Proj4 – problem1

과목명	멀티코어컴퓨팅 02분반
교수명	손봉수 교수님
제출일	2023.06.09
학　과	소프트웨어대학 소프트웨어학부
작성자	20200641 임수현

(a) environment
OS - window
CPU - 11th Gen Intel(R) Core(TM) i7-1195G7
GPU - Colab use.

(b) how to compile
openmp : using wsl and g++

```
$g++ -fopenmp openmp_ray.cpp
```

cuda: using colab

```
>> !nvcc cuda_ray.cu
```

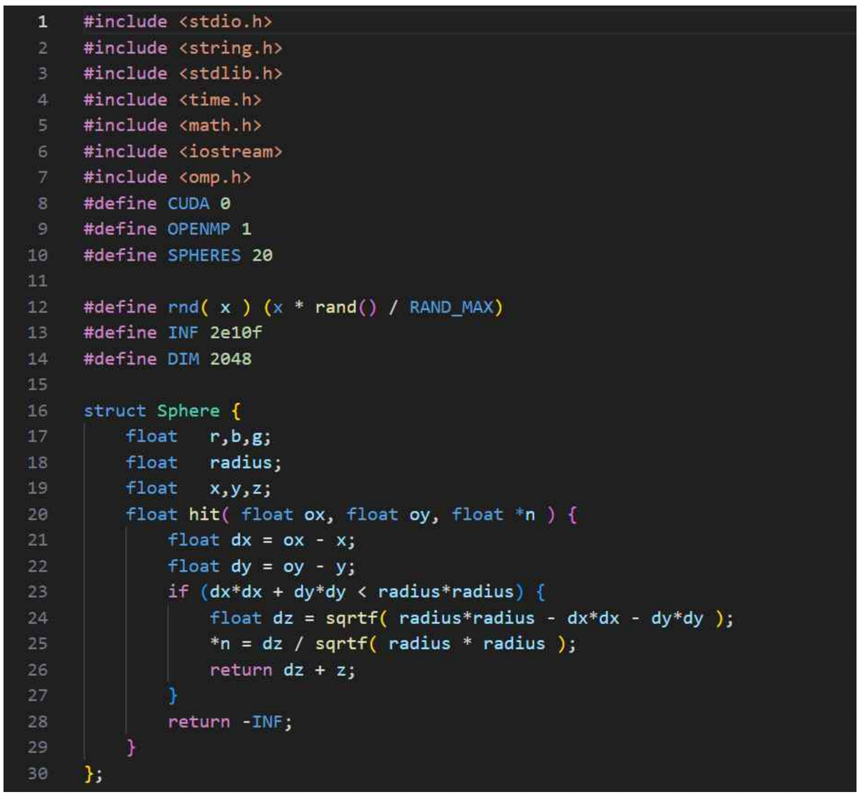
(c) how to execute
openmp: using wsl and g++

```
$./a.out 8 result.ppm
```

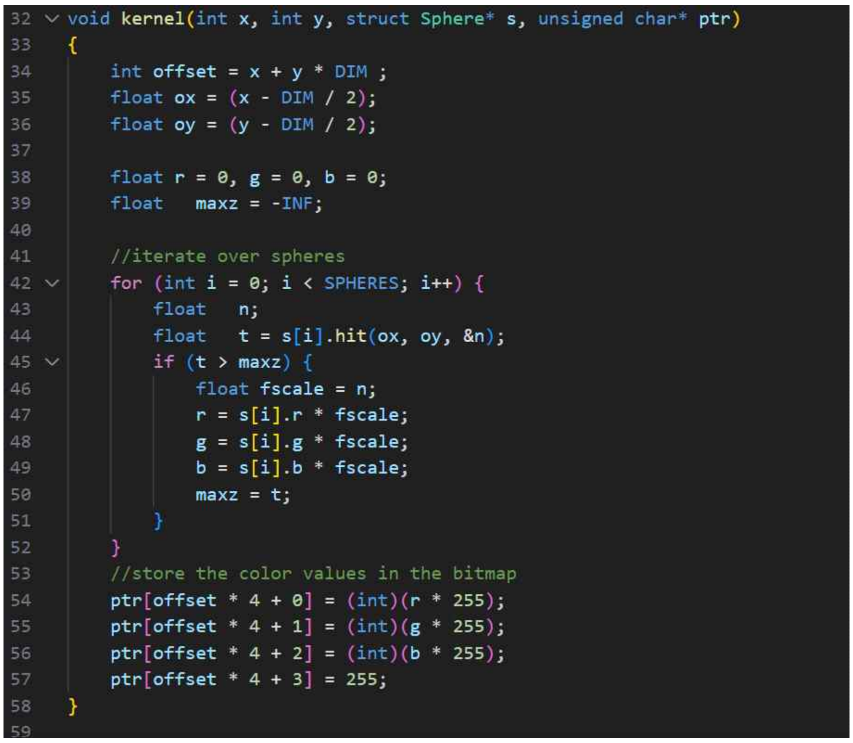
cuda: using colab

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

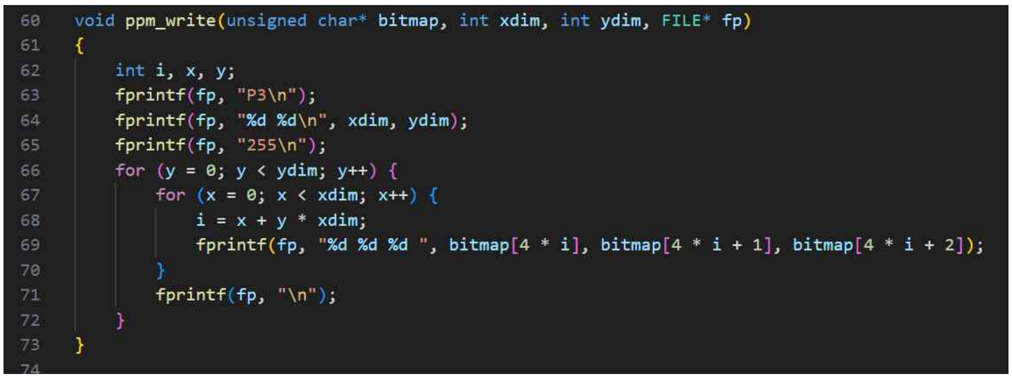
openmp_ray.cpp source code



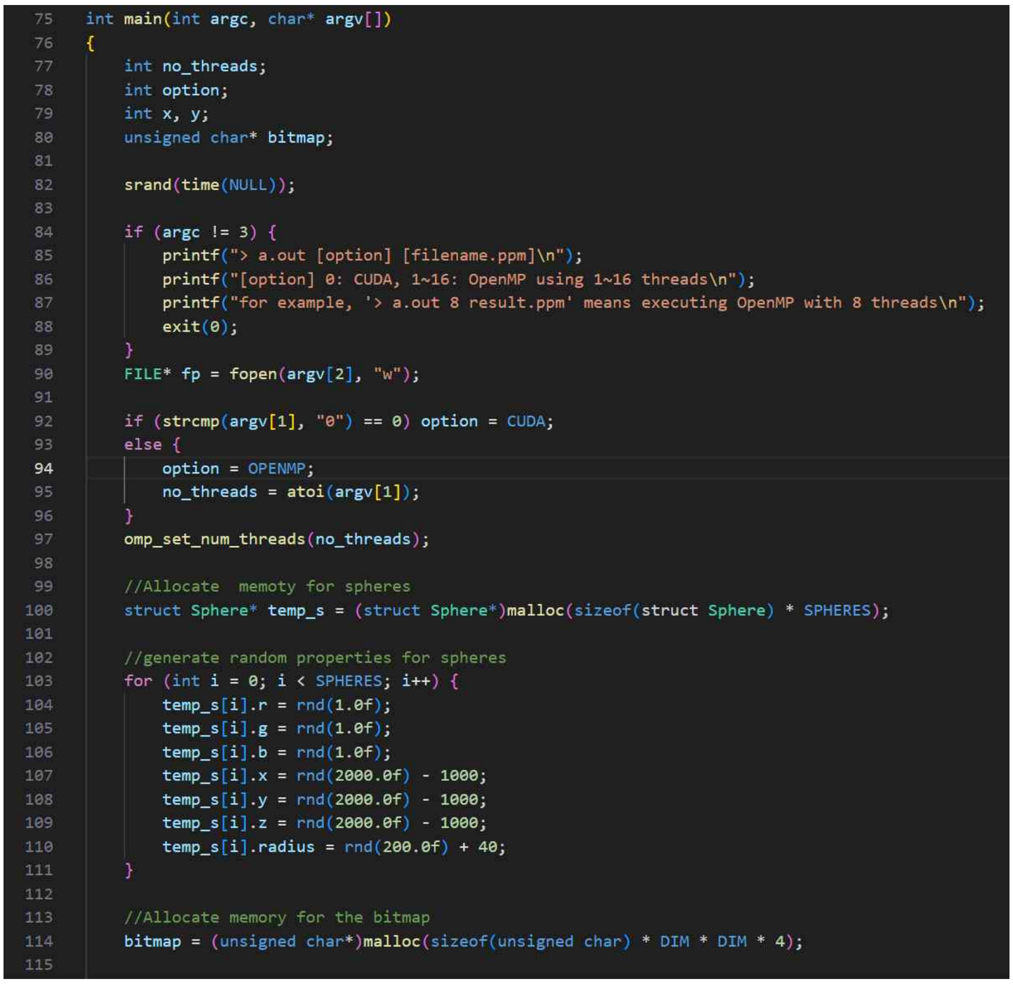
```cpp
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <iostream>
#include <omp.h>
#define CUDA 0
#define OPENMP 1
#define SPHERES 20

#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;
    }
};
```



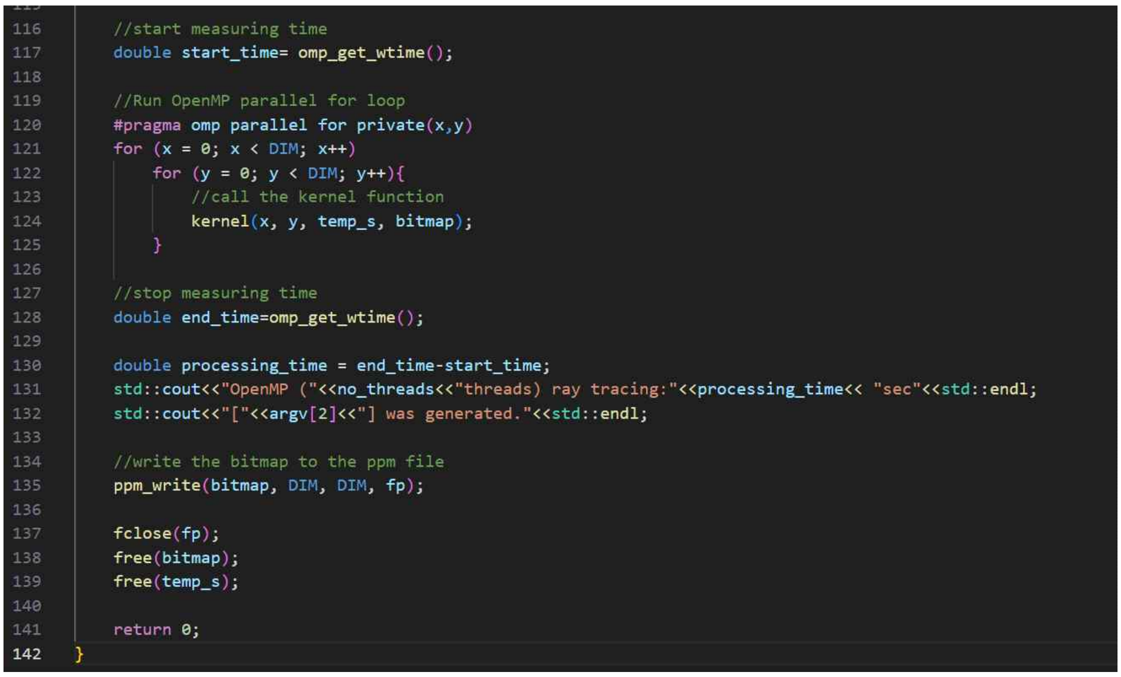
```cpp
void kernel(int x, int y, struct Sphere* s, unsigned char* ptr)
{
    int offset = x + y * DIM ;
    float ox = (x - DIM / 2);
    float oy = (y - DIM / 2);

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

    //iterate over spheres
    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;
        }
    }
    //store the color values in the bitmap
    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;
}
```



```c
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");
    }
}
```

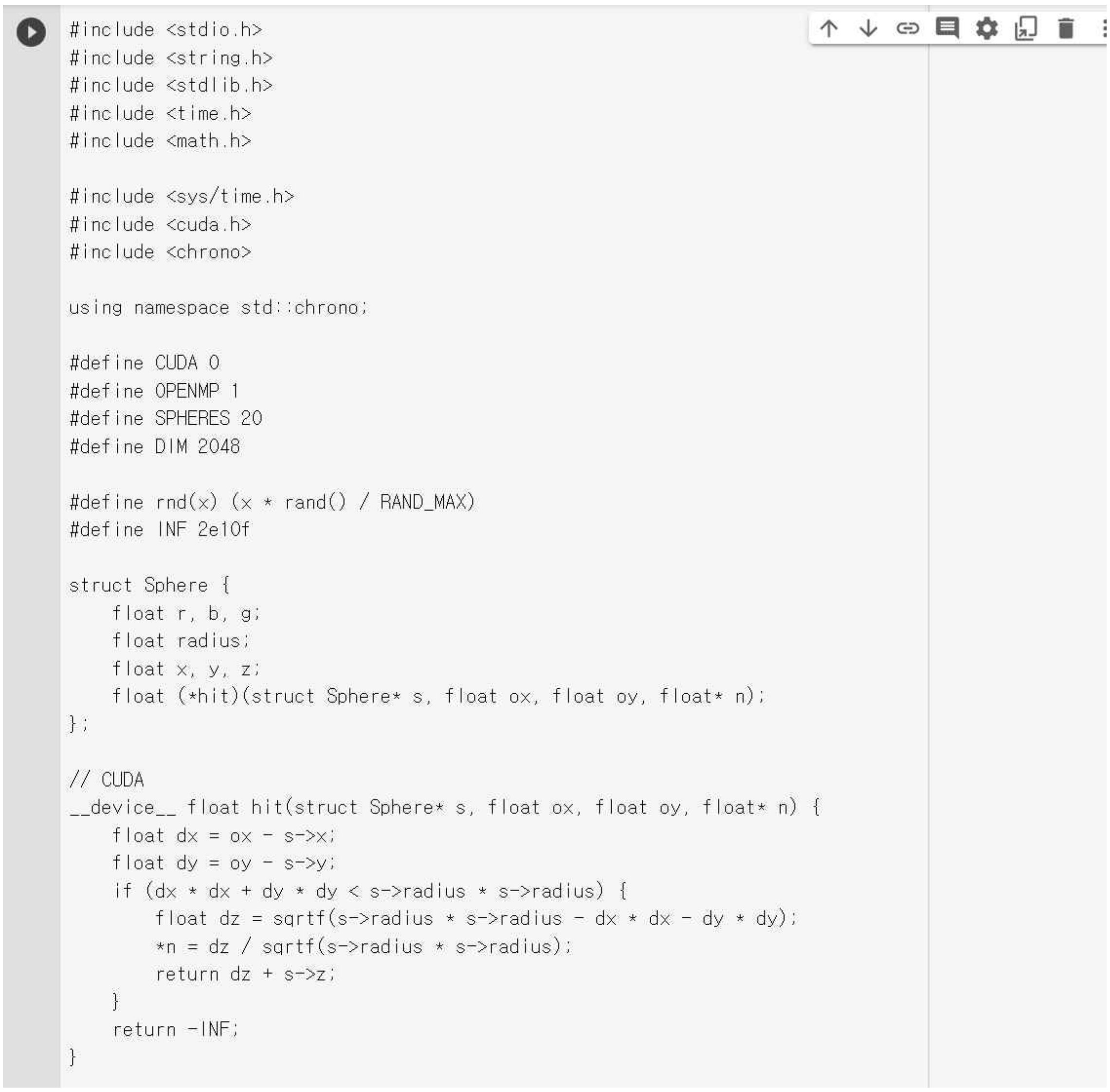


```c
int main(int argc, char* argv[])
{
    int no_threads;
    int option;
    int x, y;
    unsigned char* bitmap;

    srand(time(NULL));

    if (argc != 3) {
        printf("> a.out [option] [filename.ppm]\n");
        printf("[option] 0: CUDA, 1~16: OpenMP using 1~16 threads\n");
        printf("for example, '> a.out 8 result.ppm' means executing OpenMP with 8 threads\n");
        exit(0);
    }
    FILE* fp = fopen(argv[2], "w");

    if (strcmp(argv[1], "0") == 0) option = CUDA;
    else {
        option = OPENMP;
        no_threads = atoi(argv[1]);
    }
    omp_set_num_threads(no_threads);

    //Allocate  memoty for spheres
    struct Sphere* temp_s = (struct Sphere*)malloc(sizeof(struct Sphere) * SPHERES);

    //generate random properties for 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;
    }

    //Allocate memory for the bitmap
    bitmap = (unsigned char*)malloc(sizeof(unsigned char) * DIM * DIM * 4);

```

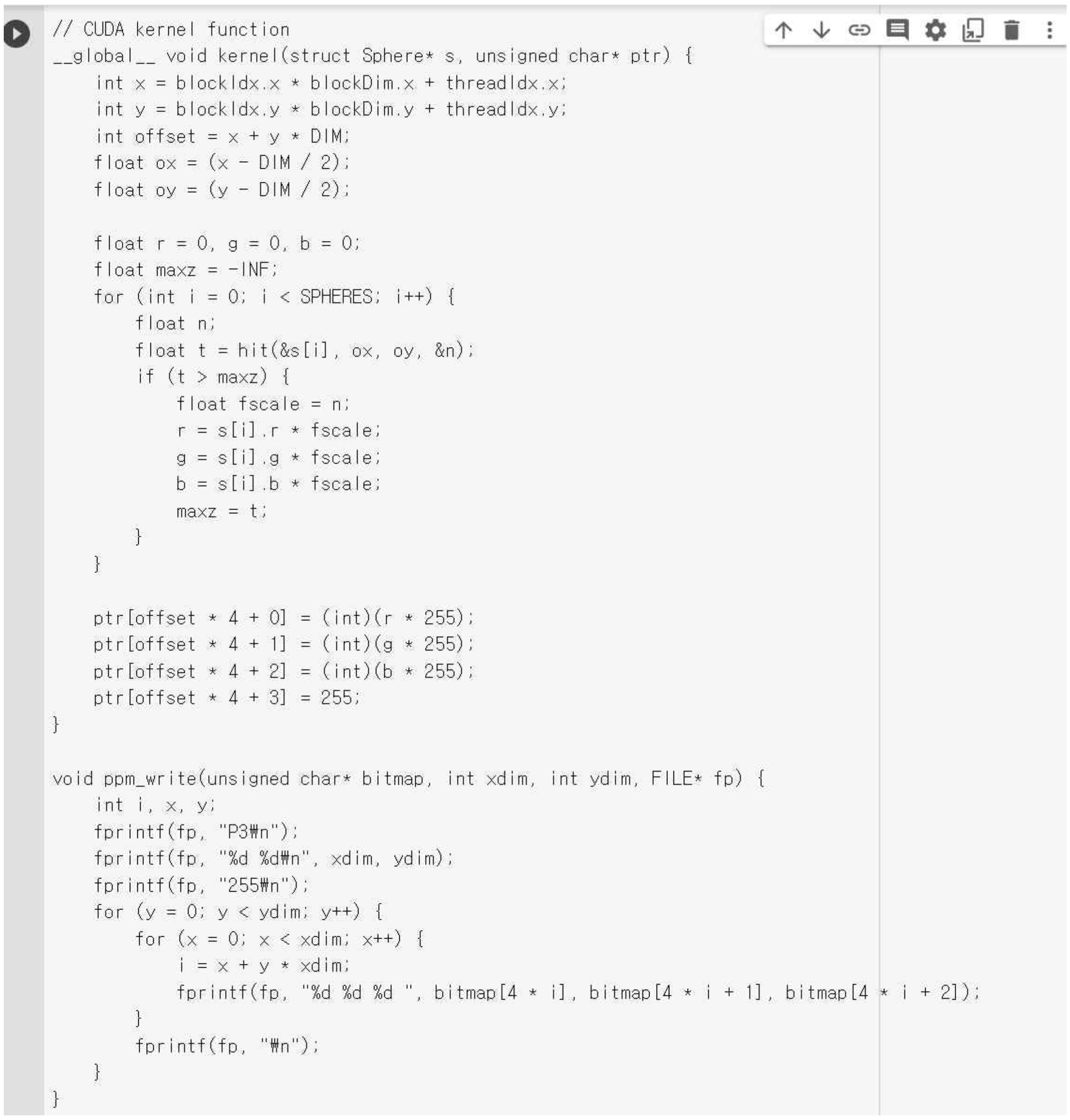


```
    //start measuring time
    double start_time= omp_get_wtime();

    //Run OpenMP parallel for loop
    #pragma omp parallel for private(x,y)
    for (x = 0; x < DIM; x++)
        for (y = 0; y < DIM; y++){
            //call the kernel function
            kernel(x, y, temp_s, bitmap);
        }

    //stop measuring time
    double end_time=omp_get_wtime();

    double processing_time = end_time-start_time;
    std::cout<<"OpenMP ("<<no_threads<<"threads) ray tracing:"<<processing_time<< "sec"<<std::endl;
    std::cout<<"["<<argv[2]<<"] was generated."<<std::endl;

    //write the bitmap to the ppm file
    ppm_write(bitmap, DIM, DIM, fp);

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

    return 0;
}
```

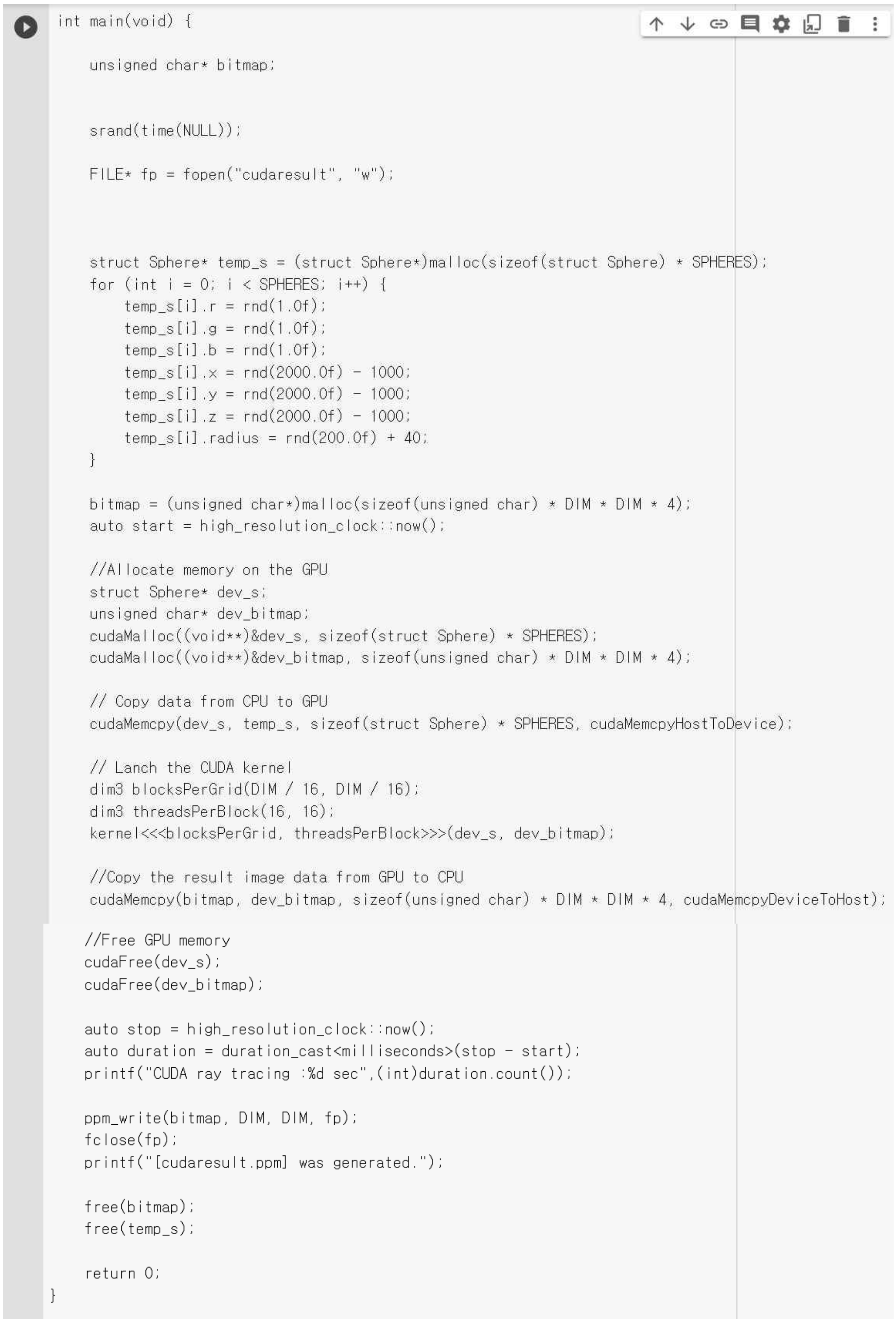
cuda_ray.cu



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

#include <sys/time.h>
#include <cuda.h>
#include <chrono>

using namespace std::chrono;

#define CUDA 0
#define OPENMP 1
#define SPHERES 20
#define DIM 2048

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

struct Sphere {
    float r, b, g;
    float radius;
    float x, y, z;
    float (*hit)(struct Sphere* s, float ox, float oy, float* n);
};

// CUDA
__device__ float hit(struct Sphere* s, float ox, float oy, float* n) {
    float dx = ox - s->x;
    float dy = oy - s->y;
    if (dx * dx + dy * dy < s->radius * s->radius) {
        float dz = sqrtf(s->radius * s->radius - dx * dx - dy * dy);
        *n = dz / sqrtf(s->radius * s->radius);
        return dz + s->z;
    }
    return -INF;
}
```



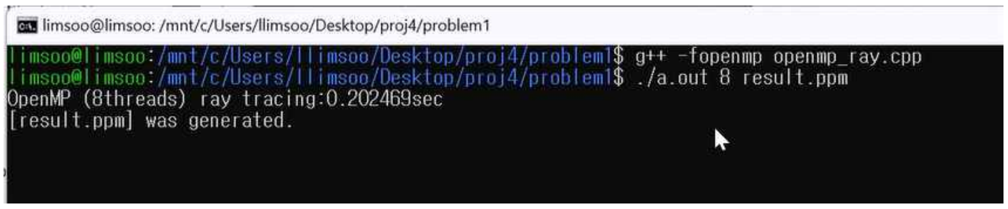
```
// CUDA kernel function
__global__ void kernel(struct Sphere* s, unsigned char* ptr) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    int offset = x + y * DIM;
    float ox = (x - DIM / 2);
    float oy = (y - DIM / 2);

    float r = 0, g = 0, b = 0;
    float maxz = -INF;
    for (int i = 0; i < SPHERES; i++) {
        float n;
        float t = hit(&s[i], 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");
    }
}
```



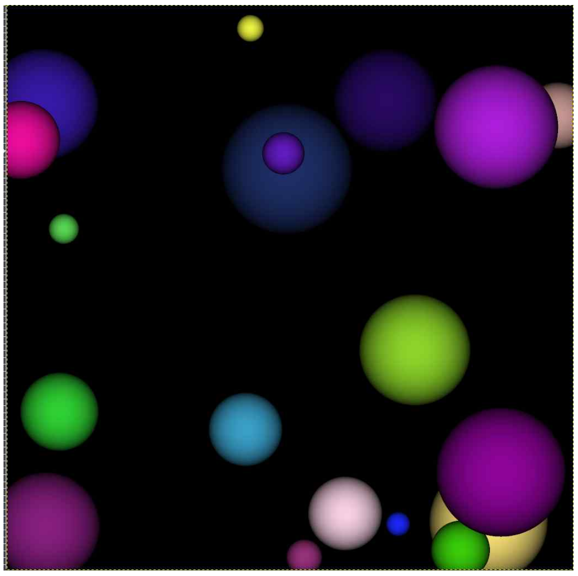
```c
int main(void) {

    unsigned char* bitmap;


    srand(time(NULL));

    FILE* fp = fopen("cudaresult", "w");



    struct Sphere* temp_s = (struct Sphere*)malloc(sizeof(struct 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);
    auto start = high_resolution_clock::now();

    //Allocate memory on the GPU
    struct Sphere* dev_s;
    unsigned char* dev_bitmap;
    cudaMalloc((void**)&dev_s, sizeof(struct Sphere) * SPHERES);
    cudaMalloc((void**)&dev_bitmap, sizeof(unsigned char) * DIM * DIM * 4);

    // Copy data from CPU to GPU
    cudaMemcpy(dev_s, temp_s, sizeof(struct Sphere) * SPHERES, cudaMemcpyHostToDevice);

    // Lanch the CUDA kernel
    dim3 blocksPerGrid(DIM / 16, DIM / 16);
    dim3 threadsPerBlock(16, 16);
    kernel<<<blocksPerGrid, threadsPerBlock>>>(dev_s, dev_bitmap);

    //Copy the result image data from GPU to CPU
    cudaMemcpy(bitmap, dev_bitmap, sizeof(unsigned char) * DIM * DIM * 4, cudaMemcpyDeviceToHost);

    //Free GPU memory
    cudaFree(dev_s);
    cudaFree(dev_bitmap);

    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<milliseconds>(stop - start);
    printf("CUDA ray tracing :%d sec",(int)duration.count());

    ppm_write(bitmap, DIM, DIM, fp);
    fclose(fp);
    printf("[cudaresult.ppm] was generated.");

    free(bitmap);
    free(temp_s);

    return 0;
}
```

Test Result

OPENMP

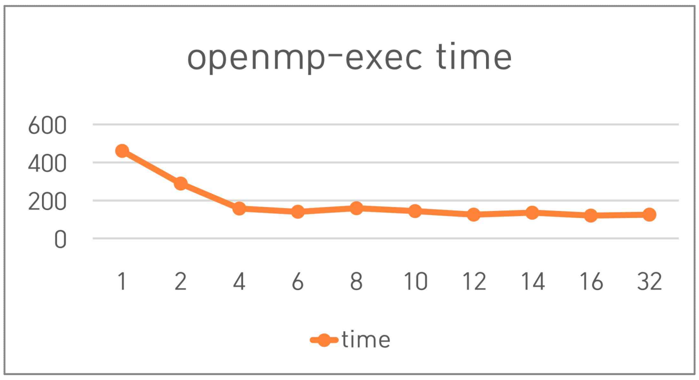


```
limsoo@limsoo: /mnt/c/Users/llimsoo/Desktop/proj4/problem1
limsoo@limsoo:/mnt/c/Users/llimsoo/Desktop/proj4/problem1$ g++ -fopenmp openmp_ray.cpp
limsoo@limsoo:/mnt/c/Users/llimsoo/Desktop/proj4/problem1$ ./a.out 8 result.ppm
OpenMP (8threads) ray tracing:0.202469sec
[result.ppm] was generated.
```



# thread	1	2	4	6	8
time	460.114	287.943	157.142	139.684	158.794

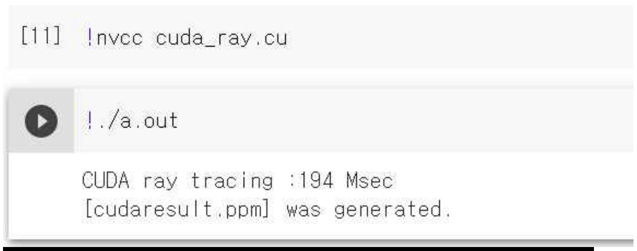
# thread	10	12	14	16	32
time	143.624	124.994	135.345	120.164	124.558



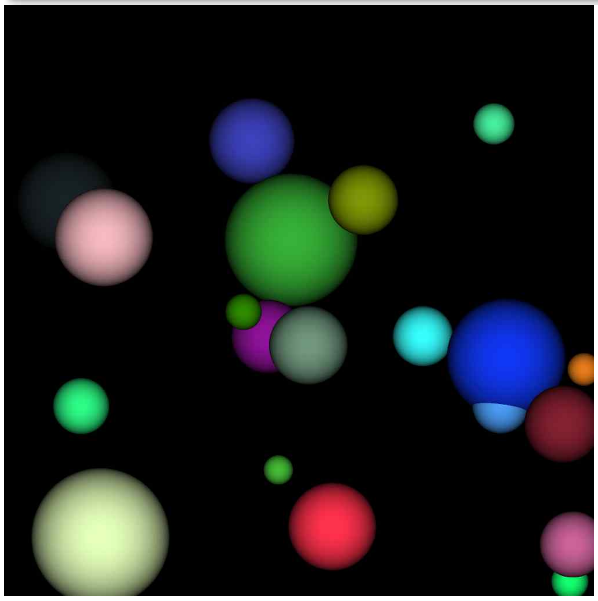
OpenMP is a programming model for multi-threading, making it easy to implement parallel tasks. Use the #pragmaompparallel for statement to run the kernel function as a multi-thread. This allows kernel functions to run simultaneously across multiple threads to parallelize calculations. This reduces calculation time. Running with OpenMP took 460 msec when calculated with a single thread, but running time was reduced to 120 msec when calculated with a multi-thread using OpenMP. This shows that using OpenMP can improve computation speed through parallel processing.

CUDA

exec time: 194 msec



```
[11] !nvcc cuda_ray.cu

!./a.out

CUDA ray tracing :194 Msec
[cudaresult.ppm] was generated.
```



CUDA allows you to improve computation speed through parallel processing compared to the CPU. CUDA code leverages the parallel processing power of the GPU to accelerate image rendering.

In the given code, the kernel function runs as the CUDA kernel. The kernel function performs ray tracing for each pixel and calculates the intersection with each spear to determine the color. Because these calculations are processed in parallel, CUDA allows multiple threads to run simultaneously to speed up the calculation.