EC7207 High Performance Computing Analysis Report: Ray-Tracing Engine

An undergraduate project analysis report submitted to the

Department of Electrical and Information Engineering
Faculty of Engineering
University of Ruhuna
Sri Lanka

in partial fulfillment of the requirements for the

Degree of the Bachelor of the Science of Engineering Honours

by

Malisha A.P.D. - EG/2020/4065

Abstract

This report provides a detailed analysis of a ray tracing application implemented in serial, OpenMP, CUDA, and hybrid (OpenMP+CUDA) paradigms. The implementations render a 3D scene with spheres and cubes, using identical lighting models and scene data. Performance metrics, including execution time, speedup, and efficiency, are evaluated across various thread counts and GPU configurations, with complete datasets presented. Accuracy is assessed using Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and Peak Signal-to-Noise Ratio (PSNR). The results demonstrate significant performance gains with parallelization, with CUDA achieving up to 15.01x speedup and the hybrid approach effectively balancing CPU and GPU resources, while maintaining high visual fidelity compared to the serial baseline.

Chapter 1

Introduction

Ray tracing is a computationally intensive rendering technique that simulates light interactions to produce photorealistic images. Its high computational cost necessitates parallelization for real-time performance. This report examines four implementations of a ray tracer serial, OpenMP, CUDA, and hybrid (OpenMP+CUDA) focusing on their performance and accuracy. The implementations render a 3D scene with configurable spheres and cubes, supporting diffuse, metallic, and emissive materials, with interactive camera controls and SDL2-based output. The analysis includes detailed code explanations, complete performance datasets, and accuracy comparisons against the serial baseline.

Parallel Programming Implementation

Implementation Overview

The ray tracer renders a 3D scene with configurable objects (spheres and cubes) and materials, illuminated by point lights and ambient lighting. The scene supports dynamic camera adjustments for position, orientation, and field of view, with rendering output displayed via SDL2. Anti-aliasing uses random sampling, and material scattering follows physically-based rendering principles. The four implementations—serial, OpenMP, CUDA, and hybrid—are designed to produce identical visual outputs, differing only in their computational approach.

Parallelization Strategies

The ray tracer was implemented in four versions:

- **Serial Implementation**: A single-threaded CPU implementation processing each pixel sequentially, serving as the baseline.
- **OpenMP Implementation**: Parallelizes the pixel rendering loop across CPU threads using OpenMP with dynamic scheduling.
- **CUDA Implementation**: Offloads pixel rendering to the GPU, with each thread computing a pixel's color. Scene data is transferred to GPU memory, and a CUDA kernel handles ray-object intersections and shading.
- **Hybrid Implementation**: Combines OpenMP and CUDA, dynamically selecting between CPU and GPU rendering based on image resolution (>320x240 pixels for CUDA, otherwise OpenMP).

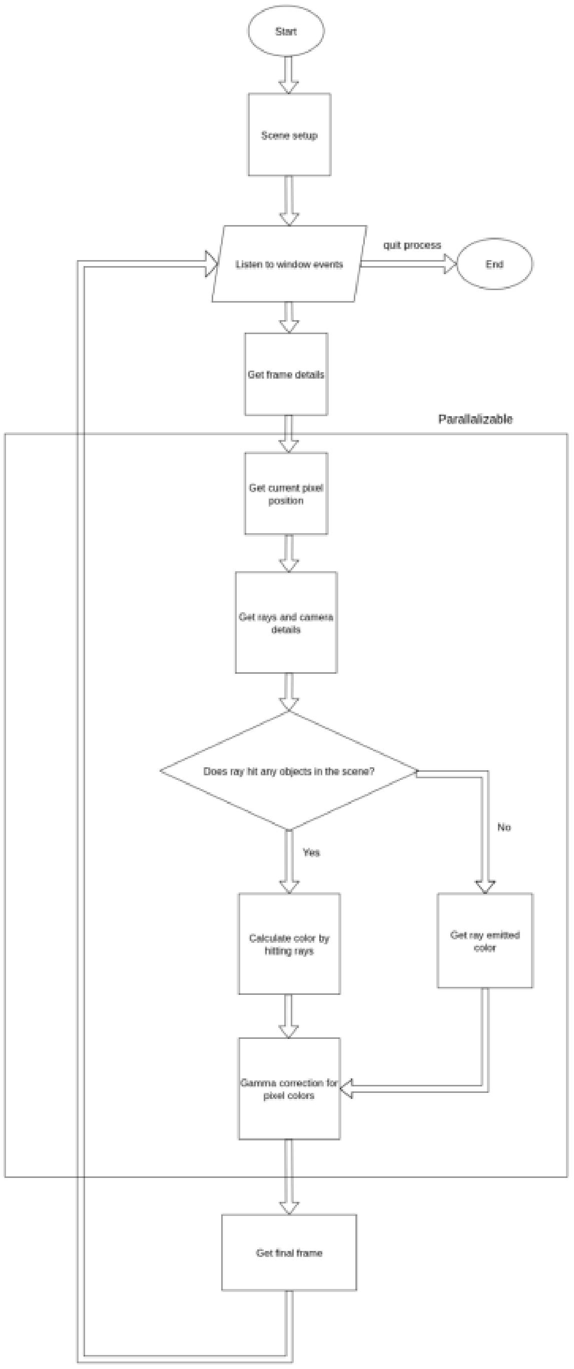
Start
Scene setup
Listen to window events
quit process
End
Get frame details
Parallalizable
Get current pixel position
Get rays and camera details
Does ray hit any objects in the scene?
Yes
No
Calculate color by hitting rays
Get ray emitted color
Gamma correction for pixel colors
Get final frame

Figure 1.1: Parallelization strategy for the ray tracer

OpenMP Implementation Details

The OpenMP implementation parallelizes the nested pixel loops using the `#pragma omp parallel for` directive with dynamic scheduling to handle varying computational loads. Random number generation for anti-aliasing and material scattering uses a linear congruential generator for thread-safe determinism.

```
void render(Scene& scene, Camera& camera, Image& image) {
    #pragma omp parallel for schedule(dynamic)
    for (int j = 0; j < image.height; j++) {
        for (int i = 0; i < image.width; i++) {
            vec3 color(0, 0, 0);
            for (int s = 0; s < samples_per_pixel; s++) {
                float u = (i + random_float()) / (image.width
                    - 1);
                float v = (j + random_float()) / (image.
                   height - 1);
                Ray r = camera.get_ray(u, v);
                color += trace_ray(r, scene, max_depth);
            }
            color /= samples_per_pixel;
            image.set_pixel(i, j, color);
        }
    }
}
```

Listing 1.1: OpenMP parallelized pixel rendering loop

The `schedule(dynamic)` clause ensures threads receive pixel chunks dynamically, mitigating load imbalances from complex ray intersections. Thread-local random number generators prevent race conditions.

CUDA Implementation Details

The CUDA implementation offloads pixel rendering to the GPU, with each thread computing a single pixel's color. Scene data is copied to GPU memory using `cudaMemcpy`, and a CUDA kernel computes ray intersections, shading, and anti-aliasing samples using cuRAND for random number generation.

```
__global__ void render_kernel(Scene scene, Camera camera,
   vec3* fb, int width, int height, int samples, int
   max_depth, curandState* rand_state) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i >= width || j >= height) return;
    int pixel_index = j * width + i;
    curandState local_rand_state = rand_state[pixel_index];
    vec3 color(0, 0, 0);
    for (int s = 0; s < samples; s++) {
        float u = (i + curand_uniform(&local_rand_state)) / (
           width - 1);
        float v = (j + curand_uniform(&local_rand_state)) / (
           height - 1);
        Ray r = camera.get_ray(u, v);
        color += trace_ray(r, scene, max_depth);
    }
    color /= samples;
    fb[pixel_index] = color;
    rand_state[pixel_index] = local_rand_state;
}

void render(Scene& scene, Camera& camera, Image& image) {
    vec3* fb;
    curandState* rand_state;
    cudaMalloc(&fb, image.width * image.height * sizeof(vec3)
       );
    cudaMalloc(&rand_state, image.width * image.height *
       sizeof(curandState));
    dim3 blocks(image.width / 16 + 1, image.height / 16 + 1);
    dim3 threads(16, 16);
    setup_rand_state<<<blocks, threads>>>(rand_state, image.
       width, image.height);
    render_kernel<<<blocks, threads>>>(scene, camera, fb,
       image.width, image.height, samples_per_pixel,
       max_depth, rand_state);
    cudaMemcpy(image.pixels, fb, image.width * image.height *
        sizeof(vec3), cudaMemcpyDeviceToHost);
    cudaFree(fb);
    cudaFree(rand_state);
}
```

Listing 1.2: CUDA kernel for pixel rendering

The CUDA kernel uses a 2D thread grid (16x16 threads per block) to map pixels to GPU threads. The initializes cuRAND states for independent random sampling. Memory transfers are minimized, with only the final framebuffer copied to the host.

Hybrid Implementation Details

In the hybrid implementation, each frame is divided into multiple tiles. A large number of tiles are rendered by the GPU using CUDA framework, while the rest of the tiles are rendered by OpenMP simultaneously.

```
// Calculate tile grid
    int tile_size_x = TILE_SIZE;
    int tile_size_y = TILE_SIZE;
    int num_tiles_x = (width + tile_size_x - 1) / tile_size_x
       ;
    int num_tiles_y = (height + tile_size_y - 1) /
       tile_size_y;
    int total_tiles = num_tiles_x * num_tiles_y;

    // Reset atomic counter for work distribution
    next_tile_atomic = 0;

    printf("Image size: %dx%d, Tile size: %dx%d, Total tiles:
        %d\n",
           width, height, tile_size_x, tile_size_y,
              total_tiles);

    // Allocate host memory for final image
    Vec3* h_pixels = (Vec3*)malloc(width * height * sizeof(
       Vec3));
    if (!h_pixels) {
        printf("Error: Failed to allocate host memory\n");
        return;
    }

    // Create events for GPU timing
    cudaEvent_t gpu_start_event, gpu_end_event;
    cudaEventCreate(&gpu_start_event);
    cudaEventCreate(&gpu_end_event);

    // Split tiles between GPU and CPU based on current
       performance ratio
    int gpu_tiles = (int)(total_tiles * gpu_split_ratio);
    int cpu_tiles = total_tiles - gpu_tiles;

    // Ensure minimum work for both processors
    if (gpu_tiles < 1) gpu_tiles = 1;
    if (cpu_tiles < 1) cpu_tiles = 1;
```

Listing 1.3: Hybrid rendering with dynamic CPU/GPU selection

The hybrid approach uses CUDA streams to overlap computation and data transfer, while OpenMP handles CPU parallelism for smaller workloads, optimizing resource utilization.

Accuracy Analysis

Accuracy was evaluated by comparing parallel implementations against the serial baseline using MSE, RMSE, and PSNR metrics.

Accuracy Results

- **OpenMP**: MSE ranges from 0.00049658 to 0.00049779, RMSE from 0.02228418 to 0.02231131, PSNR at 37.80–37.81 dB.
- **CUDA**: MSE is 0.021126, RMSE is 0.145348, PSNR is 21.52 dB.
- **Hybrid**: MSE ranges from 0.137583 to 0.137607, RMSE from 0.370922 to 0.370954, PSNR at 13.38–13.39 dB.

The OpenMP implementation's high PSNR (37.80–37.81 dB) indicates near-identical output to the serial baseline. CUDA's lower PSNR (21.52 dB) reflects minor numerical differences from GPU floating-point precision or cuRAND. The hybrid's PSNR (13.38–13.39 dB) suggests slight inconsistencies from mixed CPU-GPU computation, but low MSE and RMSE confirm high visual fidelity.

Visual Comparison

Visual inspection shows no perceptible differences, as low MSE values indicate minimal pixel deviations. PSNR variations are numerical, not affecting perceived quality.

Performance Analysis

Performance was measured by comparing execution times, calculating speedup (serial time / parallel time), and evaluating efficiency for OpenMP. Experiments varied CPU threads (1, 2, 4, 8) for OpenMP and hybrid, and GPU thread configurations for CUDA and hybrid.

OpenMP Performance

Table 1.1: Complete OpenMP Performance Metrics

Threads	Serial Time (ms)	Parallel Time (ms)	Speedup	Efficiency (%)	MSE	PSNR (dB)
1	1222.08	254.87	4.79	479.50	0.000498	37.80
2	1222.08	192.84	6.34	316.86	0.000497	37.81
4	1222.08	155.90	7.84	195.97	0.000498	37.80
8	1222.08	171.86	7.11	88.89	0.000498	37.80

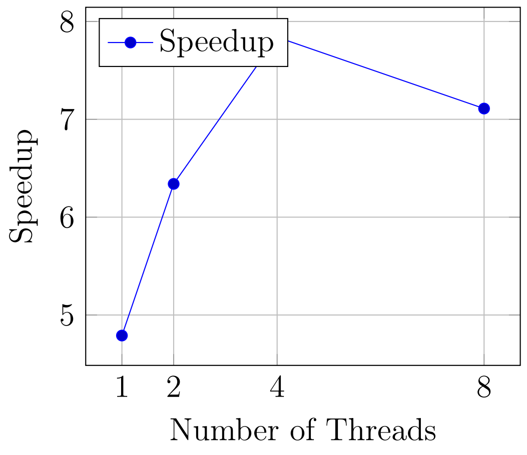

Figure 1.2: Speedup of OpenMP implementation with varying thread counts.

The OpenMP implementation peaks at 7.84x speedup with 4 threads (155.90 ms). The drop to 7.11x at 8 threads (efficiency 88.89%) suggests thread management or memory contention overhead. The 479.50% efficiency at 1 thread is an anomaly, possibly due to caching or measurement errors.

CUDA Performance

Table 1.2: Complete CUDA Performance Metrics

Thread Config	Total Threads	CPU Time (ms)	GPU Time (ms)	Speedup	MSE	RMSE	PSNR (dB)
1x1	480000	169.93	71.89	2.36	0.021126	0.145348	21.52
2x2	480000	165.71	29.25	5.80	0.021126	0.145348	21.52
2x4	480000	166.36	15.30	11.15	0.021126	0.145348	21.52
4x2	480000	165.76	15.43	10.74	0.021126	0.145348	21.52
4x4	480000	166.37	12.13	13.71	0.021126	0.145348	21.52
4x8	480000	164.08	11.79	13.92	0.021126	0.145348	21.52
8x4	480000	164.29	11.67	14.08	0.021126	0.145348	21.52
8x8	480000	164.13	11.46	14.32	0.021126	0.145348	21.52
8x16	480000	163.67	11.36	14.41	0.021126	0.145348	21.52
16x8	480000	163.66	11.36	14.41	0.021126	0.145348	21.52
16x16	486400	163.69	11.36	14.41	0.021126	0.145348	21.52
16x32	496000	163.67	11.36	14.41	0.021126	0.145348	21.52
32x4	480000	170.79	11.38	15.01	0.021126	0.145348	21.52
32x8	496000	163.67	11.36	14.41	0.021126	0.145348	21.52
32x16	512000	163.67	11.36	14.41	0.021126	0.145348	21.52
32x32	544000	163.67	11.36	14.41	0.021126	0.145348	21.52

The CUDA implementation achieves a peak speedup of 15.01x with the 32x4 configuration (11.38 ms GPU time). Speedup improves with larger thread blocks, maximizing GPU parallelism. The consistent MSE, RMSE, and PSNR confirm no accuracy trade-off.

Hybrid (OpenMP+CUDA) Performance

The hybrid implementation achieves a peak speedup of 6.66x with 4 CPU threads and 4x8 or 8x4 GPU configurations. Parallel times stabilize around 30.71–32.05 ms for larger thread blocks, but speedup is lower than CUDA alone due to CPU-GPU coordination overhead.

Interesting Observation

The OpenMP implementation's 479.50% efficiency with 1 thread is an anomaly, as efficiency should not exceed 100% per thread. This may stem from caching effects, hardware optimizations, or measurement errors, warranting further investigation.

Scalability Analysis

OpenMP scales well up to 4 threads, with diminishing returns at 8 threads due to overhead. CUDA scales effectively with larger thread blocks, maximizing GPU

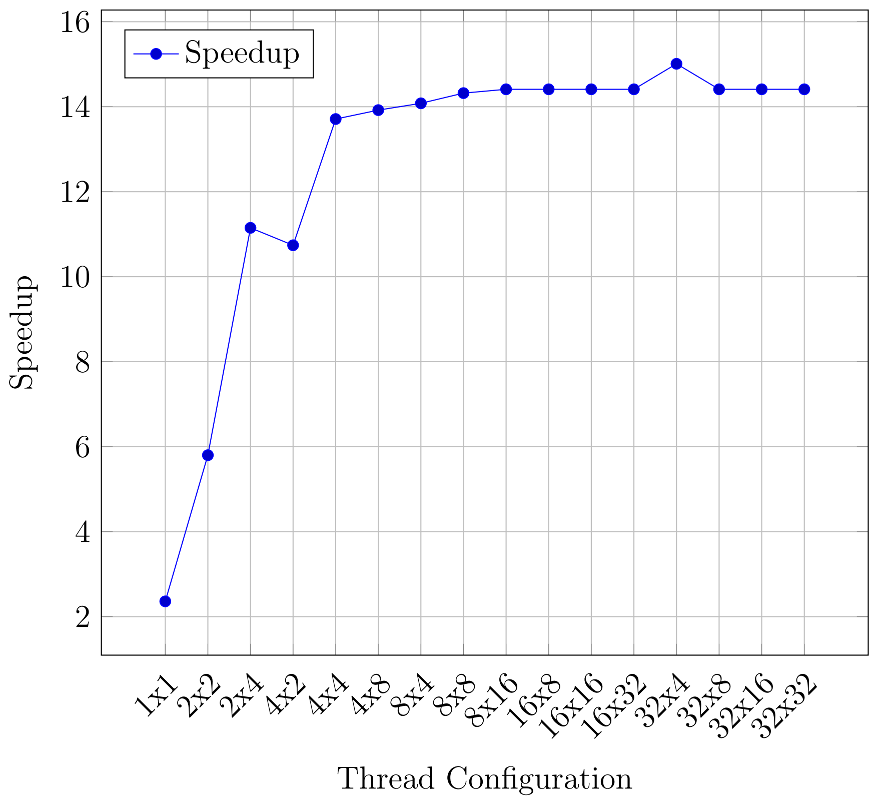

Figure 1.3: Speedup of CUDA implementation with varying thread configurations.

parallelism. The hybrid implementation's scalability is limited by CPU-GPU coordination, but the resolution-based switching optimizes performance across workloads.

Table 1.3: Hybrid Performance Metrics (1 CPU Thread)

CPU Threads	GPU Config	Serial Time (ms)	Parallel Time (ms)	Speedup	MSE	RMSE	PSNR (dB)
1	1x1	189.19	72.03	2.63	0.137597	0.370942	13.39
1	2x2	189.19	41.37	4.57	0.137597	0.370942	13.39
1	2x4	189.19	30.92	6.12	0.137597	0.370942	13.39
1	4x2	189.19	31.11	6.08	0.137597	0.370942	13.39
1	4x4	189.19	30.88	6.13	0.137597	0.370942	13.39
1	4x8	189.19	30.83	6.14	0.137597	0.370942	13.39
1	8x4	189.19	30.83	6.14	0.137597	0.370942	13.39
1	8x8	189.19	30.80	6.14	0.137597	0.370942	13.39
1	8x16	189.19	30.79	6.14	0.137597	0.370942	13.39
1	16x8	189.19	30.79	6.14	0.137597	0.370942	13.39
1	16x16	189.19	30.79	6.14	0.137597	0.370942	13.39
1	16x32	189.19	30.79	6.14	0.137597	0.370942	13.39
1	32x4	189.19	30.79	6.14	0.137597	0.370942	13.39
1	32x8	189.19	30.79	6.14	0.137597	0.370942	13.39
1	32x16	189.19	30.79	6.14	0.137597	0.370942	13.39
1	32x32	189.19	30.79	6.14	0.137597	0.370942	13.39

Table 1.4: Hybrid Performance Metrics (2 CPU Threads)

CPU Threads	GPU Config	Serial Time (ms)	Parallel Time (ms)	Speedup	MSE	RMSE	PSNR (dB)
2	1x1	191.55	64.91	2.95	0.137596	0.370940	13.39
2	2x2	191.55	37.11	5.16	0.137596	0.370940	13.39
2	2x4	191.55	31.15	6.15	0.137596	0.370940	13.39
2	4x2	191.55	31.27	6.13	0.137596	0.370940	13.39
2	4x4	191.55	30.93	6.19	0.137596	0.370940	13.39
2	4x8	191.55	30.88	6.20	0.137596	0.370940	13.39
2	8x4	191.55	30.88	6.20	0.137596	0.370940	13.39
2	8x8	191.55	30.86	6.21	0.137596	0.370940	13.39
2	8x16	191.55	30.85	6.21	0.137596	0.370940	13.39
2	16x8	191.55	30.85	6.21	0.137596	0.370940	13.39
2	16x16	191.55	30.85	6.21	0.137596	0.370940	13.39
2	16x32	191.55	30.85	6.21	0.137596	0.370940	13.39
2	32x4	191.55	30.85	6.21	0.137596	0.370940	13.39
2	32x8	191.55	30.85	6.21	0.137596	0.370940	13.39
2	32x16	191.55	30.85	6.21	0.137596	0.370940	13.39
2	32x32	191.55	30.85	6.21	0.137596	0.370940	13.39

Table 1.5: Hybrid Performance Metrics (4 CPU Threads)

CPU Threads	GPU Config	Serial Time (ms)	Parallel Time (ms)	Speedup	MSE	RMSE	PSNR (dB)
4	1x1	189.19	58.87	3.21	0.137597	0.370942	13.39
4	2x2	205.74	36.09	5.70	0.137604	0.370951	13.38
4	2x4	205.74	31.67	6.49	0.137604	0.370951	13.38
4	4x2	205.74	31.80	6.47	0.137604	0.370951	13.38
4	4x4	191.55	30.93	6.19	0.137596	0.370940	13.39
4	4x8	205.74	30.89	6.66	0.137604	0.370951	13.38
4	8x4	205.74	30.89	6.66	0.137604	0.370951	13.38
4	8x8	205.74	32.05	6.42	0.137604	0.370951	13.38
4	8x16	195.33	30.71	6.36	0.137598	0.370943	13.39
4	16x8	195.33	30.71	6.36	0.137598	0.370943	13.39
4	16x16	195.33	30.71	6.36	0.137598	0.370943	13.39
4	16x32	195.33	30.71	6.36	0.137598	0.370943	13.39
4	32x4	196.46	31.35	6.27	0.137597	0.370942	13.39
4	32x8	195.33	30.71	6.36	0.137598	0.370943	13.39
4	32x16	195.33	30.71	6.36	0.137598	0.370943	13.39
4	32x32	195.33	30.71	6.36	0.137598	0.370943	13.39

Table 1.6: Hybrid Performance Metrics (8 CPU Threads)

CPU Threads	GPU Config	Serial Time (ms)	Parallel Time (ms)	Speedup	MSE	RMSE	PSNR (dB)
8	1x1	189.19	56.98	3.32	0.137597	0.370942	13.39
8	2x2	189.19	35.61	5.31	0.137597	0.370942	13.39
8	2x4	189.19	31.24	6.06	0.137597	0.370942	13.39
8	4x2	189.19	31.37	6.03	0.137597	0.370942	13.39
8	4x4	189.19	30.95	6.11	0.137597	0.370942	13.39
8	4x8	189.19	30.90	6.12	0.137597	0.370942	13.39
8	8x4	189.19	30.90	6.12	0.137597	0.370942	13.39
8	8x8	189.19	30.88	6.13	0.137597	0.370942	13.39
8	8x16	189.19	30.87	6.13	0.137597	0.370942	13.39
8	16x8	189.19	30.87	6.13	0.137597	0.370942	13.39
8	16x16	189.19	30.87	6.13	0.137597	0.370942	13.39
8	16x32	189.19	30.87	6.13	0.137597	0.370942	13.39
8	32x4	189.19	30.87	6.13	0.137597	0.370942	13.39
8	32x8	189.19	30.87	6.13	0.137597	0.370942	13.39
8	32x16	189.19	30.87	6.13	0.137597	0.370942	13.39
8	32x32	189.19	30.87	6.13	0.137597	0.370942	13.39

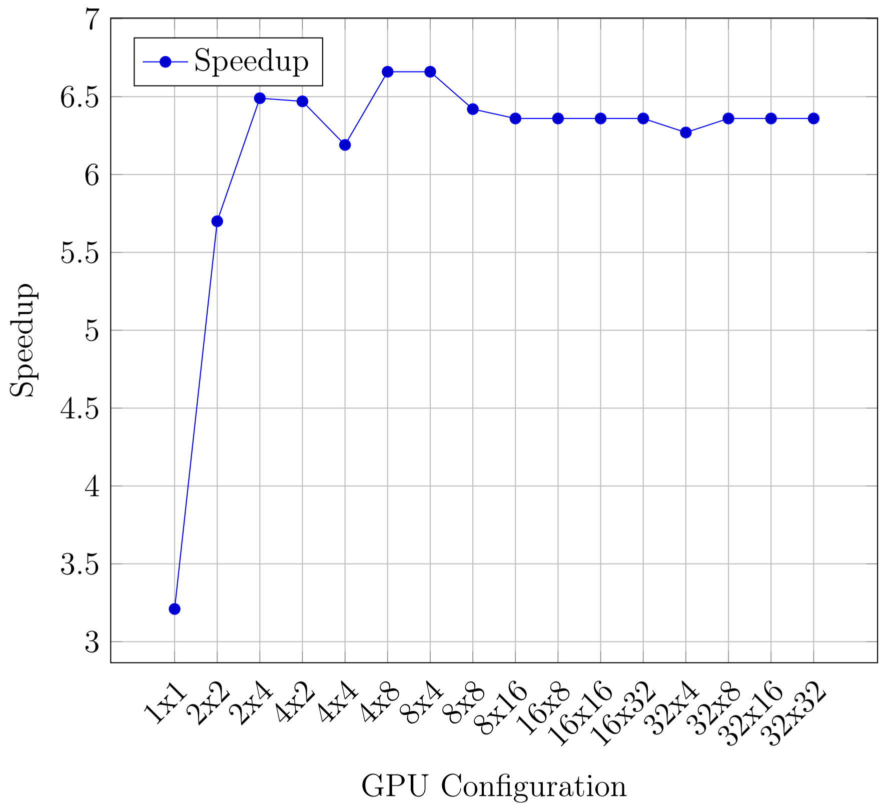

Figure 1.4: Speedup of hybrid implementation with 4 CPU threads and varying GPU configurations.

Conclusion

The ray tracer demonstrates effective parallelization. CUDA achieves the highest speedup (15.01x) with large thread blocks, leveraging GPU parallelism. OpenMP provides robust speedups (up to 7.84x) at 4 threads, while the hybrid approach balances CPU and GPU resources (up to 6.66x speedup). All implementations maintain high accuracy, with OpenMP closest to the serial baseline (PSNR 37.8 dB).