

# REPORT

#problem1\_document



과목명	멀티코어컴퓨팅 01분반
교수명	손봉수
학번	20206319
학과	소프트웨어학부 소프트웨어전공
이름	김가연

## Case I )openmp\_ray.cpp

### (a) Execution environment (OS/CPU/GPU type or Colab?)

The CPU type of this machine is 12th Gen Intel(R) Core(TM) i9-12900K.

It has 16 cores, 24 logical processors(when hyperthreading on), and clock speed is 3.20 GHz.

The memory size is 32.0 GB. The speed of the memory is 4400 MHz.

The OS type is 64-bit operating system, x64 based processor.

### (b) How to compile

This code was written in the Visual Studio Code environment. The code is written in C++. Compilation is done in Terminal. The command to compile is “gcc -fopenmp openmp\_ray.cpp -o openmp\_ray.exe”.

### (c) How to execute

According to the given output statement example, after compiling, enter “./openmp\_ray.exe #number\_of\_threads#” in the terminal to run the program.

### (d) Entire source code with appropriate comments

```
#define _CRT_SECURE_NO_WARNINGS

#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, b, g; //color of sphere
    float radius; //radius of sphere
    float x, y, z; //coordinates of the center of the sphere
    float hit(float ox, float oy, float* n) { //determines if a given point intersects a
sphere and computes the depth at the point of intersection
        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) //A function that performs ray
tracing at (x,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 = 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) //Function to save
bitmap data as PPM image file
{
    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;
    unsigned char* bitmap;

    srand(time(NULL));

    if (argc != 2) { //When two items are not entered in input
        printf("> a.out OpenMP [result.ppm]\n");
        printf("OpenMP using 1~16 threads\n");
        printf("[result.ppm] has to generated\n");
        exit(0);
    }
    FILE* fp = fopen("result.ppm", "w");
    no_threads = atoi(argv[1]); //input의 스레드 개수

    omp_set_num_threads(no_threads);

    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 = omp_get_wtime();

#pragma omp parallel for collapse(2)
    for (int x = 0; x < DIM; x++) {
        for (int y = 0; y < DIM; y++) {
            kernel(x, y, temp_s, bitmap);
        }
    }
    double end = omp_get_wtime();

    fflush(stdout); //empty buffer
    printf("OpenMP (%d threads) ray tracing: %8.5f sec\n", no_threads, end - start);
    ppm_write(bitmap, DIM, DIM, fp);
    printf("[result.ppm] was generated.\n"); //output

```

```

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

    return 0;
}

```

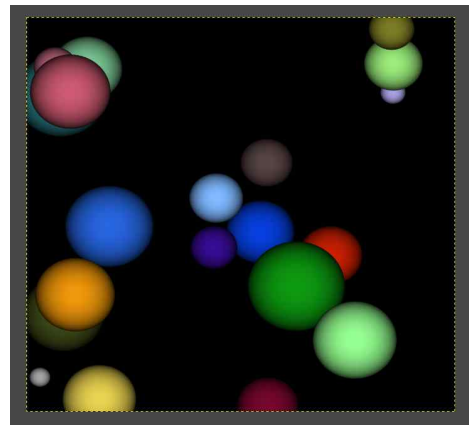
### (e) Program output results and ray-tracing result pictures

This is an example when the number of threads is 8.

```

PS C:\Users\pc03\Desktop\probi> gcc -fopenmp openmp_ray.cpp -o openmp_ray.exe
PS C:\Users\pc03\Desktop\probi> ./openmp_ray.exe 8
OpenMP (8 threads) ray tracing: 0.04600 sec
[result.ppm] was generated.
PS C:\Users\pc03\Desktop\probi>

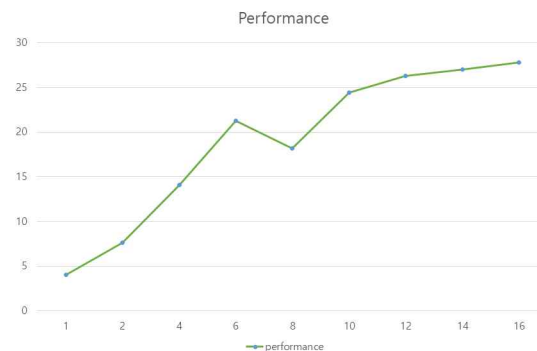
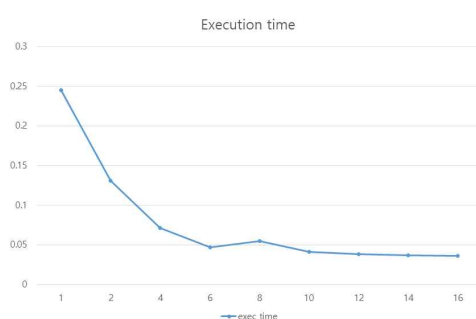
```



### (f) Experimental results : measuring the performance of OpenMP implementation and the given single threaded CPU implementation. Finally, Show the performance results with respect to various number of threads including graphs and interpretation/explanation.

\*The performance is expressed up to 3 decimal places.

	1	2	4	6	8	10	12	14	16
exec time (unit: ms)	0.245	0.131	0.071	0.047	0.055	0.041	0.038	0.037	0.036
	1	2	4	6	8	10	12	14	16
performance (1/exec time)	4.082	7.636	14.085	21.277	18.182	24.390	26.312	27.027	27.778



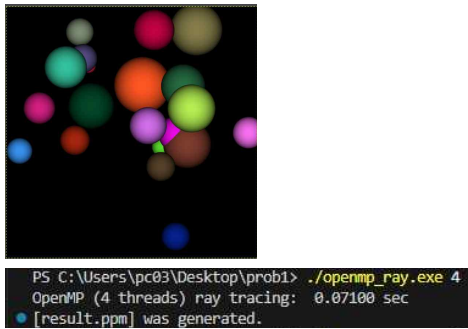
case 1)1 thread



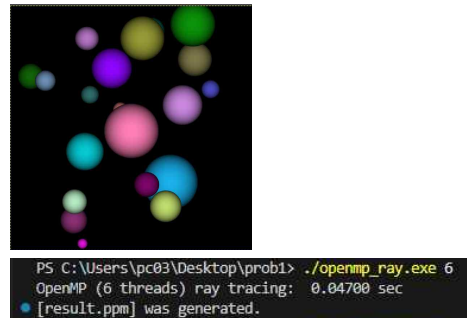
case 2)2 threads



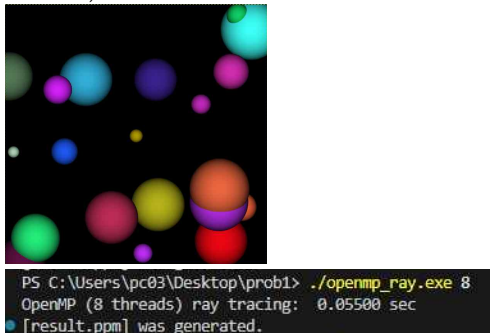
case 3)4 threads



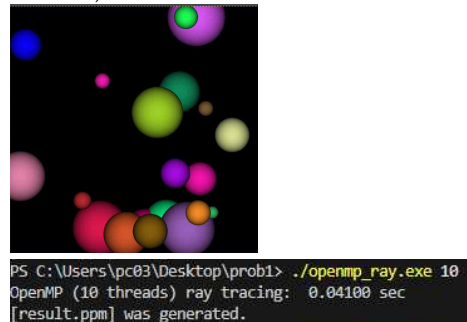
case 4)6 threads



case 5)8 thread



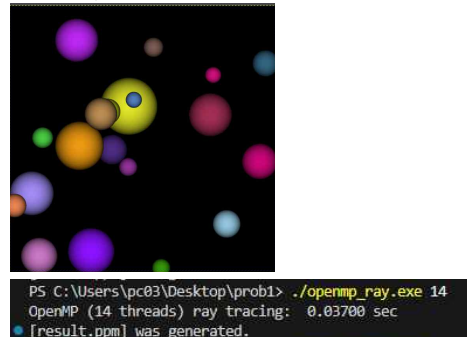
case 6)10 threads



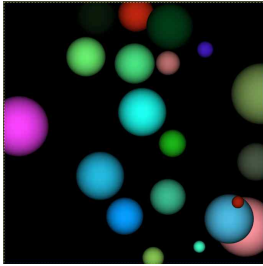
case 7)12 thread



case 8)14 threads



case 9)16 thread



```
PS C:\Users\pc03\Desktop\prob1> ./openmp_ray.exe 16
OpenMP (16 threads) ray tracing: 0.03600 sec
[result.ppm] was generated.
```

Execution time was used to measure performance. It was calculated using  $\text{performance} = 1/\text{execution time}$ .

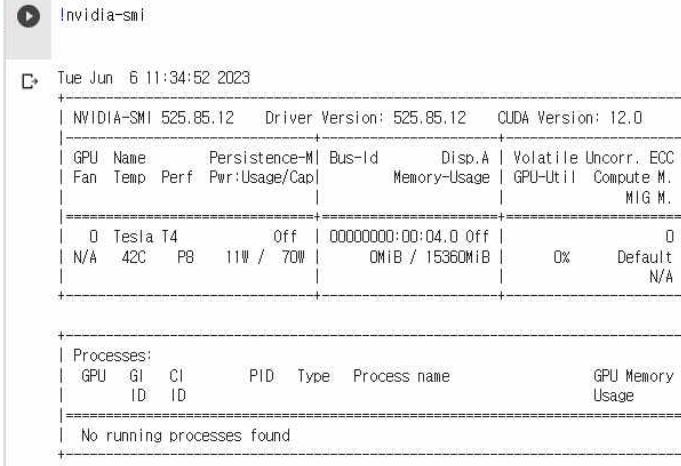
Execution time decreases as the number of threads increases, and performance, which is inversely proportional to it, increases as the number of threads increases. In the situations that have been set as tasks so far, if the number of threads is more than the number considered appropriate for the task, the execution time increases. However, in this OpenMP, the execution time gradually decreased and the performance improved, which can be attributed to the following reasons.

1. OpenMP performs parallelization tasks. As the number of threads increases, the work is broken down into smaller units to perform the work faster.
2. A certain amount of overhead occurs while creating a thread for parallelization. When the size of the task is large and the number of threads is large, the parallelization overhead is reduced and performance can be improved.
3. When processing tasks simultaneously with multiple threads, there are many situations in which data is loaded into the cache and reused. This can improve data access locality and reduce memory access cost to improve performance.
4. It may be that the current performance is good because the optimal number of threads has not been reached yet.

## Case II )cuda\_ray.cu

### (a) Execution environment (OS/CPU/GPU type or Colab?)

I used Colab. The OS is x86\_64 and the GPU is as shown in the picture below.



The screenshot shows the output of the `nvidia-smi` command in a Colab environment. The output is displayed in a terminal window with a title bar that says "Invidia-smi". The terminal shows the date and time as "Tue Jun 6 11:34:52 2023". The output of the command is as follows:

NVIDIA-SMI 525.85.12 Driver Version: 525.85.12 CUDA Version: 12.0									
GPU	Name	Persistence-M	Bus-Id	Disp.A	Volatile Uncorr. ECC				
Fan	Temp	Perf	Pwr:Usage/Cap	Memory-Usage	GPU-Util	Compute M.	MIG M.		
0	Tesla T4	Off	00000000:00:04.0	Off	0				
N/A	42C	P8	11W / 70W	0MiB / 15360MiB	0%	Default	N/A		

Processes:							
GPU	GI	CI	PID	Type	Process name	GPU Memory	Usage
ID	ID	ID					
No running processes found							

### (b) How to compile

Write “`!nvcc cuda_ray.cu -o cuda_ray.exe`” as code in Colab and press the start button.

### (c) How to execute

Write “`!./cuda_ray.exe`” as code in Colab and press the start button.

### (d) Entire source code with appropriate comments

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

#define SPHERES 20
#define DIM 2048

struct Sphere {
    float r, g, b; //color of sphere
    float radius; //radius of sphere
    float x, y, z; //coordinates of the center of the sphere
    __device__ float hit(float ox, float oy, float *n) { //determines if a given point intersects
a sphere and computes the depth at the point of intersection
        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 -INFINITY;
}
};

__global__ void kernel(Sphere *spheres, unsigned char *bitmap) { //A function that
performs ray tracing at (x,y)
    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 = -INFINITY;
    for (int i = 0; i < SPHERES; i++) {
        float n;
        float t = spheres[i].hit(ox, oy, &n);
        if (t > maxz) {
            float fscale = n;
            r = spheres[i].r * fscale;
            g = spheres[i].g * fscale;
            b = spheres[i].b * fscale;
            maxz = t;
        }
    }
    bitmap[offset * 4 + 0] = (int)(r * 255);
    bitmap[offset * 4 + 1] = (int)(g * 255);
    bitmap[offset * 4 + 2] = (int)(b * 255);
    bitmap[offset * 4 + 3] = 255;
}

void ppm_write(unsigned char *bitmap, int xdim, int ydim, const char *filename) {
//Function to save bitmap data as PPM image file
    FILE *fp = fopen(filename, "wb");
    fprintf(fp, "P6\n%d %d\n255\n", xdim, ydim);
    fwrite(bitmap, 1, xdim * ydim * 4, fp);
    fclose(fp);
}

int main() {
    Sphere *temp_s = (Sphere *)malloc(sizeof(Sphere) * SPHERES);
    for (int i = 0; i < SPHERES; i++) {
        temp_s[i].r = static_cast<float>(rand()) / RAND_MAX;
        temp_s[i].g = static_cast<float>(rand()) / RAND_MAX;
    }
}

```

```

        temp_s[i].b = static_cast<float>(rand()) / RAND_MAX;
        temp_s[i].x = static_cast<float>(rand() % 2000 - 1000);
        temp_s[i].y = static_cast<float>(rand() % 2000 - 1000);
        temp_s[i].z = static_cast<float>(rand() % 2000 - 1000);
        temp_s[i].radius = static_cast<float>(rand() % 160 + 40);
    }

    unsigned char *bitmap = (unsigned char *)malloc(sizeof(unsigned char) * DIM * DIM *
4);
    memset(bitmap, 0, sizeof(unsigned char) * DIM * DIM * 4);

    Sphere *d_spheres;
    unsigned char *d_bitmap;

    cudaMalloc((void **)&d_spheres, sizeof(Sphere) * SPHERES);
    cudaMalloc((void **)&d_bitmap, sizeof(unsigned char) * DIM * DIM * 4);

    cudaMemcpy(d_spheres, temp_s, sizeof(Sphere) * SPHERES, cudaMemcpyHostToDevice);

    dim3 dimGrid(DIM / 16, DIM / 16);
    dim3 dimBlock(16, 16);

    clock_t start = clock();

    kernel<<<dimGrid, dimBlock>>>(d_spheres, d_bitmap);

    cudaMemcpy(bitmap, d_bitmap, sizeof(unsigned char) * DIM * DIM * 4,
cudaMemcpyDeviceToHost);

    clock_t end = clock();
    double elapsed_sec = static_cast<double>(end - start) / CLOCKS_PER_SEC;

    printf("CUDA ray tracing: %.3f sec\n", elapsed_sec);
    ppm_write(bitmap, DIM, DIM, "result.ppm");
    printf("[result.ppm] was generated.\n"); //output

    cudaFree(d_spheres);
    cudaFree(d_bitmap);
    free(bitmap);
    free(temp_s);

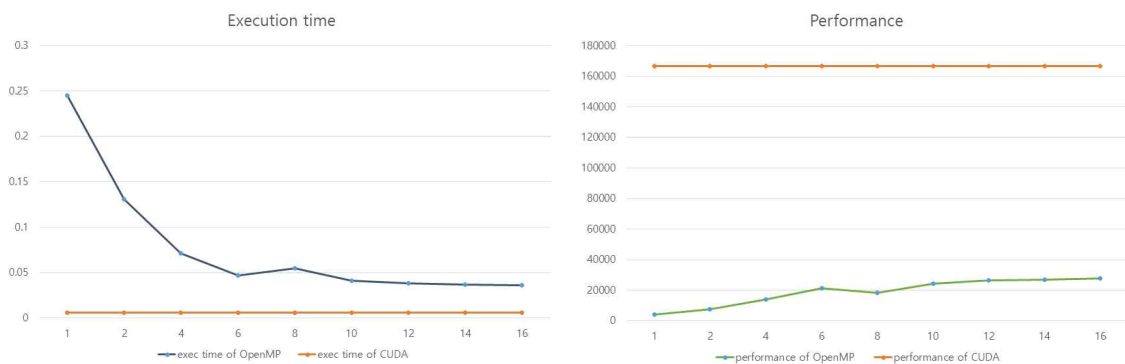
    return 0;
}

```

(e) Program output results and ray-tracing result pictures



(f) Experimental results : measuring the performance of OpenMP implementations vs CUDA implementation. Finally, Show the performance results with respect to various number of threads including graphs and interpretation/explanation.



As a result of the graph, CUDA runs much faster and the performance is better than OpenMP.

Ray-tracing requires extensive computation. CUDA is a technology designed for parallel processing using NVIDIA GPUs. Since it provides high performance by utilizing the GPU's multiple cores and parallel processing architecture, code using CUDA can perform significantly better.

Also, GPUs have a large number of cores, making them suitable for parallel tasks. CUDA can efficiently manage data by utilizing the GPU's memory hierarchy to reduce the overhead of data transfer and access.

In addition, it provides GPU-specific optimization techniques and has good performance because it can process large amounts of data quickly due to its high-bandwidth memory.