2025.1 Multicore Computing Project #4

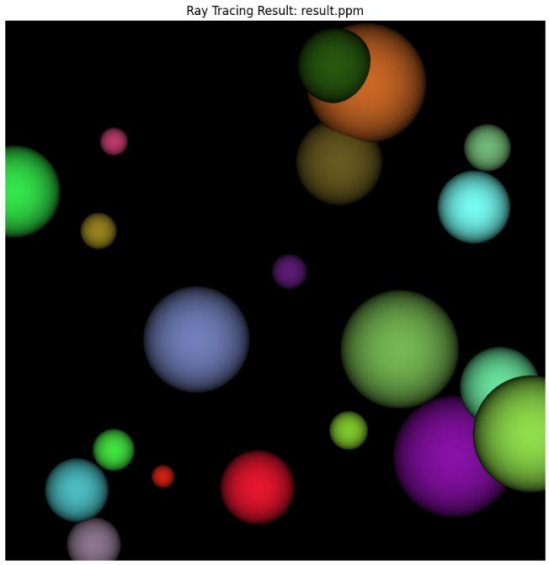
-problem1-

20223961 김수아

1. Environment
   - Google Colab (T4 GPU)
2. Result
   A. Capture Image
      i. OpenMP
         1. Execute Result

```
Overwriting openmp_ray.cpp

[8]  !g++ -fopenmp -O3 -o openmp_ray openmp_ray.cpp

[9]  !./openmp_ray 8

     OpenMP (8 threads) ray tracing: 0.319 sec
     [result.ppm] was generated.
```

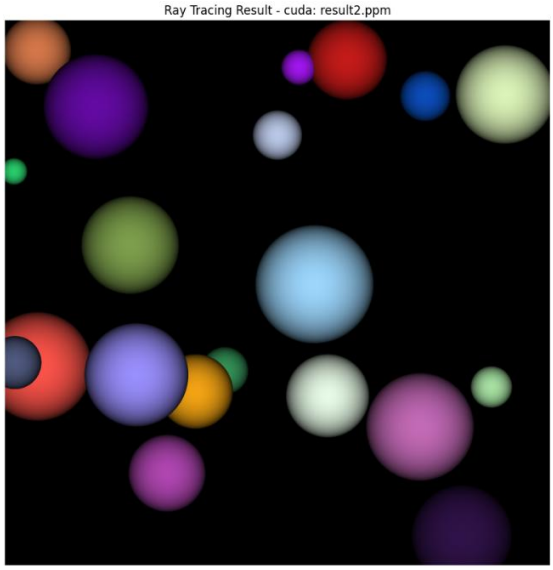
         2. Generated Image



      ii. CUDA
         1. Execute Result

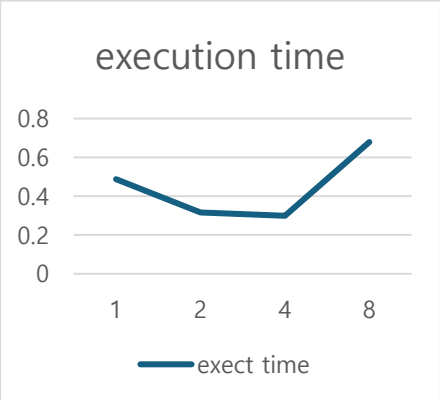
```
Writing cuda_ray.cu

[13] !nvcc -arch=sm_75 -o cuda_ray cuda_ray.cu

[14] !./cuda_ray

CUDA ray tracing: 0.013 sec
[result2.ppm] was generated.
```

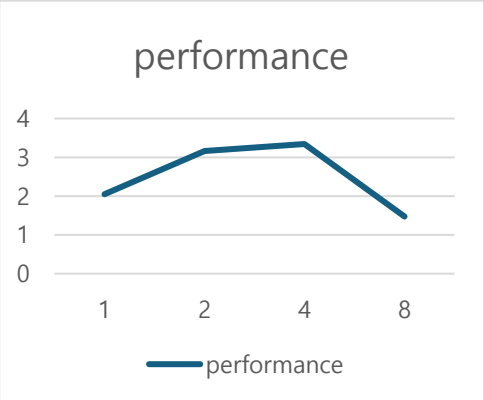
2. Generated Image



B. Execution Time, Performance

thread num	1	2	4	8	cuda
exec time	0.488	0.316	**0.299**	0.679	**0.013**
performance	2.049	3.165	**3.344**	1.473	**76.923**





C. Result Explanation

i. OpenMP

Optimal performance was achieved with 4 threads and performance decrease with 8 threads. Up to 4 threads, each thread could be allocated sufficient computational resources. Performance degradation may be caused by overhead and context switching costs. For example

performance loss during switching threads too often.

ii. CUDA

CUDA version was the fastest performance, it was approximately 23 times faster than the best record of OpenMP. It may be caused by much more number of cores in CUDA working simultaneously. Ray tracing can be made by calculating simple but many operation, and CUDA is strong to calculate simple operation effectively.

3. Compile

A. OpenMP

- %%writefile openmp_ray.cpp
  [total source code]
- !g++ -fopenmp -O3 -o openmp_ray openmp_ray.cpp
- !./openmp_ray [# of threads]

B. CUDA

- %%writefile cuda_ray.cpp
  [total source code]
- !nvcc -arch=sm_75 -o cuda_ray cuda_ray.cu
- !./cuda_ray

4. Source Code

A. openmp_ray.cpp

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

#define SPHERES 20 //spheres number to rendering
#define rnd( x ) (x * rand() / RAND_MAX) //macro for making random
number
#define INF 2e10f
#define DIM 2048 //image size 2048*2048

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

```
        float   radius;
        float   x,y,z;
        //check if ray and sphere hit each other
        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;
        for(int i=0; i<SPHERES; i++) {
            float   n;
            float   t = s[i].hit( ox, oy, &n );
            //if present sphere is more closer
            if (t > maxz) {
                float fscale = n;
                r = s[i].r * fscale;
                g = s[i].g * fscale;
                b = s[i].b * fscale;
                maxz = t;
```

```
            }
        }
        //calculated color value -> 0~225 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;
}
//bitmap data -> ppm image file
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; //thread number
        if (argc != 2) {
            printf("Usage: %s [number_of_threads]₩n", argv[0]);
            exit(1);
        }
        no_threads = atoi(argv[1]);

        omp_set_num_threads(no_threads);
```

```
	srand(time(NULL));

	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;
	}

	unsigned char* bitmap = (unsigned char*)malloc(sizeof(unsigned
char)*DIM*DIM*4);
	//calculate execution time
	double start_time = omp_get_wtime();
	//parallelization with openMP
	//make 2 for loop as one parallel area
	//make thread to be allocated taskes dynamically
	#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, temp_s, bitmap);
		}
	}

	double end_time = omp_get_wtime();
	double elapsed_time = end_time - start_time;

	printf("OpenMP (%d threads) ray tracing: %.3f sec₩n", no_threads,
elapsed_time);

	FILE* fp = fopen("result.ppm", "w");
```

```
    if (fp) {
        ppm_write(bitmap, DIM, DIM, fp);
        fclose(fp);
        printf("[result.ppm] was generated.₩n");
    } else {
        printf("Error: Could not create result.ppm₩n");
    }

    free(bitmap);
    free(temp_s);

    return 0;
}
```

B. cuda_ray.cu

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

#define SPHERES 20 //spheres number to rendering
#define rnd( x ) (x * rand() / RAND_MAX) //macro for making random
number
#define INF 2e10f
#define DIM 2048 //image size 2048*2048

struct Sphere {
    float r, b, g;
    float radius;
    float x, y, z;
    //check if ray and sphere hit each other
    //it is a function that can be run at GPU device
    __device__ float hit(float ox, float oy, float *n) {
        float dx = ox - x;
```

```
            float dy = oy - y;
            //if pixel value is in sphere
            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 cuda_kernel(Sphere* s, unsigned char* ptr) {
        //calculate current thread's 2D location
        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 present sphere is more closer
            if (t > maxz) {
                float fscale = n;
                r = s[i].r * fscale;
                g = s[i].g * fscale;
                b = s[i].b * fscale;
                maxz = t;
```

```
            }
        }
        //calculated color value -> 0~225 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;
}
//bitmap data -> ppm image file
void ppm_write(unsigned char* bitmap, int xdim, int ydim, const char*
filename) {
        FILE* fp = fopen(filename, "w");
        if (!fp) {
            printf("Error: Cannot create file %s₩n", filename);
            return;
        }

        fprintf(fp, "P3₩n");
        fprintf(fp, "%d %d₩n", xdim, ydim);
        fprintf(fp, "255₩n");

        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);
}
//Check if error occured after call CUDA API
#define CUDA_CHECK(call) ₩
        do { ₩
            cudaError_t error = call; ₩
```

```
            if (error != cudaSuccess) { ₩
                printf("CUDA error at %s:%d - %s₩n", __FILE__, __LINE__,
cudaGetErrorString(error)); ₩
                exit(1); ₩
            } ₩
        } while(0)

int main() {
    srand(time(NULL));

    Sphere *h_spheres = (Sphere*)malloc(sizeof(Sphere) * SPHERES);
    unsigned char *h_bitmap = (unsigned char*)malloc(sizeof(unsigned
char) * DIM * DIM * 4);

    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;
    }
    //GPU memory pointer
    Sphere *d_spheres;
    unsigned char *d_bitmap;
    //allocate GPU memory
    CUDA_CHECK(cudaMalloc((void**)&d_spheres, sizeof(Sphere) *
SPHERES));
    CUDA_CHECK(cudaMalloc((void**)&d_bitmap, sizeof(unsigned char) *
DIM * DIM * 4));
    //copy SPHERE data host -> device
    CUDA_CHECK(cudaMemcpy(d_spheres, h_spheres, sizeof(Sphere) *
SPHERES, cudaMemcpyHostToDevice));
```

```
    dim3 blockSize(16, 16);   // 16x16 threads per block
    dim3 gridSize((DIM + blockSize.x - 1) / blockSize.x, (DIM + blockSize.y -
1) / blockSize.y);
    //calculate execution time
    clock_t start_time = clock();
    //run CUDA kernel
    cuda_kernel<<<gridSize, blockSize>>>(d_spheres, d_bitmap);
    CUDA_CHECK(cudaGetLastError());
    CUDA_CHECK(cudaDeviceSynchronize());
    CUDA_CHECK(cudaMemcpy(h_bitmap, d_bitmap, sizeof(unsigned char) *
DIM * DIM * 4, cudaMemcpyDeviceToHost));
    clock_t end_time = clock();
    double elapsed_time = ((double)(end_time - start_time)) /
CLOCKS_PER_SEC;

    printf("CUDA ray tracing: %.3f sec₩n", elapsed_time);

    ppm_write(h_bitmap, DIM, DIM, "result2.ppm");
    printf("[result2.ppm] was generated.₩n");
    CUDA_CHECK(cudaFree(d_spheres));
    CUDA_CHECK(cudaFree(d_bitmap));
    free(h_spheres);
    free(h_bitmap);

    return 0;
}
```