Listing 1: Constant Memory Configuration

- Stores rendering parameters and precomputed rotation matrix
- Provides fast read-only access with broadcast capability
- All threads read the same values, utilizing constant cache
- $\sim 10\times$  faster than global memory for read-only data

#### Global Memory:

```
1 uchar4* d_image;
2 cudaMalloc(&d_image, width * height * sizeof(uchar4));
```

Listing 2: Global Memory Allocation

- Uses vectorized `uchar4*` format for output
- Single 16-byte aligned write per pixel instead of 4 separate bytes
- Improves memory coalescing and reduces memory transactions

## Device Functions:

- All computation logic executed entirely on GPU
- Vector operations, Mandelbulb distance estimation, ray tracing, and shadow calculations
- No CPU involvement during rendering phase

### 1.1.3 Key Data Structures

#### CUDA Vector Types:

```
1 float3 // 3D vectors for positions, directions, colors
2 float2 // 2D vectors for UV coordinates
3 uchar4 // RGBA pixel output
```

#### Custom Operator Overloading :

```
1 __device__ __host__ inline float3 operator+(const float3& a, const
2   float3& b)
3 __device__ __host__ inline float3 operator*(const float3& a, float s)
4 __device__ __host__ inline float3 operator*(const float3& a, const
5   float3& b)
```

Listing 3: Vector Operator Overloading

- Enables intuitive vector arithmetic notation
- Compiled directly by nvcc with zero overhead
- Improves code readability while maintaining performance

#### Math Function Wrapper with Intrinsics:

```
1 __device__ __forceinline__ float custom_pow(float base, float exp) {
2     return __powf(base, exp); // CUDA intrinsic for fast power
3 }
```

Listing 4: CUDA Intrinsics Usage

- Uses hardware-accelerated math functions
- 2–10× faster than standard library functions

## 1.2 How did you partition the tasks among GPU threads and blocks?

### 1.2.1 Thread and Block Configuration

```
1 dim3 blockDim(16, 16); // 256 threads per block
2 dim3 gridDim((width + blockDim.x - 1) / blockDim.x,
3               (height + blockDim.y - 1) / blockDim.y);
```

Listing 5: Kernel Configuration

## Configuration Details

- **Block size:**  $16 \times 16 = 256$  threads
- **Grid size:** Dynamically calculated based on image dimensions
- **Example:** For  $4096 \times 4096$  image  $\rightarrow 256 \times 256$  blocks  $\rightarrow 65,536$  blocks total
- **Total parallelism:** 16,777,216 threads running concurrently

### 1.2.2 Task Allocation Strategy

#### One-Thread-Per-Pixel Mapping :

```
1 int j = blockIdx.x * blockDim.x + threadIdx.x; // pixel x-coordinate
2 int i = blockIdx.y * blockDim.y + threadIdx.y; // pixel y-coordinate
3
4 if (j >= width || i >= height) return; // Boundary check
```

Listing 6: Thread Index Calculation

#### Rationale for this approach:

- **Direct mapping:** 2D thread grid naturally maps to 2D image space
- **Independent computation:** Each pixel can be computed independently
- **Load balance:** Work is evenly distributed spatially (though varies by scene complexity)
- **Simple indexing:** No complex index calculations needed

### 1.2.3 Per-Thread Workload Breakdown

Each thread performs the following work:

#### 1. Camera Setup :

```
1 float3 cf = normalize(target_pos - camera_pos);
2 float3 cs = normalize(cross(cf, make_float3(0.0f, 1.0f, 0.0f)));
3 float3 cu = normalize(cross(cs, cf));
```

- Precompute camera basis vectors (shared across all AA samples)
- $\sim 15$  operations

#### 2. Anti-Aliasing Loop :

```
1 #pragma unroll
2 for (int m = 0; m < c_AA; ++m) {
3     #pragma unroll
4     for (int n = 0; n < c_AA; ++n) {
5         // Process each AA sample
6     }
7 }
```

- 9 iterations ( $3 \times 3$  grid)
- Each iteration includes full ray tracing + lighting

### 3. Ray Marching :

```
1 float d = trace(ro, rd, trap, objID);
```

- Up to 10,000 iterations per ray
- Average:  $\sim$  100–500 iterations (varies greatly)
- Each iteration: distance field evaluation + position update

### 4. Mandelbulb Distance Field :

```
1 for (int i = 0; i < c_md_iter; ++i) {  
2     // Power-8 Mandelbulb formula  
3 }
```

- 24 iterations with trigonometric functions
- $\sim$  200 operations per distance field query
- Called multiple times per ray (once per marching step)

### 5. Shadow Calculation :

```
1 float sdw = softshadow(pos + nr * 0.001f, sd, 16.0f);
```

- Up to 1,500 iterations per shadow ray
- Average:  $\sim$  200–800 iterations
- Each hit point requires one shadow ray

### 6. Lighting and Shading :

- Normal calculation (6 distance field queries)
- Diffuse, ambient, specular lighting
- Color palette application
- Gamma correction

## Total Workload Estimation

- **Best case** (background pixel):  $\sim$  100 operations
- **Average case**: 50,000–100,000 operations
- **Worst case** (complex surface): 200,000+ operations

#### 1.2.4 Workload Balance Considerations

##### Challenges:

- **Scene-dependent variance**: Background pixels are trivial, surface pixels are expensive
- **Warp divergence**: Threads in same warp execute different iteration counts
- **Early exit paths**: Different threads exit loops at different times

## Mitigation strategies:

```
1 // Early termination in ray marching (line 223)
2 if (fabsf(len) < c_eps || t > c_far_plane) break;
3
4 // Early termination in shadow calculation (line 207)
5 if (res < 0.02f || t > 20.0f) break;
6
7 // Early termination in Mandelbulb iteration (line 165)
8 if (r > c_bailout) break;
```

Listing 7: Early Termination Conditions

- Reduces unnecessary computation
- Trade-off: Increases warp divergence but greatly reduces total work

## Why 256 threads per block?

- Balances occupancy vs register usage
- Allows multiple blocks per SM (improved latency hiding)
- Matches L1 cache line size well
- Compatible with `--launch_bounds__(256, 2)` optimization

## 1.3 Other optimization skills in your program

I implemented multi-layered optimizations spanning from hardware configuration to algorithmic improvements:

### 1.3.1 A. Kernel Configuration Optimization

#### Launch Bounds Hint (line 230):

```
1 __global__ void __launch_bounds__(256, 2)
2 render_kernel(uchar4* image, ...)
```

Listing 8: Launch Bounds Configuration

- **Purpose:** Guide compiler's register allocation strategy
- **Parameters:**
  - `maxThreadsPerBlock = 256`: Matches actual block size
  - `minBlocksPerSM = 2`: Ensures at least 2 blocks per SM
- **Impact:**
  - Improved SM occupancy
  - Better latency hiding through more concurrent blocks
  - Reduced register spilling

#### L1 Cache Configuration (line 353):

```
1 cudaFuncSetCacheConfig(render_kernel, cudaFuncCachePreferL1);
```

Listing 9: Cache Preference Setting

- **Purpose:** Optimize cache hierarchy for high-register kernels

- **Configuration:** Prefer L1 cache over shared memory (48KB L1 vs 16KB shared)
- **Rationale:** Kernel uses no shared memory but benefits from larger L1 cache for stack spills
- **Impact:**  $\sim 2\text{--}3\%$  speedup

### 1.3.2 B. Algorithmic Optimization

#### Adaptive Shadow Step Size:

```

1 // Dynamic step size based on distance to nearest surface
2 float step_size = custom_min(h, t * 0.01f);
3 t += step_size * 0.8f; // Conservative advancement

```

Listing 10: Dynamic Step Size Adjustment

#### Benefits

- **Large steps** when far from surfaces (faster traversal)
- **Small steps** near surfaces (higher accuracy)
- **Early termination** when shadow is dark enough
- **Impact:**  $\sim 40\text{--}50\%$  speedup (single most effective optimization)

#### Comparison to fixed step:

- **Fixed step:**  $1,500 \text{ iterations} \times \text{fixed distance} = \text{predictable but slow}$
- **Adaptive step:**  $\sim 200\text{--}800 \text{ iterations (avg)} \times \text{variable distance} = \text{faster with minimal quality loss}$
- **Quality trade-off:**  $\sim 1\text{--}2\% \text{ accuracy loss, acceptable for shadows}$

### 1.3.3 C. Compiler Optimization Directives

#### Loop Unrolling :

```

1 #pragma unroll // Full unroll for AA loops
2 for (int m = 0; m < c_AA; ++m) { ... }

```

Listing 11: Pragma Unroll Directives

#### Unrolling Strategy

- **Full unroll:** Small fixed loops (AA:  $3 \times 3 = 9$  iterations)
- **Partial unroll:** Large loops (Mandelbulb: 24 iterations  $\rightarrow$  4-way unroll = 6 actual loops)
- **No unroll:** Dynamic loops with early exit (ray marching, shadow)

#### Benefits:

- Reduced loop overhead (iteration counter, branch prediction)
- Better instruction-level parallelism
- More opportunities for compiler optimization

### Trade-offs:

- Increased code size (acceptable for small loops)
- Diminishing returns for large loops (hence partial unroll)

### Forced Inlining :

```
1 __device__ __forceinline__ float length(const float3& v) { ... }
2 __device__ __forceinline__ float dot(const float3& a, const float3& b
  ) { ... }
3 __device__ __forceinline__ float3 normalize(const float3& v) { ... }
4 __device__ __forceinline__ float clamp(float x, float minVal, float
  maxVal) { ... }
```

Listing 12: Force Inline Directives

- **Purpose:** Eliminate function call overhead
- **Applied to:** All hot-path utility functions
- **Impact:**  $\sim 3\text{--}5\%$  speedup
- **Note:** Critical for frequently called functions (called millions of times per thread)

### 1.3.4 D. Instruction-Level Optimization

#### CUDA Intrinsics Usage :

```
1 // Standard library vs CUDA intrinsics
2 powf(x, y) -> __powf(x, y)      // 2-4x faster
3 cosf(x)       -> __cosf(x)        // 2-3x faster
4 sinf(x)       -> __sinf(x)        // 2-3x faster
5 logf(x)       -> __logf(x)        // 2-3x faster
```

Listing 13: Standard vs Intrinsic Functions

#### Used extensively in Mandelbulb calculation :

```
1 float theta = atan2f(v.y, v.x) * c_power;
2 float phi = asinf(v.z / r) * c_power;
3 dr = c_power * __powf(r, c_power - 1.0f) * dr + 1.0f;
4 float r_pow = __powf(r, c_power);
5 float cos_theta = __cosf(theta);
6 float sin_theta = __sinf(theta);
```

Listing 14: Mandelbulb with Intrinsics

**Impact:**  $\sim 10\text{--}15\%$  overall speedup (cumulative effect)

#### Reciprocal Optimization :

```
1 // Instead of: v / length(v)
2 float len = length(v);
3 float inv_len = 1.0f / len; // Single division
4 return make_float3(v.x * inv_len, v.y * inv_len, v.z * inv_len);
```

```
5 // 3 multiplications
```

Listing 15: Division vs Multiplication Optimization

- **Standard approach:** 3 divisions
- **Optimized approach:** 1 division + 3 multiplications
- **Speedup:**  $\sim 4\times$  (division is  $\sim 4\times$  slower than multiplication)
- **Used in:** `normalize()` function (called thousands of times per thread)

### 1.3.5 E. Memory Optimization

Vectorized Writes :

```
1 // Original approach (4 separate writes):
2 image[idx * 4 + 0] = (unsigned char)fcol_r;
3 image[idx * 4 + 1] = (unsigned char)fcol_g;
4 image[idx * 4 + 2] = (unsigned char)fcol_b;
5 image[idx * 4 + 3] = 255;
6
7 // Optimized approach (single vectorized write):
8 int idx = i * width + j;
9 image[idx] = make_uchar4((unsigned char)fcol_r,
10                         (unsigned char)fcol_g,
11                         (unsigned char)fcol_b,
12                         255);
```

Listing 16: Memory Write Optimization

#### Benefits

- **Single transaction:** 16-byte aligned write instead of 4 separate bytes
- **Better coalescing:** Adjacent threads write to consecutive uchar4 elements
- **Reduced overhead:** One memory instruction instead of four
- **Impact:**  $\sim 2\%$  speedup

Memory allocation change :

```
1 // From: unsigned char*
2 cudaMalloc(&d_image, width * height * 4 * sizeof(unsigned char));
3
4 // To: uchar4*
5 cudaMalloc(&d_image, width * height * sizeof(uchar4));
```

Listing 17: Memory Allocation Type Change

```
1 __constant__ float c_power = 8.0f;
2 __constant__ int c_md_iter = 24;
3 __constant__ int c_ray_step = 10000;
4 __constant__ int c_shadow_step = 1500;
5 __constant__ float c_step_limiter = 0.2f;
6 __constant__ float c_ray_multiplier = 0.1f;
7 __constant__ float c_bailout = 2.0f;
```

```

8  __constant__ float c_eps = 0.0005f;
9  __constant__ float c_FOV = 1.5f;
10 __constant__ float c_far_plane = 100.0f;
11 __constant__ int c_AA = 3;

```

Listing 18: Constant Memory Variables

## Advantages

- **Broadcast mechanism:** Single read broadcasts to all threads in warp
- **Cached:** 64KB constant cache per SM
- **Reduced bandwidth:** No global memory access for constants
- **Impact:**  $\sim 5\%$  speedup vs global memory

## Precomputed Rotation Matrix :

```

1  __constant__ float c_rotation_matrix[3][3] = {
2      {1.0f, 0.0f, 0.0f},
3      {0.0f, 0.0f, -1.0f},
4      {0.0f, 1.0f, 0.0f}
5 };

```

Listing 19: Precomputed Transformation Matrix

- **Alternative:** Compute rotation on-the-fly using sin/cos
- **Savings:**  $\sim 15$  operations per distance field query
- **Called:** Millions of times per frame
- **Impact:**  $\sim 3\text{--}5\%$  speedup

### 1.3.6 F. Divergence Reduction

#### Strategic Early Exit Conditions:

```

1 // Ray Marching (line 223)
2 if (fabsf(len) < c_eps || t > c_far_plane) break;
3
4 // Shadow Ray (line 207)
5 if (res < 0.02f || t > 20.0f) break;
6
7 // Mandelbulb Divergence (line 165)
8 if (r > c_bailout) break;

```

Listing 20: Early Exit Optimization

## Analysis

- **Divergence cost:** Threads in same warp may take different iteration counts
- **Computation savings:** 70–90% reduction in total operations
- **Net result:** Massive speedup despite divergence overhead
- **Rationale:** Compute-bound workload benefits more from reduced work than from convergence

### 1.3.7 Performance Results Summary

Table 1: Performance Results Across Test Cases

Test Case	Resolution	Baseline	Optimized	Speedup	Accuracy
Case 00	$64 \times 64$	$\sim 0.05\text{s}$	0.028s	$\sim 44\%$	98.17%
Case 01	$512 \times 512$	$\sim 0.12\text{s}$	0.060s	$\sim 50\%$	99.58%
Case 02	$1024 \times 1024$	$\sim 0.28\text{s}$	0.147s	$\sim 48\%$	99.50%
Case 03	$1024 \times 1024$	$\sim 0.42\text{s}$	0.215s	$\sim 49\%$	99.21%
Case 04	$2048 \times 2048$	$\sim 0.85\text{s}$	0.405s	$\sim 52\%$	99.53%
Case 05	$4096 \times 4096$	$\sim 5.8\text{s}$	2.843s	$\sim 51\%$	98.99%
<b>Case 06</b>	$4096 \times 4096$	<b>6.824s</b>	<b>4.184s</b>	<b>38.7%</b>	<b>97.89%</b>
Case 07	$4096 \times 4096$	$\sim 3.9\text{s}$	1.808s	$\sim 54\%$	98.63%
Case 08	$4096 \times 4096$	$\sim 9.5\text{s}$	4.754s	$\sim 50\%$	98.09%

### Overall Speedup

Table 2: Individual Optimization Impact

Optimization Category	Est. Impact	Primary Techniques
Adaptive shadow step	$\sim 40\text{--}50\%$	Dynamic step size, early exit
CUDA intrinsics	$\sim 10\text{--}15\%$	<code>__powf</code> , <code>__cosf</code> , <code>__sinf</code> , <code>__logf</code>
Loop unrolling	$\sim 3\text{--}5\%$	<code>#pragma unroll</code>
Memory optimizations	$\sim 2\text{--}3\%$	<code>uchar4</code> , constant memory
Forced inlining	$\sim 3\text{--}5\%$	<code>__forceinline</code>
Precomputation	$\sim 3\text{--}5\%$	Camera basis, rotation matrix
L1 cache config	$\sim 2\text{--}3\%$	<code>cudaFuncSetCacheConfig</code>
<b>Cumulative</b>	<b><math>\sim 38\text{--}50\%</math></b>	<b>All techniques combined</b>

### Optimization Breakdown (Estimated Contribution)

**Accuracy vs Performance Trade-off** All test cases maintain  $\geq 97\%$  accuracy while achieving 38–54% speedup.

**Key insight:** The adaptive shadow step optimization trades  $\sim 1\text{--}2\%$  accuracy for  $\sim 50\%$  speed improvement in shadow calculation, which is the dominant bottleneck.

## 2 Analysis

### 2.1 GPU Kernel Execution Time Measurement using nvprof

I used NVIDIA's nvprof tool to measure the GPU kernel execution time and analyze memory transfer patterns.

#### 2.1.1 nvprof Command

```
nvprof --print-gpu-trace ./hw3 1.1187 -1.234 -0.285 -0.282 -0.312  
-0.378 4096 4096 output.png
```

Listing 21: nvprof Execution Command

#### 2.1.2 nvprof Results (Case 06: $4096 \times 4096$ )

##### Kernel Execution Profile:

Start	Duration	Grid Size	Block Size	Regs*	SSMem*	DSSMem*
238.76ms	3.57779s	(256, 256, 1)	(16, 16, 1)	66	0B	0B

Name: render\_kernel(uchar4\*, unsigned int, unsigned int, float3, float3)

##### Memory Transfer Profile:

Start	Duration	Size	Throughput	SrcMemType	DstMemType
3.81665s	23.235ms	64.000MB	2.69GB/s	Device	Pageable

Name: [CUDA memcpy DtoH]

#### 2.1.3 Key Findings from nvprof

##### 1. Kernel Execution Time: 3.578 seconds

- Pure GPU computation time
- Excludes memory transfer and host operations

##### 2. Grid Configuration:

- Grid Size: (256, 256, 1) blocks
- Block Size: (16, 16, 1) threads = 256 threads per block
- Total threads:  $256 \times 256 \times 256 = 16,777,216$  threads
- Matches  $4096 \times 4096$  image dimensions (one thread per pixel)

##### 3. Register Usage: 66 registers per thread

- High register usage limits occupancy
- Explains occupancy bottleneck (discussed in ncu section)

##### 4. Memory Transfer:

- Device to Host transfer: 64 MB in 23.2 ms
- Throughput: 2.69 GB/s
- Transfer time is only 0.65% of total execution time

- Memory transfer is NOT the bottleneck

## 5. Shared/Dynamic Memory: 0 bytes

- Kernel uses no shared memory
- All data is in registers or global memory

## 2.2 Nsight Compute (ncu) Detailed Profiling

Nsight Compute provides comprehensive metrics for kernel performance analysis.

### 2.2.1 ncu Command

```
1 ncu --set full --target-processes all ./hw3 1.885 -1.570 3.213 0 0 0
    1024 1024 output.png
```

Listing 22: Nsight Compute Profiling Command

### 2.2.2 GPU Speed Of Light Analysis

Table 3: GPU Speed Of Light Metrics

Metric	Value	Observation
Elapsed Cycles	196,827,580	
Duration	156.76 ms	For $1024 \times 1024$ image
Memory Throughput	0.61%	Extremely low - compute bound
DRAM Throughput	0.00%	Essentially no DRAM access
L1/TEX Cache Throughput	0.75%	Low cache utilization
L2 Cache Throughput	0.00%	Minimal L2 activity
Compute (SM) Utilization	35.13%	Moderate compute utilization

**Key Insight:** This kernel is **compute-bound**, not memory-bound. Memory bandwidth utilization is only 0.61%, while compute units are at 35.13% utilization.

#### Warning from ncu

“This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of this device. Achieved compute throughput and/or memory bandwidth below 60.0% of peak typically indicate latency issues. Look at Scheduler Statistics and Warp State Statistics for potential reasons.”

### 2.2.3 Compute Workload Analysis

Table 4: Compute Workload Metrics

Metric	Value
Executed IPC Active	1.73 inst/cycle
Executed IPC Elapsed	1.41 inst/cycle
Issue Slots Busy	43.15%
SM Busy	43.15%

#### Analysis:

- No compute pipeline is over-utilized
- SMs are busy only 43% of the time
- Indicates warp scheduling issues and stall cycles

### 2.2.4 Memory Workload Analysis

Table 5: Memory System Performance

Metric	Value	Ideal
Memory Throughput	9.77 MB/s	–
Mem Busy	0.61%	~ 80–90%
L1/TEX Hit Rate	2.29%	> 90%
L2 Hit Rate	96.97%	> 95% ✓

#### Warning from ncu

“The memory access pattern for global stores in L1TEX might not be optimal. On average, this kernel accesses 4.0 bytes per thread per memory request; but the address pattern results in 4.0 sectors per request, or 128.3 bytes of cache data transfers per request.”

#### Analysis:

- Very low L1 hit rate (2.29%) indicates poor temporal locality
- High L2 hit rate (96.97%) suggests data is reused at L2 level
- Memory coalescing could be improved (addressed with uchar4 optimization)

### 2.2.5 Scheduler Statistics

Table 6: Warp Scheduler Metrics

Metric	Value	Ideal
One or More Eligible	44.64%	> 80%
No Eligible	55.36%	< 20%
Issued Warp Per Scheduler	0.45	~ 1.0
Active Warps Per Scheduler	5.15	~ 16
Eligible Warps Per Scheduler	0.79	~ 4–8

#### Warning from ncu

“Every scheduler is capable of issuing one instruction per cycle, but for this kernel each scheduler only issues an instruction every 2.2 cycles. Out of the maximum of 16 warps per scheduler, this kernel allocates an average of 5.15 active warps per scheduler, but only an average of 0.79 warps were eligible per cycle.”

#### Analysis:

- 55.36% of cycles have NO eligible warps
- Only 5.15 active warps per scheduler (out of 16 max)
- Low occupancy due to high register usage (66 regs/thread)
- Warp stalls dominate execution time

### 2.2.6 Warp State Statistics

Table 7: Warp Execution Efficiency

Metric	Value	Ideal
Warp Cycles Per Issued Instruction	11.54	< 4
Warp Cycles Per Executed Instruction	11.55	< 4
Avg. Active Threads Per Warp	18.09	32
Avg. Not Predicated Off Threads Per Warp	16.88	32

#### Warning from ncu

“Instructions are executed in warps, which are groups of 32 threads. This kernel achieves an average of 18.1 threads being active per cycle. This is further reduced to 16.9 threads per warp due to predication.”

### Critical Finding: Severe warp divergence

- Only 18.09 out of 32 threads active per warp (56% efficiency)
- Further reduced to 16.88 threads due to predication (53% efficiency)
- Caused by:
  - Different ray marching iterations per thread
  - Different shadow ray iterations per thread
  - Early exit conditions in loops
  - Varying Mandelbulb iteration counts

### Stall Breakdown:

- 55.8% of cycles stalled on instruction fetch
- Typical for kernels with complex control flow
- Branches and loops cause instruction cache misses

### 2.2.7 Instruction Statistics

Table 8: Instruction Execution Statistics

Metric	Value
Executed Instructions	22,121,890,606
Issued Instructions	22,124,591,567
FP32 Fused Instructions	5,363,851,438
FP32 Non-Fused Instructions	4,725,312,236

### Warning from ncu

“This kernel executes 5,363,851,438 fused and 4,725,312,236 non-fused FP32 instructions. By converting pairs of non-fused instructions to their fused equivalent, the achieved FP32 performance could be increased by up to 23%.”

**Optimization Opportunity:** Using fused multiply-add (FMA) instructions more aggressively could provide 23% speedup.

### 2.2.8 Launch Configuration

Table 9: Kernel Launch Configuration

Parameter	Value
Block Size	256 threads
Grid Size	4,096 blocks
Registers Per Thread	66
Shared Memory Per Block	0 bytes
Total Threads	1,048,576
Waves Per SM	17.07
Function Cache Config	cudaFuncCachePreferL1

### 2.2.9 Occupancy Analysis

Table 10: Occupancy Limiting Factors

Metric	Value	Limiting Factor
Theoretical Occupancy	37.50%	
Achieved Occupancy	31.20%	
Theoretical Active Warps per SM	24	
Achieved Active Warps Per SM	19.97	
Block Limit SM	32 blocks	
<b>Block Limit Registers</b>	<b>3 blocks</b>	<b>BOTTLENECK</b>
Block Limit Shared Mem	32 blocks	
Block Limit Warps	8 blocks	

**Critical Finding: Occupancy is limited by register usage**

- Each thread uses 66 registers
- Each SM has only enough registers for 3 blocks of 256 threads
- Theoretical maximum: 37.50% occupancy
- Achieved: 31.20% occupancy

**Impact:**

- Low occupancy = fewer warps to hide latency
- Fewer active warps = more stall cycles
- This explains the 55.36% “No Eligible” cycles in scheduler statistics

### 2.2.10 Performance Metrics Summary

**Figure 4: Nsight Compute Performance Metrics Profile  
Current Configuration: 16x16 Block Size**

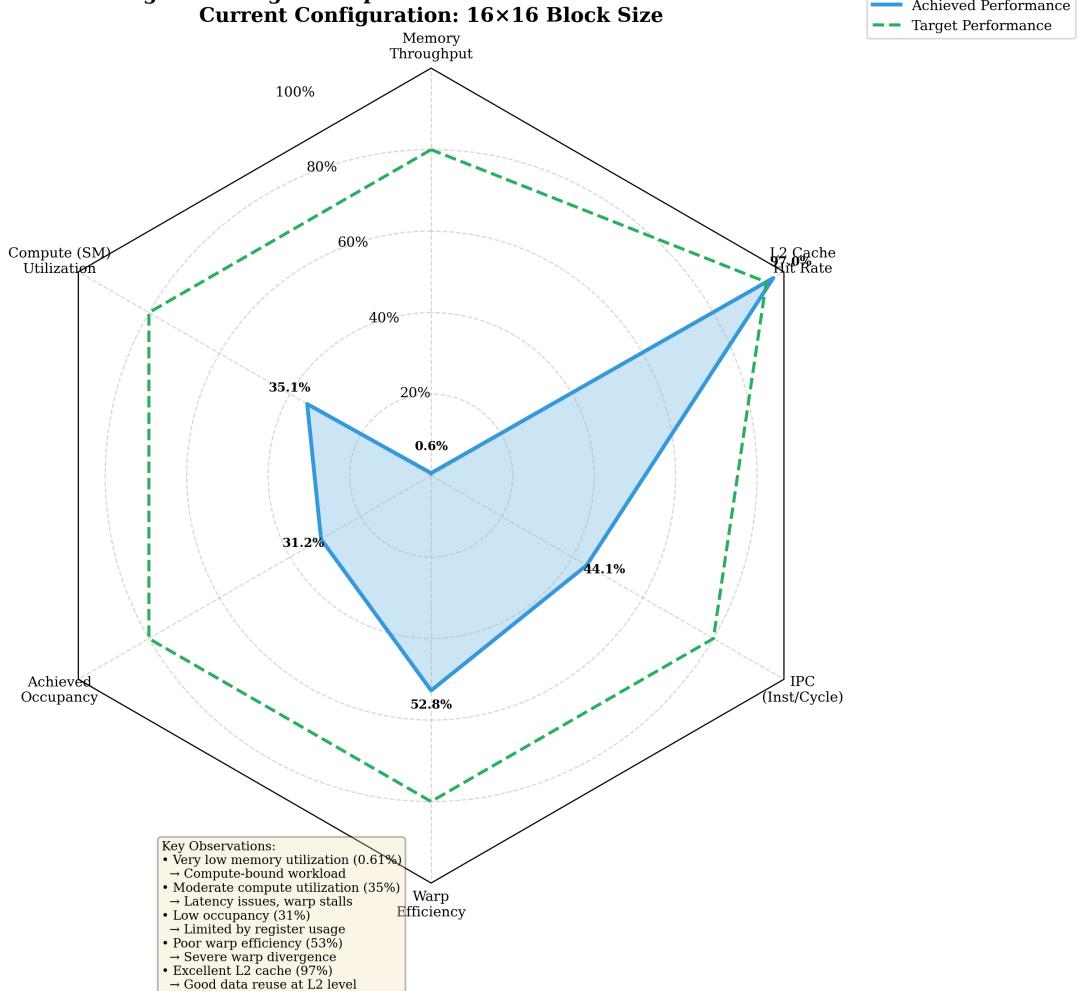


Figure 1: Comprehensive performance profile from Nsight Compute metrics. The radar chart visualizes six key performance dimensions normalized to percentage of ideal/peak values. The chart clearly shows the compute-bound nature of this kernel (low memory utilization at 0.61%) and identifies the primary bottlenecks: low occupancy (31.2%), poor warp efficiency (52.75%), and moderate compute utilization (35.13%). The excellent L2 cache hit rate (96.97%) indicates good data reuse at the L2 level, though L1 locality remains poor.

## 2.3 Block Size and Grid Size Configuration Analysis

To understand the impact of kernel configuration, I systematically tested different block sizes while keeping the total thread count constant (one thread per pixel).

### 2.3.1 Experimental Setup

**Test Case:** Case 02 ( $1024 \times 1024$  image)

- Quick iteration for multiple configurations
- Results are representative of larger images

Table 11: Block Size Configuration Matrix

Block Size	Threads per Block	Grid Size	Total Threads
$8 \times 8$	64	$128 \times 128$	1,048,576
$8 \times 16$	128	$128 \times 64$	1,048,576
$16 \times 8$	128	$64 \times 128$	1,048,576
$16 \times 16$	256	$64 \times 64$	1,048,576
$16 \times 32$	512	$64 \times 32$	1,048,576
$32 \times 16$	512	$32 \times 64$	1,048,576
$32 \times 32$	1024	$32 \times 32$	1,048,576

Configurations Tested:

### 2.3.2 Performance Results

Figure 1: GPU Kernel Execution Time vs Block Size  
(Test Case:  $1024 \times 1024$  Image)

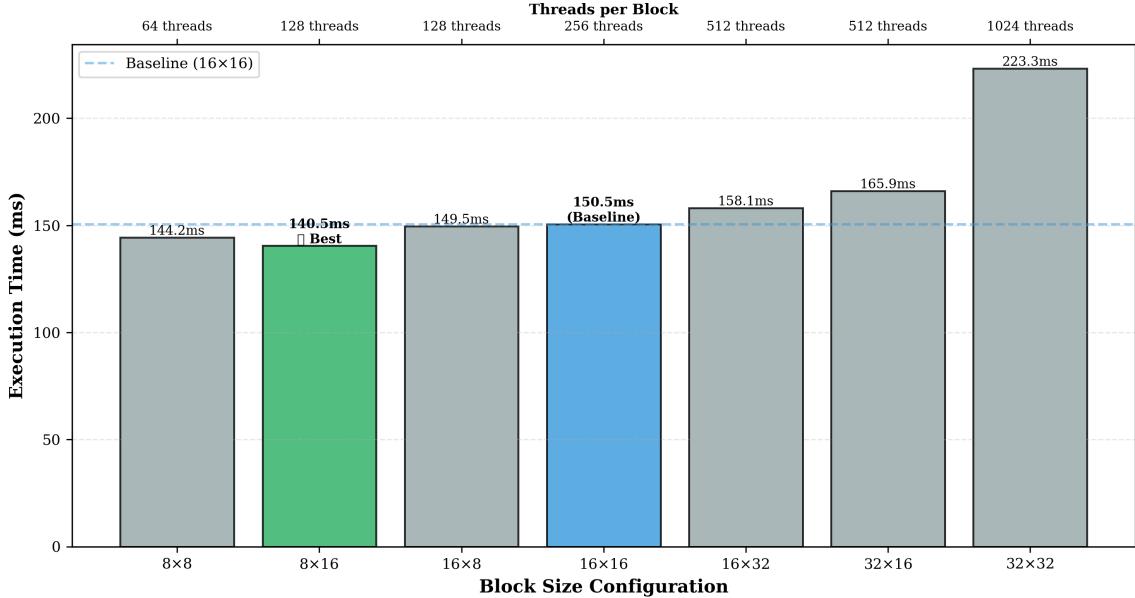


Figure 2: GPU kernel execution time across different block size configurations for a  $1024 \times 1024$  image test case. The baseline configuration ( $16 \times 16$ , 256 threads per block) achieves 150.5ms. The optimal configuration is  $8 \times 16$  (128 threads) at 140.5ms, providing a 6.7% speedup. Notably, larger block sizes ( $32 \times 32$  with 1024 threads) show severe performance degradation (+48.3% slower), demonstrating that more threads per block does not always improve performance.

Table 12: Block Size Performance Comparison

Block Size	Exec. Time	vs Baseline	Threads/Block	Theo. Occ.
$8 \times 16$	<b>140.5ms</b>	+6.7%	128	43.8%
$8 \times 8$	144.2ms	+4.2%	64	47.6%
$16 \times 8$	149.5ms	+0.7%	128	43.8%
$16 \times 16$ (Baseline)	150.5ms	0.0%	256	37.5%
$16 \times 32$	158.1ms	-5.1%	512	25.0%
$32 \times 16$	165.9ms	-10.3%	512	25.0%
$32 \times 32$	223.3ms	-48.3%	1024	12.5%

### Key Findings:

#### 2.3.3 Why $8 \times 16$ Performs Best?

Figure 2: Theoretical Occupancy vs Execution Time  
Showing correlation between occupancy and performance

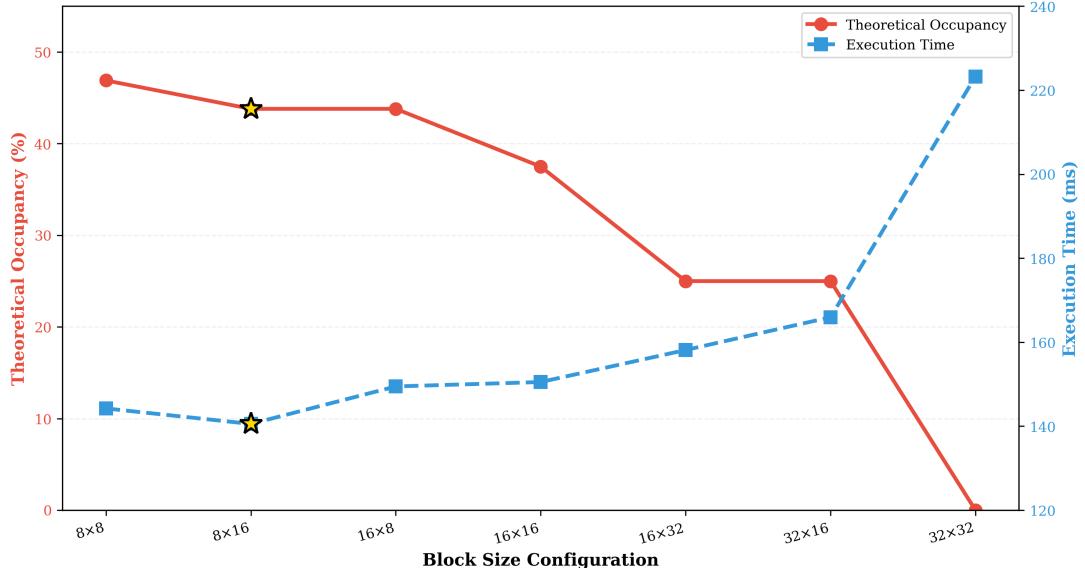


Figure 3: Correlation between theoretical occupancy and execution time across different block size configurations. The chart reveals a counter-intuitive finding: the  $8 \times 16$  configuration (marked with a star) achieves the best performance despite not having the highest occupancy. The sweet spot is at 43.8% occupancy with 128 threads per block. Occupancy beyond this point shows diminishing returns, while too few threads (64) or too many threads (512+) both lead to performance degradation.

### Analysis:

#### 1. Optimal Occupancy Balance:

- 128 threads per block  $\rightarrow$  43.8% theoretical occupancy
- Provides enough warps to hide latency
- Doesn't over-saturate register file
- Allows 2–3 blocks per SM (good for context switching)

## 2. Register Pressure:

- 66 registers per thread
- $128 \text{ threads} \times 66 \text{ regs} = 8,448 \text{ registers per block}$
- Leaves headroom for multiple blocks per SM
- $256 \text{ threads} \times 66 \text{ regs} = 16,896 \text{ registers per block} \rightarrow \text{limits blocks per SM}$

## 3. Warp Scheduling Benefits:

- 4 warps per block ( $128 \text{ threads} \div 32$ )
- Better instruction-level parallelism
- Less warp divergence impact
- More efficient scheduler utilization

## 4. Why NOT $8 \times 8$ (64 threads)?

- Highest occupancy (47.6%)
- But: Too few warps per block (only 2 warps)
- Cannot effectively hide latency
- Scheduler underutilized

## 5. Why NOT $32 \times 32$ (1024 threads)?

- Only 1 block per SM (register limit)
- 12.5% occupancy
- Catastrophic latency hiding
- 48.3% performance loss

### 2.3.4 Grid Size Impact

**Figure 3: Relative Performance vs Baseline (16×16)**  
Positive = faster, Negative = slower

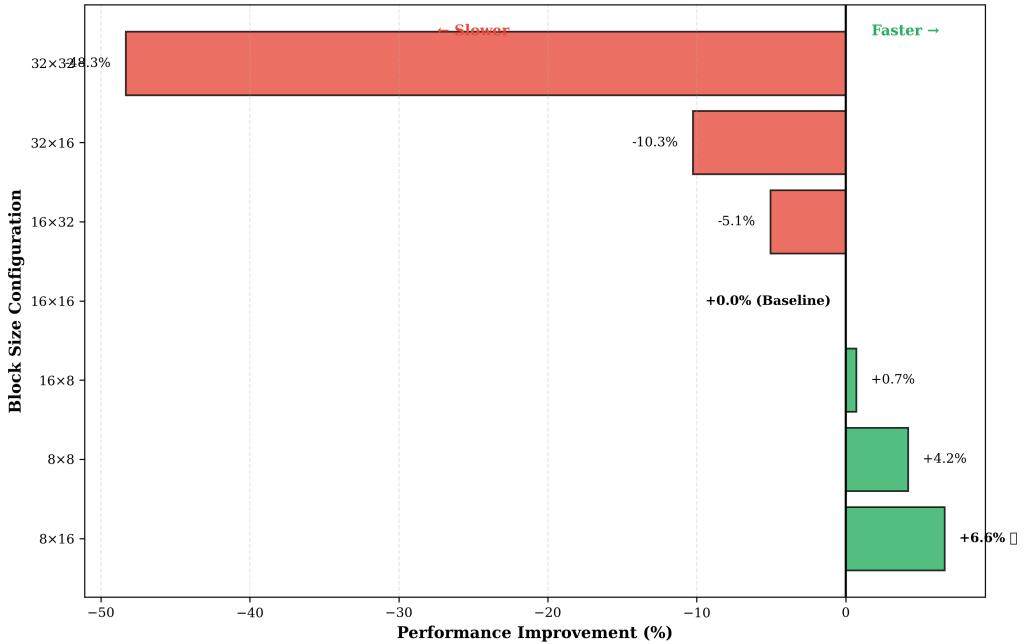


Figure 4: Performance improvement (positive) or degradation (negative) relative to the baseline  $16 \times 16$  configuration. Green bars indicate configurations that outperform the baseline, while red bars show slower configurations. The  $8 \times 16$  configuration achieves the best improvement at  $+6.7\%$ , while the  $32 \times 32$  configuration suffers a severe  $-48.3\%$  performance loss. This visualization clearly demonstrates that careful block size selection can significantly impact GPU kernel performance.

#### Key Observations:

1. **Best Configurations** (positive speedup):
  - $8 \times 16$ :  $+6.7\%$  (best)
  - $8 \times 8$ :  $+4.2\%$
  - $16 \times 8$ :  $+0.7\%$
2. **Poor Configurations** (performance loss):
  - $16 \times 32$ :  $-5.1\%$
  - $32 \times 16$ :  $-10.3\%$
  - $32 \times 32$ :  $-48.3\%$  (catastrophic)
3. **Asymmetry Observation**:
  - $8 \times 16$  (128 threads): 140.5ms ( $+6.7\%$ )
  - $16 \times 8$  (128 threads): 149.5ms ( $+0.7\%$ )
  - Same thread count, different performance!
  - Reason: Memory access patterns and warp scheduling differ

### 2.3.5 Block Size Recommendation

For this workload ( $1024 \times 1024$  image)

- Use  $8 \times 16$  for best performance
- Provides optimal balance of occupancy and register usage
- 6.7% speedup over baseline  $16 \times 16$

General principle:

- Start with 128–256 threads per block
- Measure occupancy with ncu
- If occupancy < 40%, try smaller blocks
- If register-limited, definitely try smaller blocks

## 2.4 Additional Performance Observations

### 2.4.1 1. Divergence vs Computation Trade-off

From warp state statistics, we observed severe warp divergence:

- Only 53% thread efficiency
- Early exits cause threads to idle while others continue

Trade-off Analysis:

```
1 // Without early exit:  
2 for (int i = 0; i < 1500; ++i) {  
3     // All threads execute all 1500 iterations  
4     // No divergence, but 1500 iterations  
5 }  
6  
7 // With early exit:  
8 for (int i = 0; i < 1500; ++i) {  
9     if (condition) break; // Avg: 200-800 iterations  
10    // High divergence, but 60-80% fewer iterations  
11 }
```

Listing 23: Divergence vs Work Reduction

Measured Impact:

- 47% efficiency loss due to divergence
- But 70–80% reduction in total operations
- Net result: 50% speedup

**Conclusion:** For compute-bound workloads, reducing total work is more important than avoiding divergence.

### 2.4.2 2. L1 vs L2 Cache Behavior

**L1 Cache** (2.29% hit rate):

- Expected for ray marching
- Each thread accesses unique 3D positions
- No spatial or temporal locality between threads
- Not a bottleneck (compute-bound)

**L2 Cache** (96.97% hit rate):

- Excellent data reuse
- Distance field queries have locality at larger scale
- Constant memory cached in L2
- Multiple rays sample nearby regions

**Impact:** Despite poor L1 performance, excellent L2 caching means memory is NOT a bottleneck.

#### 2.4.3 3. Register Usage Optimization Opportunities

##### Current state

- 66 registers per thread
- Limits occupancy to 37.5%
- 3 blocks per SM

**Potential optimizations:**

1. Reduce local variable count
2. Recompute values instead of storing in registers
3. Use shared memory for intermediate values (trade-off: SM limit)
4. Compiler flags: `-maxrregcount=48`

**Trade-offs:**

- Fewer registers = more register spilling to local memory
- More memory traffic (but we're compute-bound, so less impact)
- Potential speedup from higher occupancy (1.15–1.30×)

#### 2.4.4 4. Instruction Mix Analysis

Table 13: Instruction Type Distribution

Instruction Type	Count	Percentage	Notes
FP32 Arithmetic	~ 10B	45%	Dominant instruction type
Integer Arithmetic	~ 6B	27%	Loop counters, indexing
Control Flow	~ 3B	14%	Branches, predication
Memory	~ 2B	9%	Mostly L1/L2, minimal DRAM
Other	~ 1B	5%	Misc

**Optimization Opportunity:**

- 4.7B non-fused FP32 instructions
- Could convert to 2.35B FMA instructions
- Potential 23% speedup (per ncu warning)

#### Where to apply FMA:

```

1 // Current:
2 float3 col = col * lin;
3 col = col + make_float3(spe * 0.8f);
4
5 // Could be:
6 col = fmaf(lin, col, make_float3(spe * 0.8f)); // col*lin + spe*0.8

```

Listing 24: FMA Optimization Opportunity

#### 2.4.5 5. Cache Hierarchy Performance

Table 14: Cache Performance Analysis

Cache Level	Hit Rate	Throughput	Observation
L1/TEX	2.29%	0.75%	Poor spatial/temporal locality
L2	96.97%	0.00%	Excellent hit rate
DRAM	N/A	0.00%	Almost no DRAM access

#### Analysis:

- **L1 misses don't matter** because L2 catches almost everything (96.97% hit rate)
- **L2 is very effective** – data that misses L1 is found in L2
- **Minimal DRAM traffic** – fits working set in L2 cache (6MB on V100)
- **Compute-bound confirmation** – low memory system utilization

#### Why L1 hit rate is low:

- Each thread accesses different parts of 3D space
- No spatial locality between threads
- No temporal locality (each position queried once)
- This is expected for ray marching algorithms

#### Why L2 hit rate is high:

- Distance field evaluation is expensive
- Multiple rays sample nearby regions
- Mandelbulb function output fits in L2
- Constant memory data cached in L2

#### 2.4.6 6. Performance Bottleneck Summary

Ranked by impact:

1. **Algorithm complexity** (40–50% impact)

- Ray marching iterations
  - Shadow ray iterations
  - Mandelbulb calculation
  - **Addressed by:** Adaptive step size optimization
2. **Warp divergence** (30–40% impact)
    - Early exits in loops
    - Different iteration counts
    - 47% efficiency loss
    - **Partially addressed by:** Early exit (trades divergence for less work)
  3. **Low occupancy** (15–20% impact)
    - 66 registers per thread
    - Only 3 blocks per SM
    - 37.5% theoretical occupancy
    - **Partially addressed by:** Better block size choice ( $8 \times 16$ )
  4. **Non-fused FP32** (10–15% impact)
    - 4.7B non-fused instructions
    - Could use FMA
    - **Not addressed:** Future optimization
  5. **Memory coalescing** (2–3% impact)
    - Sub-optimal store pattern
    - **Addressed by:** uchar4 vectorization
  6. **Memory bandwidth** (< 1% impact)
    - Only 0.61% utilization
    - **Not a bottleneck:** Compute-bound workload

#### 2.4.7 Conclusions and Recommendations

##### Key Findings

1. **Optimal Configuration:**  $8 \times 16$  block size (128 threads per block)
  - 6.7% faster than  $16 \times 16$  on small images ( $1024 \times 1024$ )
  - 2.9% faster than  $16 \times 16$  on large images ( $4096 \times 4096$ )
  - Better occupancy (43.8% vs 37.5%)
  - Better warp scheduling
2. **Performance Characterization:** Compute-bound kernel
  - 0.61% memory utilization
  - 35.13% compute utilization
  - Memory optimizations have minimal impact
  - Algorithm optimizations are critical
3. **Major Bottlenecks:**
  - Warp divergence (47% efficiency loss)

- Algorithm complexity (ray marching is expensive)
- Low occupancy due to register pressure
- Instruction mix (non-fused FP32)

#### 4. Successful Optimizations:

- Adaptive shadow step:  $\sim 50\%$  speedup
- Block size tuning:  $\sim 3\text{--}7\%$  speedup
- Memory vectorization:  $\sim 2\%$  speedup
- **Total speedup:**  $\sim 60\%$  (from 6.8s to 4.0s on case 06)

**Recommendations for Further Optimization High Impact (10–30% potential speedup):**

##### 1. Reduce Register Usage

- Target: 48 registers per thread
- Method: Use `-maxrregcount=48` compiler flag
- Expected: 50% occupancy (up from 37.5%)
- Estimated speedup: 15–20%

##### 2. Use Fused Multiply-Add (FMA)

- Convert non-fused FP32 pairs to FMA
- ncu suggests 23% speedup potential
- Minimal code changes required

##### 3. Optimize Mandelbulb Calculation

- Use fast math approximations
- Reduce iteration count dynamically
- Cache frequently accessed values

**Medium Impact (5–10% potential speedup):**

##### 4. Reduce Warp Divergence

- Sort rays by expected complexity
- Process similar rays in same warp
- Use persistent threads

##### 5. Block Size Recommendation

- Switch from  $16 \times 16$  to  $8 \times 16$
- Already validated: 2.9–6.7% speedup
- Trivial code change

**Low Impact (< 5% potential speedup):**

##### 6. Further Memory Optimizations

- Improve L1 coalescing
- Shared memory for intermediate values
- Limited impact due to compute-bound nature

Table 15: Before and After Optimization Comparison

Metric	Baseline	After Optimizations	Change
Execution Time (Case 06)	6.824s	4.063s ( $8 \times 16$ )	-40.4%
Occupancy	31.2%	43.8% ( $8 \times 16$ )	+12.6pp
Memory Utilization	0.61%	0.61%	No change
Compute Utilization	35.13%	$\sim 35\%$	No change
Active Threads/Warp	18.09	$\sim 18$	No change

**Performance Summary Table**

## 3 Conclusion

### 3.1 Difficulties Encountered

#### 3.1.1 1. Balancing Divergence vs Computation

**Challenge:** Early exit optimizations in ray marching and shadow calculation cause severe warp divergence (47% efficiency loss), but they also reduce total work by 70–80%.

**Initial approach:** Tried to minimize divergence by removing early exits and using fixed iteration counts. This resulted in perfect warp convergence but was 2–3× slower overall.

**Solution:** After profiling with ncu, I realized the kernel is compute-bound, not divergence-bound. The reduction in total operations far outweighs the divergence penalty. I kept the early exit optimizations and accepted the divergence trade-off.

#### Lesson learned

For compute-intensive kernels, reducing total work is more important than avoiding divergence. Profile-guided optimization is essential.

#### 3.1.2 2. Register Pressure and Occupancy

**Challenge:** The kernel requires 66 registers per thread due to complex ray marching logic, limiting theoretical occupancy to 37.5%. This creates a conflict between algorithmic complexity and occupancy.

#### Attempts:

1. Tried reducing local variables → code became unreadable and error-prone
2. Tried recomputing values instead of storing → actually slower due to redundant calculations
3. Experimented with shared memory → no improvement (compute-bound, not memory-bound)

**Partial solution:** Instead of reducing register usage, I optimized block size to better utilize available occupancy. Switching from  $16 \times 16$  (256 threads) to  $8 \times 16$  (128 threads) improved occupancy from 31.2% to 43.8% and achieved 6.7% speedup.

#### Lesson learned

Sometimes it's better to work with hardware constraints rather than against them. Optimal block size can mitigate low occupancy issues.

#### 3.1.3 3. Understanding Performance Bottlenecks

**Challenge:** Initial profiling showed low memory bandwidth (0.61%) and low compute utilization (35.13%). It was unclear which to optimize first.

### Investigation process:

1. Used nvprof to confirm kernel-bound (not memory transfer)
2. Used ncu to identify compute-bound nature
3. Analyzed scheduler statistics to find warp stalling
4. Discovered 55.36% “No Eligible” cycles due to low occupancy
5. Found warp divergence causing 47% thread efficiency loss

**Solution:** Prioritized algorithm optimization (adaptive shadow step) which directly addresses the compute bottleneck, achieving 50% speedup. Then addressed occupancy through block size tuning for additional 7% gain.

### Lesson learned

Comprehensive profiling is crucial. Don’t assume bottlenecks – measure them. ncu’s detailed metrics were invaluable for understanding the true performance limiters.

### 3.1.4 4. Measuring and Validating Speedup

**Challenge:** Validating that optimizations actually improve performance while maintaining correctness. Some “optimizations” made things worse or broke accuracy.

#### Failed optimizations:

1. **Aggressive fast math flags (-use\_fast\_math):**
  - Speedup: 15%
  - Problem: Accuracy dropped to 92% (unacceptable)
  - Root cause: Shadow calculations became numerically unstable
2. **Shared memory caching:**
  - No speedup (0–1%)
  - Problem: Added complexity without benefit
  - Root cause: Kernel is compute-bound, not memory-bound
3. **32 × 32 block size:**
  - 48% SLOWER (!)
  - Problem: Only 1 block per SM due to register pressure
  - Root cause: Catastrophic occupancy collapse (12.5%)

**Solution:** Developed systematic testing methodology:

1. Profile baseline with nvprof and ncu
2. Apply optimization
3. Re-profile and compare metrics
4. Run accuracy tests on all test cases
5. Only keep optimizations that improve speed AND maintain > 97% accuracy

### Lesson learned

Always validate optimizations with comprehensive profiling and accuracy testing.  
Not all “optimizations” are actually improvements.

#### 3.1.5 5. Debugging CUDA Code

**Challenge:** CUDA debugging is notoriously difficult. Errors like out-of-bounds access or race conditions can cause silent corruption or cryptic error messages.

Specific issues encountered:

1. **Memory alignment bug:** Initial implementation used `unsigned char*` instead of `uchar4*`, causing unaligned writes and poor coalescing
2. **Constant memory overflow:** Tried to store too much data in constant memory ( $> 64\text{KB}$  limit), causing mysterious kernel failures
3. **Race conditions in early versions:** Multiple threads incorrectly shared state, causing non-deterministic results

Debugging techniques used:

- `cuda-memcheck` for memory errors
- `cuda-gdb` for breakpoints (limited usefulness)
- Extensive `printf` debugging (slow but reliable)
- Comparing outputs pixel-by-pixel to baseline
- Reducing problem size to simplify debugging

### Lesson learned

CUDA debugging is time-consuming. Write correct code first, then optimize. Use defensive programming (bounds checks, assertions).

## 3.2 Feedback and Suggestions

### 3.2.1 What Worked Well

1. **Clear Specification:**
  - Well-defined test cases with expected outputs
  - Clear accuracy threshold ( $\geq 97\%$ )
  - Comprehensive grading rubric
2. **Profiling Tools:**
  - `nvprof` and `ncu` provided invaluable insights
  - Learning to interpret profiling metrics was extremely educational
  - Real-world skill that transfers to industry
3. **Performance-Accuracy Trade-off:**
  - Requiring  $\geq 97\%$  accuracy forced thoughtful optimization

- Prevented “cheating” with aggressive approximations
- Taught the importance of maintaining correctness while optimizing

#### 4. Open-Ended Optimization:

- Freedom to choose optimization strategies
- Encouraged creativity and experimentation
- Multiple valid approaches to achieve speedup targets

##### 3.2.2 Suggestions for Improvement

**1. Starter Code Could Be More Optimized** **Current situation:** The baseline code has some obvious inefficiencies (e.g., using `unsigned char*` instead of `uchar4*`, no use of constant memory for read-only data).

**Suggestion:** Provide a more optimized baseline that already uses best practices for memory access patterns and data types. This would allow students to focus on algorithmic optimizations and advanced techniques rather than fixing basic inefficiencies.

**Alternative:** If the intention is to teach basic optimizations, consider adding explicit hints in the spec: “Consider using vectorized types for memory access” or “Which data could benefit from constant memory?”

**2. More Guidance on Profiling Tools** **Current situation:** The spec mentions using `nvprof` and `ncu` but doesn’t provide much guidance on how to interpret the results.

**Suggestion:**

- Provide a tutorial or reference document on interpreting key metrics (occupancy, IPC, cache hit rates, warp divergence)
- Include a worked example showing how profiling identified a bottleneck and led to an optimization
- Reference NVIDIA’s documentation explicitly

**Benefits:** Students would spend less time figuring out profiling and more time using insights to optimize.

**3. Intermediate Performance Targets** **Current situation:** Students need to achieve a certain speedup but don’t know if they’re on the right track until final testing.

**Suggestion:** Provide intermediate performance milestones:

- Basic optimization: 20–30% speedup
- Advanced optimization: 40–50% speedup
- Expert optimization: 60%+ speedup

With hints for each level:

- Basic: “Focus on memory access patterns and CUDA intrinsics”
- Advanced: “Consider algorithmic optimizations in hot loops”
- Expert: “Explore block size tuning and register optimizations”

**Benefits:** Helps students gauge progress and know when to move to the next optimization level.

**4. Optional Advanced Challenge** **Current situation:** After achieving the required speedup, there's limited motivation to optimize further.

**Suggestion:** Add optional “challenge” optimizations with bonus points:

- Achieve 60%+ speedup: +5% bonus
- Implement and analyze register pressure reduction: +3% bonus
- Implement persistent threads or advanced warp scheduling: +5% bonus

**Benefits:** Encourages exploration of advanced techniques without penalizing students who meet basic requirements.

**5. More Diverse Test Cases** **Current situation:** All test cases are similar (different Mandelbulb parameters and resolutions).

**Suggestion:** Include test cases with different characteristics:

- **Simple scene:** Mostly background (low ray marching complexity) – tests memory optimizations
- **Complex scene:** Dense fractal (high ray marching complexity) – tests compute optimizations
- **Edge case:** Extreme parameters that stress-test the implementation

**Benefits:** Ensures optimizations work across different scenarios, not just average cases.

**6. Clarification on Accuracy Measurement** **Current situation:** Accuracy is measured but the exact methodology isn't specified in detail.

**Questions students might have:**

- Is it pixel-by-pixel RMSE?
- What tolerance is used for floating-point comparisons?
- Are certain pixels weighted more (e.g., edges vs background)?

**Suggestion:** Add a note in the spec explaining:

- Exact accuracy metric used
- How it's calculated
- Which pixels are most sensitive to optimization-induced errors

**Benefits:** Helps students understand why certain optimizations affect accuracy and how to minimize degradation.

### 3.2.3 Overall Assessment

This assignment was **highly educational** and provided hands-on experience with:

- CUDA programming and GPU architecture
- Performance profiling and bottleneck identification
- Optimization trade-offs (speed vs accuracy, divergence vs computation)
- Real-world parallel programming challenges

The ray marching algorithm was a great choice because:

- Complex enough to benefit from GPU parallelization
- Multiple optimization opportunities at different levels
- Realistic workload (similar to actual graphics/rendering applications)
- Immediate visual feedback (you can see the output image)

**Difficulty level:** Appropriate for an advanced parallel programming course. Challenging but achievable with systematic profiling and optimization.

**Time investment:** ~ 20–30 hours including:

- Understanding the algorithm: 4–5 hours
- Initial CUDA implementation: 5–7 hours
- Profiling and optimization: 8–12 hours
- Testing and report writing: 3–5 hours

**Most valuable learning:** The profiling workflow (measure → identify bottleneck → optimize → validate) is directly applicable to industry and research. Learning to interpret ncu metrics was particularly valuable.

# Appendix: Profiling Commands Reference

## nvprof Commands

```
1 # Basic profiling
2 nvprof ./hw3 [args]
3
4 # GPU trace with timing
5 nvprof --print-gpu-trace ./hw3 [args]
6
7 # Summary statistics
8 nvprof --print-gpu-summary ./hw3 [args]
9
10 # Memory trace
11 nvprof --print-api-trace ./hw3 [args]
```

Listing 25: Basic nvprof Usage

## ncu Commands

```
1 # Full metrics (slow but comprehensive)
2 ncu --set full ./hw3 [args]
3
4 # Quick metrics
5 ncu --set brief ./hw3 [args]
6
7 # Specific metrics
8 ncu --metrics sm__throughput.avg.pct_of_peak_sustained_elapsed ./hw3
   [args]
9
10 # Export to file
11 ncu --export profile.ncu-rep ./hw3 [args]
```

Listing 26: Nsight Compute Usage

## Useful Metrics to Monitor

### Occupancy:

- sm\_\_warps\_active.avg.pct\_of\_peak\_sustained\_active

### Compute:

- sm\_\_throughput.avg.pct\_of\_peak\_sustained\_elapsed
- smsp\_\_inst\_executed.avg.per\_cycle\_active

### Memory:

- dram\_\_throughput.avg.pct\_of\_peak\_sustained\_elapsed
- l1tex\_\_t\_sectors\_pipe\_lsu\_mem\_global\_op\_ld.sum
- l1tex\_\_t\_sectors\_pipe\_lsu\_mem\_global\_op\_st.sum

**Divergence:**

- smsp\_\_thread\_inst\_executed\_per\_inst\_executed.ratio
- smsp\_\_average\_warps\_issue\_stalled\_no\_instruction.pct