Project 4, problem 1

Table of Contents

Installation

Requirements

- C++ compiler (g++, clang++, msvc++)
- OpenMp
- CUDA
- XMake
- Git

Installation

```
git clone git@github.com:GlassAlo/CAU_Multicore.git
cd CAU_Multicore/proj4
```

Compile with xmake (easier)

```
xmake f -m release && xmake -y
```

- **-m release** is used to compile the project in release mode, which is faster than debug mode.
- **xmake -y** is used to compile the project. The **-y** flag is used to skip the confirmation prompt.
- **xmake** will create a **bin** folder with the executables inside.

Compile manually

```
cd problem1
g++ -O3 -fopenmp openmp_ray.cpp -o openmp_ray
nvcc -O3 -arch=sm_75 cuda_ray.cu -o cuda_ray
```

Usage

```
./bin/cuda_ray <filename>
default <filename> is <result.ppm>, optional argument
./bin/openmp_ray <number of threads> <filename>
<number of threads>, required
default <filename> is <result.ppm>, optional argument
```

Test

🖥️ Hardware Specifications

- **OS**: Garuda Linux Broadwing x86_64
- **Kernel**: 6.13.8-zen1-1-zen
- **CPU**: AMD Ryzen 9 5900HS with Radeon Graphics (16) @ 4.680GHz
  - Cores 8
    - Uniform core design
  - Threads 16
  - Base clock 3.0GHz
  - Max boost clock up to 4.6GHz
  - L3 cache 16MB
  - Memory PCIe 3.0
  - Supports Simultaneous Multithreading (SMT), with each cores supporting two threads
- **Integrated GPU**: AMD ATI Radeon Vega Series / Radeon Vega Mobile Series
- **Discrete GPU**: NVIDIA GeForce RTX 3080 Mobile / Max-Q (8GB/16GB)
- **RAM**: 32GB
- **Disk**: 1TB SSD

- **Shell**: zsh

- Using Arch Linux comes with a cost, the CPU / GPU drivers might not be very efficient, stable or up to date.

OpenMP version

🧪 Source Code

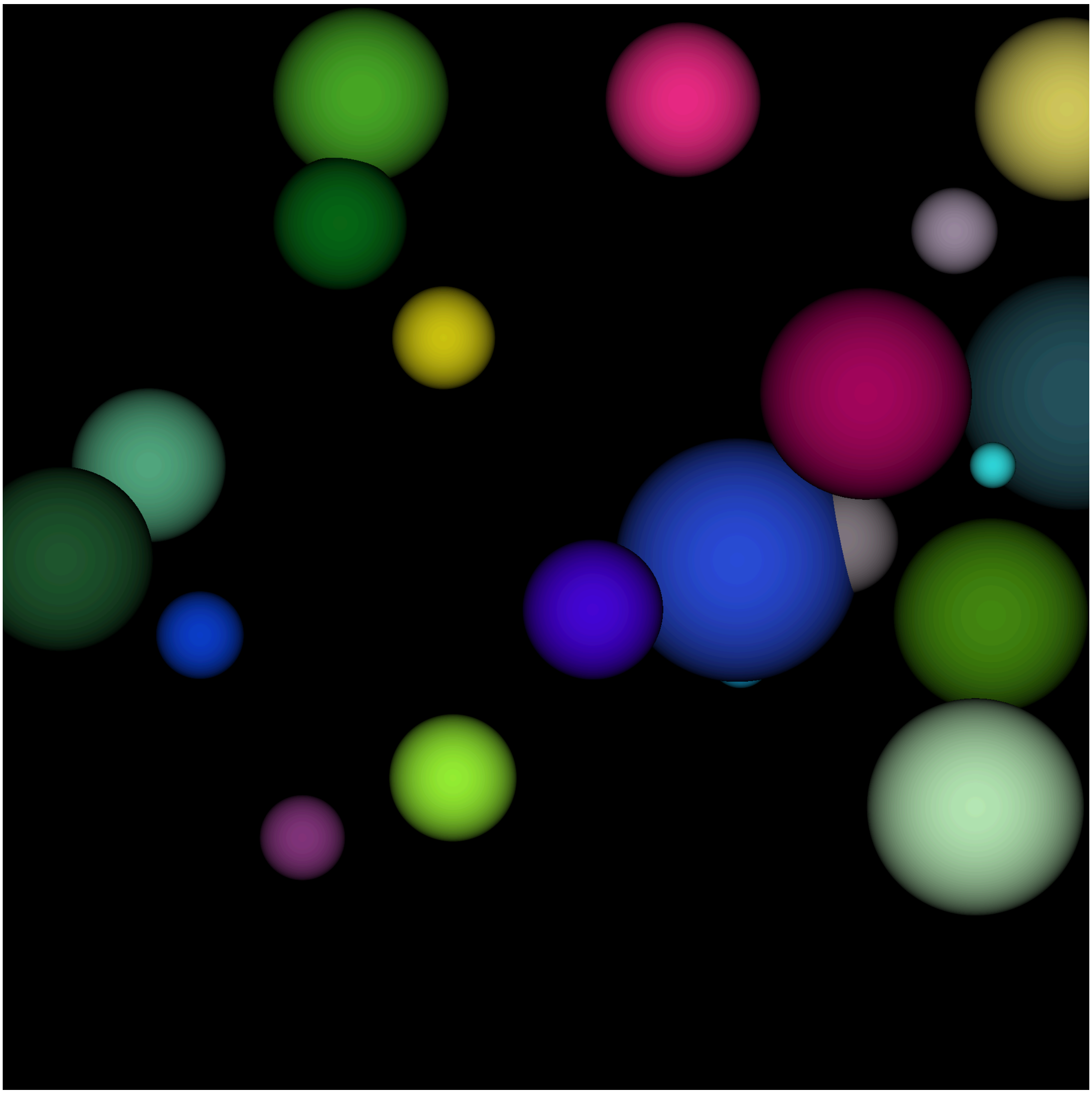
```
#include <math.h>
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <string>
#include <time.h>

#define CUDA    0
#define OPENMP  1
#define SPHERES 20

#define rnd(x) (x * rand() / RAND_MAX)
#define INF    2e10f
#define DIM    2048

struct Sphere
{
       float r, b, g;
       float radius;
       float x, y, z;
       float hit(float ox, float oy, float *n)
       {
           float dx = ox - x;
           float dy = oy - y;
           if (dx * dx + dy * dy < radius * radius) {
               float dz = sqrtf(radius * radius - dx * dx - dy * dy);
               *n = dz / sqrtf(radius * radius);
               return dz + z;
           }
           return -INF;
       }
};

void kernel(int x, int y, Sphere *s, unsigned char *ptr)
{
   int offset = x + y * DIM;
```

```
    float ox = (x - DIM / 2);
    float oy = (y - DIM / 2);

    // printf("x:%d, y:%d, ox:%f, oy:%f\n",x,y,ox,oy);

    float r = 0, g = 0, b = 0;
    float maxz = -INF;
    // using schedule(dynamic) to allow dynamic load balancing among threads
    // This is particularly useful when the workload is unevenly distributed
across iterations.
    // The parallel for loop iterates over the spheres, checking for hits
with the ray.
#pragma omp parallel for schedule(dynamic)
    for (int i = 0; i < SPHERES; i++) {
        float n;
        float t = s[i].hit(ox, oy, &n);
        if (t > maxz) {
            float fscale = n;
            r = s[i].r * fscale;
            g = s[i].g * fscale;
            b = s[i].b * fscale;
            maxz = t;
        }
    }

    ptr[offset * 4 + 0] = (int) (r * 255);
    ptr[offset * 4 + 1] = (int) (g * 255);
    ptr[offset * 4 + 2] = (int) (b * 255);
    ptr[offset * 4 + 3] = 255;
}

void ppm_write(unsigned char *bitmap, int xdim, int ydim, FILE *fp)
{
    int i, x, y;
    fprintf(fp, "P3\n");
    fprintf(fp, "%d %d\n", xdim, ydim);
    fprintf(fp, "255\n");
    for (y = 0; y < ydim; y++) {
        for (x = 0; x < xdim; x++) {
            i = x + y * xdim;
            fprintf(fp, "%d %d %d ", bitmap[4 * i], bitmap[4 * i + 1],
bitmap[4 * i + 2]);
        }
        fprintf(fp, "\n");
    }
}

int main(int argc, char *argv[])
```

```
{
    int no_threads;
    int option;
    int x, y;
    unsigned char *bitmap;

    srand(time(NULL));
    std::string filename = "result.ppm";
    if (argc < 2 || argc > 3) {
        printf("Usage: %s <no_threads> [<filename>]\n", argv[0]);
        return 1;
    }

    no_threads = atoi(argv[1]);
    omp_set_num_threads(no_threads);
    if (argc == 3) {
        filename = argv[2];
    }
    FILE *fp = fopen(filename.c_str(), "w");

    Sphere *temp_s = (Sphere *) malloc(sizeof(Sphere) * SPHERES);
    for (int i = 0; i < SPHERES; i++) {
        temp_s[i].r = rnd(1.0f);
        temp_s[i].g = rnd(1.0f);
        temp_s[i].b = rnd(1.0f);
        temp_s[i].x = rnd(2000.0f) - 1000;
        temp_s[i].y = rnd(2000.0f) - 1000;
        temp_s[i].z = rnd(2000.0f) - 1000;
        temp_s[i].radius = rnd(200.0f) + 40;
    }

    bitmap = (unsigned char *) malloc(sizeof(unsigned char) * DIM * DIM * 4);
    double start_time = omp_get_wtime();

    // The collapse(2) clause allows the two nested loops to be treated as a
single loop,
    // enabling better load balancing across threads.
    // The schedule(dynamic) clause allows the iterations to be distributed
dynamically among the threads,
    // which can help improve performance for workloads with varying
execution times.
#pragma omp parallel for collapse(2) schedule(dynamic)
    for (x = 0; x < DIM; x++)
        for (y = 0; y < DIM; y++)
            kernel(x, y, temp_s, bitmap);
    double end_time = omp_get_wtime();
    printf("OpenMP Execution Time: %f sec\n", end_time - start_time);
    ppm_write(bitmap, DIM, DIM, fp);
```

```
    printf("[%s] was generated.", filename.c_str());

    fclose(fp);
    free(bitmap);
    free(temp_s);

    return 0;
}
```

🟢 Output

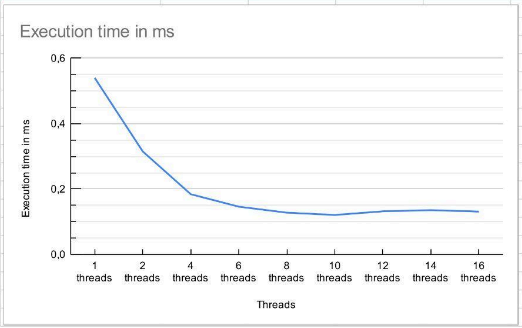


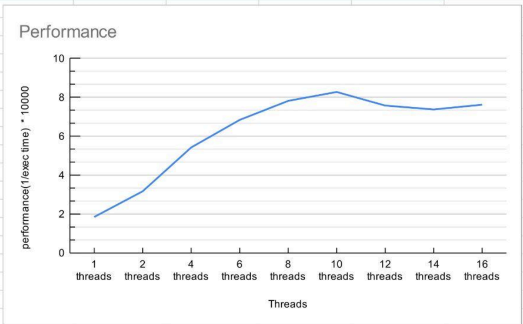
📊 Performance Chart

```
~/Documents/CAU_Multi/proj4 git:(main)±8 (0.908s)
./bin/openmp_ray 1
OpenMP Execution Time: 0.540127 sec
[result.ppm] was generated.%

~/Documents/CAU_Multi/proj4 git:(main)±8 (0.702s)
./bin/openmp_ray 2
OpenMP Execution Time: 0.315157 sec
[result.ppm] was generated.%

~/Documents/CAU_Multi/proj4 git:(main)±8 (0.581s)
./bin/openmp_ray 4
OpenMP Execution Time: 0.184273 sec
[result.ppm] was generated.%

~/Documents/CAU_Multi/proj4 git:(main)±8 (0.552s)
./bin/openmp_ray 6
OpenMP Execution Time: 0.146117 sec
[result.ppm] was generated.%

~/Documents/CAU_Multi/proj4 git:(main)±8 (0.574s)
./bin/openmp_ray 8
OpenMP Execution Time: 0.127839 sec
[result.ppm] was generated.%

~/Documents/CAU_Multi/proj4 git:(main)±8 (0.584s)
./bin/openmp_ray 10
OpenMP Execution Time: 0.120762 sec
[result.ppm] was generated.%

~/Documents/CAU_Multi/proj4 git:(main)±8 (0.622s)
./bin/openmp_ray 12
OpenMP Execution Time: 0.131898 sec
[result.ppm] was generated.%

~/Documents/CAU_Multi/proj4 git:(main)±8 (0.664s)
./bin/openmp_ray 14
OpenMP Execution Time: 0.135467 sec
[result.ppm] was generated.%

~/Documents/CAU_Multi/proj4 git:(main)±8 (0.645s)
./bin/openmp_ray 16
OpenMP Execution Time: 0.131134 sec
[result.ppm] was generated.%
```

exec time in sec	1 threads	2 threads	4 threads	6 threads	8 threads	10 threads	12 threads	14 threads	16 threads
OpenMP	0,540127	0,315157	0,184273	0,146117	0,127839	0,120762	0,131898	0,135467	0,131134

performance (1/exec time)	1 threads	2 threads	4 threads	6 threads	8 threads	10 threads	12 threads	14 threads	16 threads
OpenMP	1,851416426	3,1730217	5,426730992	6,843830629	7,822339036	8,280750567	7,581616097	7,3818716	7,625787363





🧠 Interpretation

- Performance improves significantly as the number of threads increases, particularly up to 10 threads. Execution time drops from **0.54 s (1 thread)** to **0.12 s (10 threads)**, with performance (1/time) peaking at this point. This aligns well with the **8-core / 16-thread architecture** of the CPU AMD Ryzen 9 5900HS used for this test.
- Beyond 10 threads, performance **plateaus or slightly degrades**, likely due to **overhead** or **thread synchronization costs**. This suggests that while OpenMP scales well initially, optimal parallel performance is reached at **10 threads**. Using more threads **is basically useless**. Further gains would require deeper algorithmic optimizations or offloading to the GPU (CUDA).

CUDA version

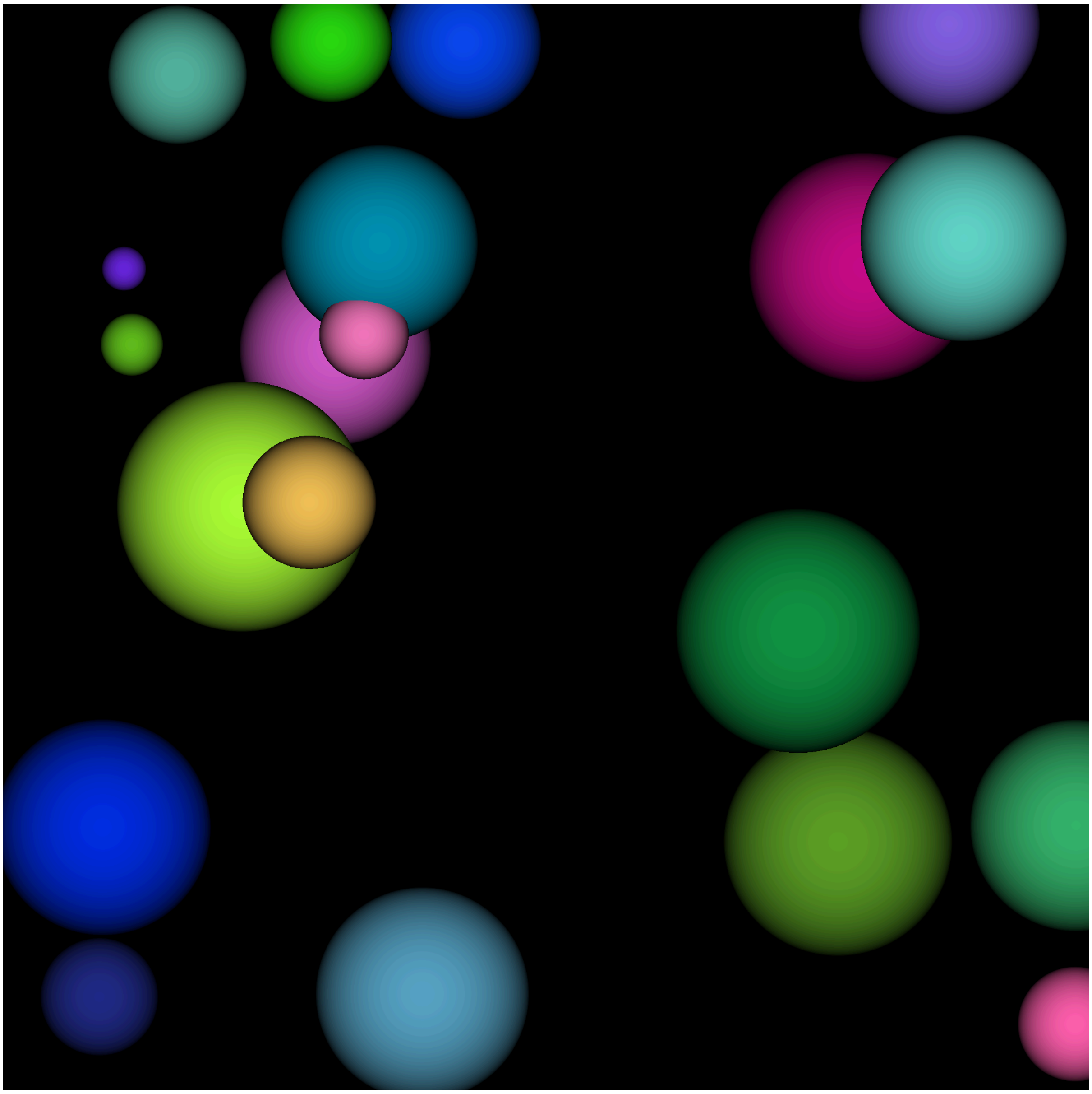
🧪 Source Code

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <cuda_runtime.h>

#define SPHERES 20
#define INF     2e10f
#define DIM     2048
#define rnd(x)  (x * rand() / RAND_MAX)
```

```
struct Sphere
{
       float r, g, b;
       float radius;
       float x, y, z;

       __device__ float hit(float ox, float oy, float *n) const
       {
           float dx = ox - x;
           float dy = oy - y;
           float rr = radius * radius;
           float d2 = dx * dx + dy * dy;
           if (d2 < rr) {
               float dz = sqrtf(rr - d2);
               *n = dz / radius;
               return dz + z;
           }
           return -INF;
       }
};

// Move spheres to constant memory for faster cached access
__constant__ Sphere d_spheres[SPHERES];

__global__ void render_kernel(unsigned char *ptr)
{
   int x = blockIdx.x * blockDim.x + threadIdx.x;
   int y = blockIdx.y * blockDim.y + threadIdx.y;
   if (x >= DIM || y >= DIM)
       return;

   int offset = x + y * DIM;
   float ox = x - DIM * 0.5f;
   float oy = y - DIM * 0.5f;

   float r = 0, g = 0, b = 0;
   float maxz = -INF;

   for (int i = 0; i < SPHERES; i++) {
       float n;
       float t = d_spheres[i].hit(ox, oy, &n);
       if (t > maxz) {
           maxz = t;
           r = d_spheres[i].r * n;
           g = d_spheres[i].g * n;
           b = d_spheres[i].b * n;
       }
   }
```

```
    // Use __saturatef to clamp color values and avoid branching
    ptr[offset * 4 + 0] = (unsigned char) (__saturatef(r) * 255.0f);
    ptr[offset * 4 + 1] = (unsigned char) (__saturatef(g) * 255.0f);
    ptr[offset * 4 + 2] = (unsigned char) (__saturatef(b) * 255.0f);
    ptr[offset * 4 + 3] = 255;
}

void ppm_write(const unsigned char *bitmap, int xdim, int ydim, FILE *fp)
{
    fprintf(fp, "P3\n%d %d\n255\n", xdim, ydim);
    for (int y = 0; y < ydim; y++) {
        for (int x = 0; x < xdim; x++) {
            int i = x + y * xdim;
            fprintf(fp, "%d %d %d ", bitmap[4 * i], bitmap[4 * i + 1],
bitmap[4 * i + 2]);
        }
        fprintf(fp, "\n");
    }
}

int main(int argc, char *argv[])
{
    char *filename = "result.ppm";
    if (argc > 1) {
        filename = argv[1];
    }
    srand(time(NULL));

    Sphere spheres_h[SPHERES];
    for (int i = 0; i < SPHERES; i++) {
        spheres_h[i].r = rnd(1.0f);
        spheres_h[i].g = rnd(1.0f);
        spheres_h[i].b = rnd(1.0f);
        spheres_h[i].x = rnd(2000.0f) - 1000;
        spheres_h[i].y = rnd(2000.0f) - 1000;
        spheres_h[i].z = rnd(2000.0f) - 1000;
        spheres_h[i].radius = rnd(200.0f) + 40;
    }

    // Copy host spheres to device constant memory
    cudaMemcpyToSymbol(d_spheres, spheres_h, sizeof(Sphere) * SPHERES);

    unsigned char *bitmap_h = nullptr;
    unsigned char *bitmap_d = nullptr;
    // Allocate device memory for image buffer
    cudaMalloc((void **) &bitmap_d, DIM * DIM * 4);
    // Allocate pinned (page-locked) host memory for image buffer
```

🟢 Output



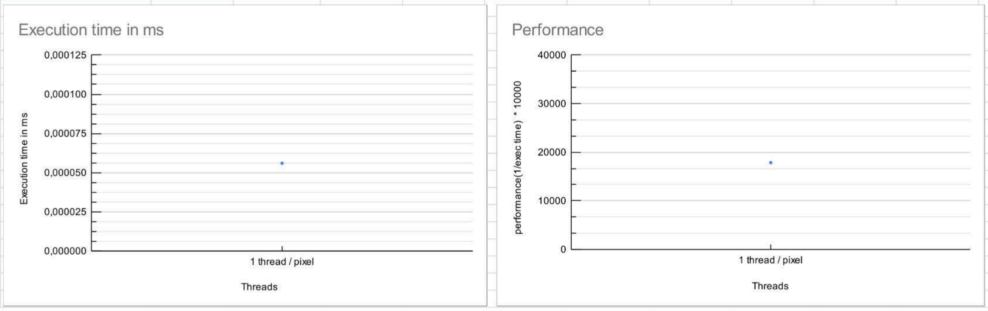
```
~/Documents/CAU_Multi/proj4 git:(main)±7 (0.637s)
./bin/cuda_ray

CUDA ray tracing: 0.000056 sec
[result.ppm] was generated.
```

📊 Performance Chart

```
~/Documents/CAU_Multi/proj4 git:(main)±8 (2.174s)
./bin/cuda_ray

CUDA ray tracing: 0.000056 sec
[result.ppm] was generated.
```

exec time in sec	1 thread / pixel
CUDA	0,000056

performance (1/exec time)	1 thread / pixel
CUDA	17857,14286



🧠 Interpretation

- The execution time per pixel using **1 CUDA thread per pixel** is **0.000056 seconds (56 microseconds)**. This results in a performance of **17,857 (1/sec) far higher** than the OpenMP version.
- This performance reflects how well the GPU handles massively parallel workloads. By assigning **one thread per pixel**, CUDA can compute at the same time thousands of threads on the **NVIDIA GeForce RTX 3080 Mobile**..

🏁 Conclusion

Metric	OpenMP (Best: 10 threads)	CUDA (1 thread/pixel)
Execution time (s)	0.120762	0.000056
Performance (1/time)	8.28	17,857.14
Speedup over 1-thread OpenMP	4.47×	~9,645×
Parallelism granularity	Thread per block	Thread per pixel (massive)
Hardware used	CPU (8-core/16-thread)	GPU (RTX 3080 Mobile)

- **CUDA dominates OpenMP** for this type of task. The GPU version is **~9,600 times faster** than the single-threaded CPU version and over **140 times faster** than the fastest OpenMP (10 threads) configuration.

- GPUs are ideal for pixel-based parallel tasks like raytracing, whereas CPUs are better suited for control-heavy or sequential logic.

- While OpenMP is useful for quick multithreaded speedup on CPU, **CUDA is far more effective** for performance-critical applications if a capable GPU is available.