Multicore Computing Assignement 4

Joshua BRIONNE - 50241647

🧰 Compilation & Execution

Compilation has been performed on **Ubuntu 22.04.3 LTS (Linux)**.

---

1. Environment

Property	Value
OS Name	Linux
OS Version	6.8.0-52-generic
Architecture	amd64
Available processors (cores)	8
Max memory (MB)	3936
CUDA Environment	Google Colab

---

2. Compilation

2.1 OpenMP

```
g++ -fopenmp openmp_ray.cpp

./a.out <number_of_threads>
```

2.2 CUDA

```
nvcc -arch=sm_75 cuda_ray.cu

./a.out
```

⏱️ Execution Time (in seconds)

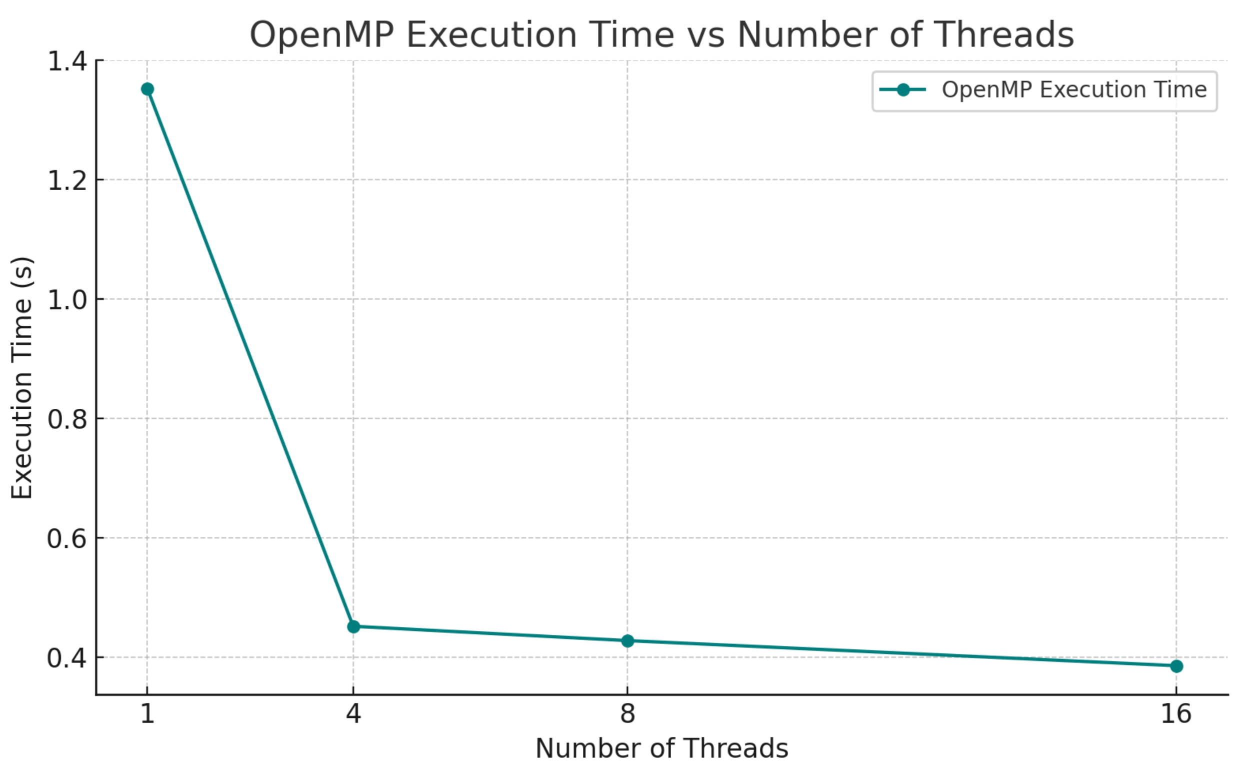
OpenMP

Threads	Time (s)
1	1.352
4	0.452
8	0.428
16	0.386

CUDA

Type	Time (s)
CUDA	0.135

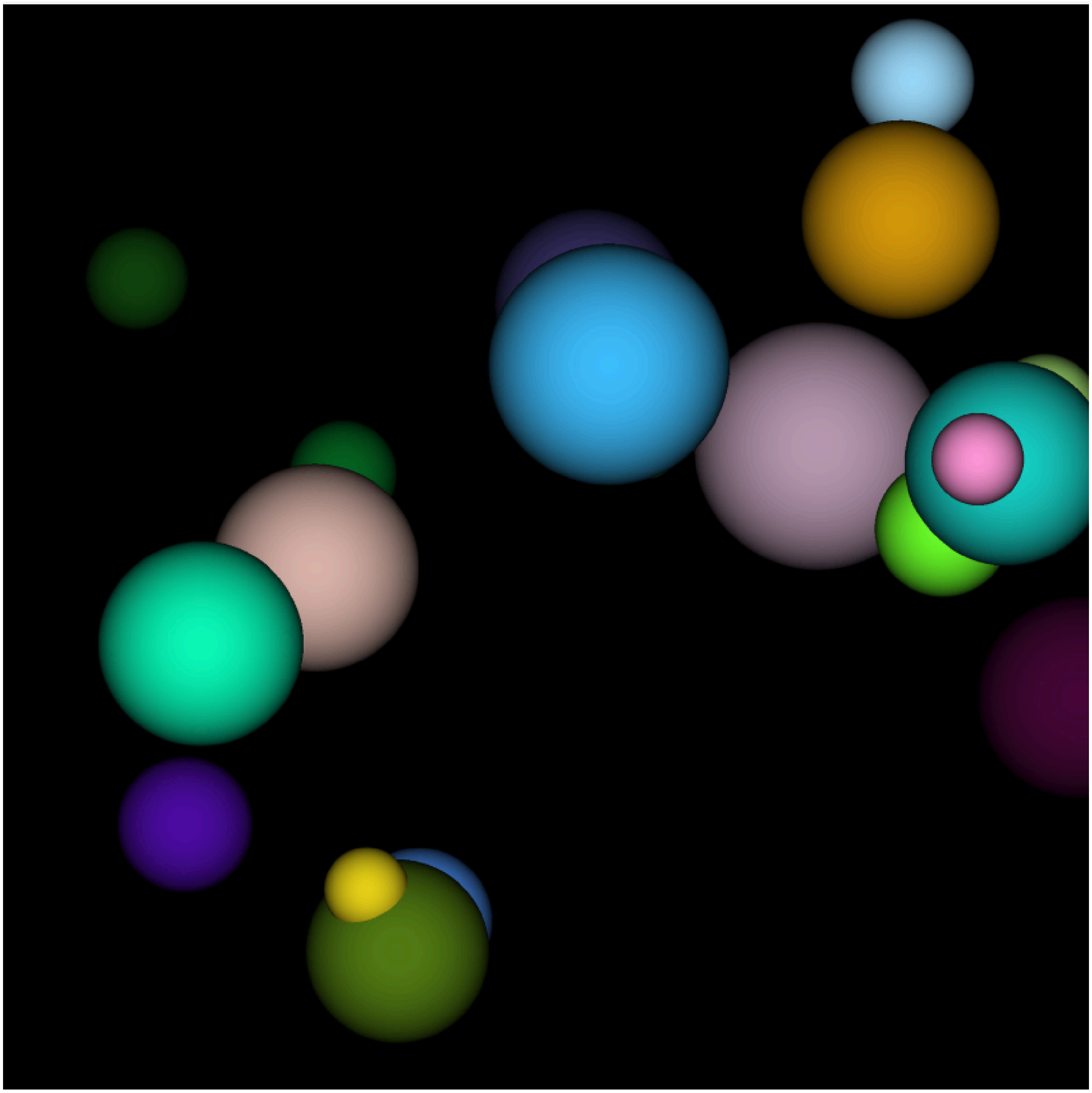
Execution Time Graph



3. Screenshots

3.1 OpenMP

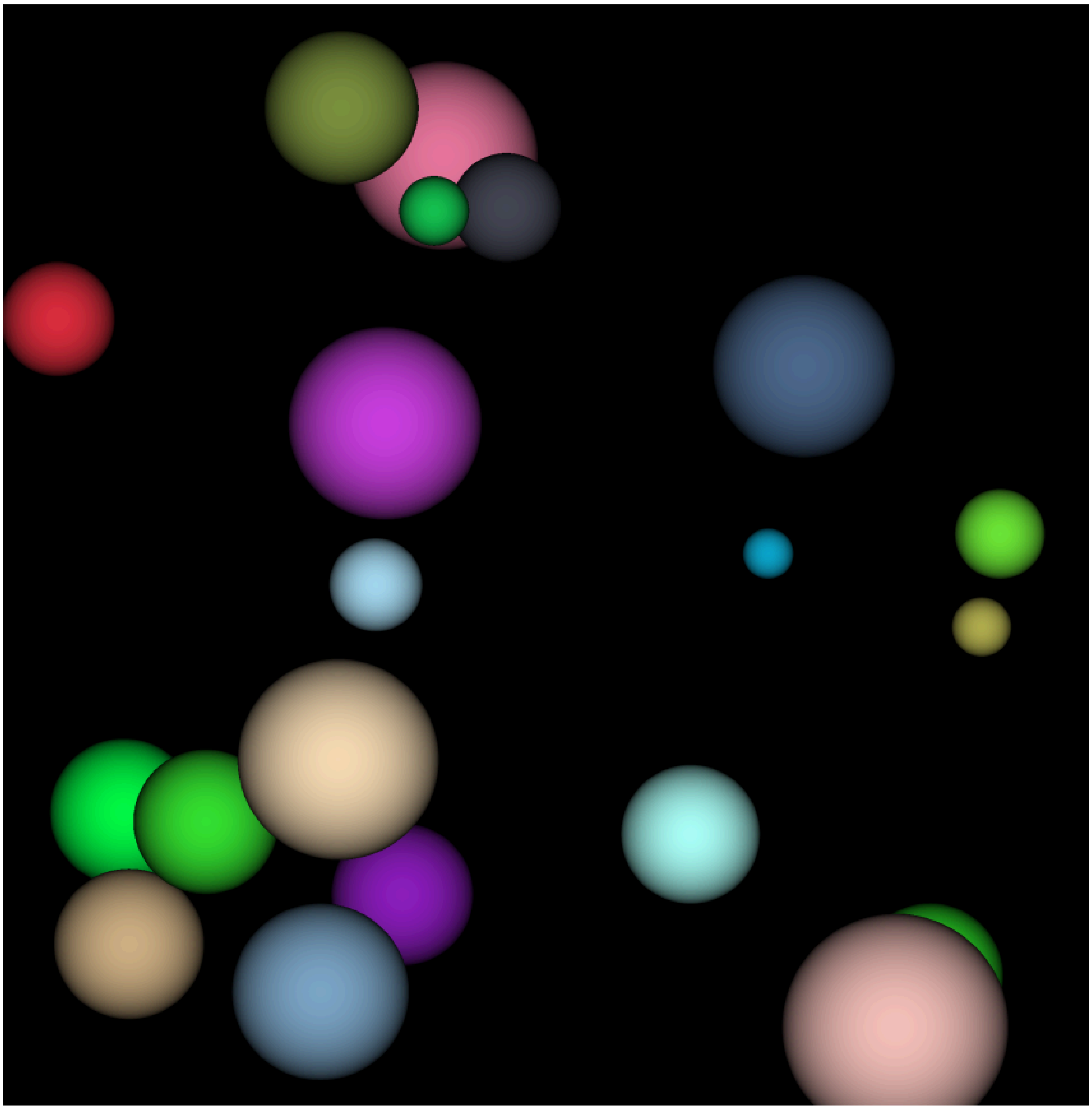
```
➜  problem1 git:(main) ✗ ./a.out 16
OpenMP (16 threads) ray tracing: 0.386 sec
[result.ppm] was generated.
➜  problem1 git:(main) ✗
```



4.2 Cuda

```
!./a.out
```

```
CUDA ray tracing: 0.001 sec
[result.ppm] was generated.
```



4. Performance Analysis

4.1 OpenMP

The OpenMP implementation shows a significant reduction in execution time as the number of threads increases. The time taken to complete the ray tracing task decreases from 1.352 seconds with 1 thread to 0.386 seconds with 16 threads, demonstrating the effectiveness of parallelization in reducing computation time.

4.2 CUDA

The CUDA implementation achieves an impressive execution time of just 0.001 seconds, showcasing the power of GPU acceleration for parallel processing tasks. This is a substantial improvement over the OpenMP implementation, highlighting the efficiency of using CUDA for computationally intensive tasks like ray tracing. This demonstrates how a GPU can handle parallel tasks much more efficiently than a CPU, especially for operations that can be massively parallelized and repetitive, such as ray tracing in computer graphics.

5. Performance Analysis

5.1 OpenMP

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

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

struct Sphere {
   float r, g, b;
   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);

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

   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;
       }
   }
```

```c
    // RGB values are clamped to the range [0, 255]
    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;
}

// Writting PPM files format
void ppm_write(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[]) {
    if (argc != 2) {
        printf("> Usage: %s [number_of_threads]\n", argv[0]);
        return 1;
    }

    int no_threads = atoi(argv[1]);
    omp_set_num_threads(no_threads);

    Sphere* spheres = (Sphere*)malloc(sizeof(Sphere) * SPHERES);
    unsigned char* bitmap = (unsigned char*)malloc(sizeof(unsigned char) *
DIM * DIM * 4);

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

    double start = omp_get_wtime();

    // Process image rendering for each sphere in parallel
    #pragma omp parallel for collapse(2) schedule(dynamic)
    for (int x = 0; x < DIM; x++) {
        for (int y = 0; y < DIM; y++) {
            kernel(x, y, spheres, bitmap);
        }
    }
```

```
    double end = omp_get_wtime();
    double elapsed = end - start;

    FILE* fp = fopen("result.ppm", "w");
    if (!fp) {
        fprintf(stderr, "Failed to open file for writing.\n");
        return 1;
    }
    ppm_write(bitmap, DIM, DIM, fp);
    fclose(fp);

    printf("OpenMP (%d threads) ray tracing: %.3f sec\n", no_threads,
elapsed);
    printf("[result.ppm] was generated.\n");

    free(bitmap);
    free(spheres);
    return 0;
}
```

5.2 Cuda Google Colab

```
// %%writefile cuda_ray.cu
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <cuda.h>
#include <ctime>

#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)
   {
       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;
   }
};

__global__ void kernel(Sphere *s, 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 / 2);
   float oy = (y - DIM / 2);

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

   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, const char
*filename)
{
    FILE *fp = fopen(filename, "w");
    if (!fp)
    {
        fprintf(stderr, "Failed to write file.\n");
        return;
    }
    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");
    }
    fclose(fp);
}

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

```
    Sphere *d_spheres;
    unsigned char *d_bitmap;
    unsigned char *h_bitmap = (unsigned char *)malloc(DIM * DIM * 4);

    cudaMalloc(&d_spheres, sizeof(Sphere) * SPHERES);
    cudaMemcpy(d_spheres, h_spheres, sizeof(Sphere) * SPHERES,
cudaMemcpyHostToDevice);

    cudaMalloc(&d_bitmap, DIM * DIM * 4);

    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks((DIM + 15) / 16, (DIM + 15) / 16);

    // Start timing
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start);

    kernel<<<numBlocks, threadsPerBlock>>>(d_spheres, d_bitmap);
    cudaDeviceSynchronize();

    cudaEventRecord(stop);
    cudaEventSynchronize(stop);

    float milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start, stop);

    // Copy result back
    cudaMemcpy(h_bitmap, d_bitmap, DIM * DIM * 4, cudaMemcpyDeviceToHost);
    ppm_write(h_bitmap, DIM, DIM, "result.ppm");

    printf("CUDA ray tracing: %.3f sec\n", milliseconds / 1000.0f);
    printf("[result.ppm] was generated.\n");

    // Cleanup
    free(h_bitmap);
    cudaFree(d_bitmap);
    cudaFree(d_spheres);

    return 0;
}
```