

Multicore Computing Project #4 – problem1



Department : Software
Name : Jeong eui chan
Student ID : 20195914

<Excution environment>

For openMP I used a MAC m1 laptop. The computer information of my MAC m1 is as shown in the picture below.

```
jeong-uichan ~ system_profiler SPHardwareDataType
Hardware:

Hardware Overview:

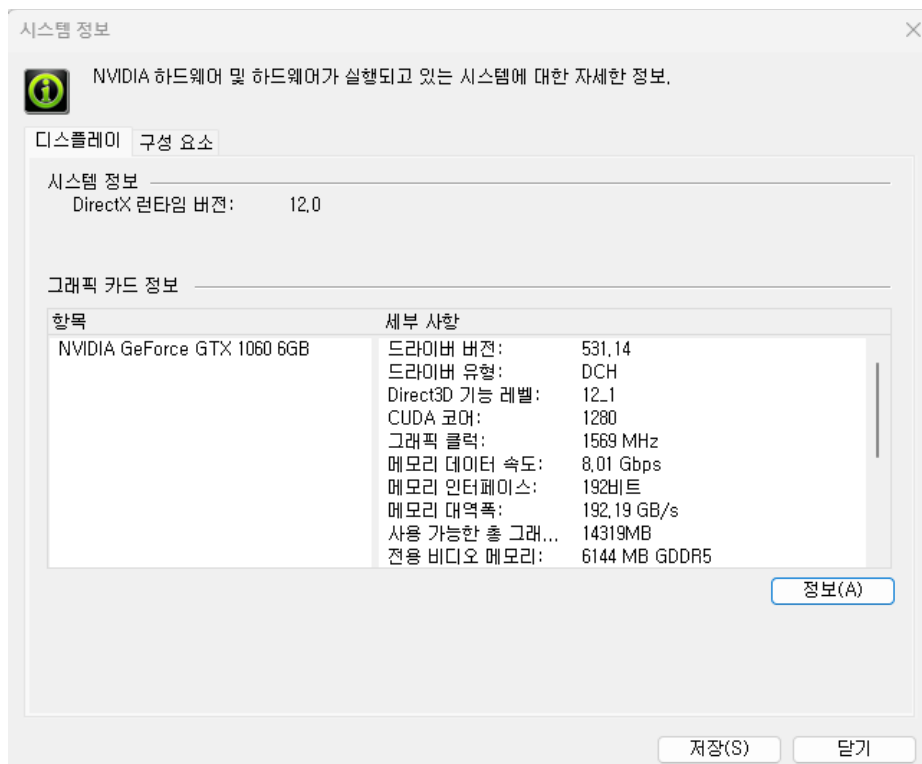
Model Name: MacBook Air
Model Identifier: MacBookAir10,1
Model Number: Z1250002VKH/A
Chip: Apple M1
Total Number of Cores: 8 (4 performance and 4 efficiency)
Memory: 16 GB
System Firmware Version: 8422.121.1
OS Loader Version: 8422.121.1
Serial Number (system): FVFFC3ADQ6LT
Hardware UUID: EE341825-7FB8-52C6-8F1A-E68A1656E926
Provisioning UDID: 00008103-001539513CEA001E
Activation Lock Status: Enabled
```

For CUDA, I used my desktop with an NVIDIA graphics card. The information on the desktop is like the picture below.

You can check the information of the type of OS and the CPU being used through the picture.



You can check the information of the NVIDIA graphics card used in my desktop, and you can check the number of CUDA cores.

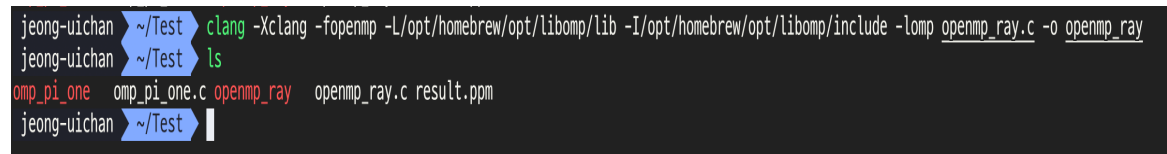


<How to compile – openmp_ray.c>

In the case of the openmp_ray.c file, enter the command as shown in the picture below in the mac m1 terminal

Compiled openmp_ray.c.

```
clang -Xclang -fopenmp -L/opt/homebrew/opt/libomp/lib -I/opt/homebrew/opt/libomp/include -lomp openmp_ray.c -o openmp_ray
```



```
jeong-uichan ~/Test clang -Xclang -fopenmp -L/opt/homebrew/opt/libomp/lib -I/opt/homebrew/opt/libomp/include -lomp openmp_ray.c -o openmp_ray
jeong-uichan ~/Test ls
omp_pi_one omp_pi_one.c openmp_ray openmp_ray.c result.ppm
jeong-uichan ~/Test
```

<How to execute – openmp_ray.c>

In the case of the openmp_ray.c file, I entered and executed the command as shown in the picture below in the mac m1 terminal.

By specifying the number of threads 1,2,4,10,12 We checked the difference in the result time according to the number of threads



```
jeong-uichan ~/Test ./openmp_ray 1
> OpenMP (1 threads) ray tracing: 0.667 sec
jeong-uichan ~/Test ./openmp_ray 2
> OpenMP (2 threads) ray tracing: 0.343 sec
jeong-uichan ~/Test ./openmp_ray 4
> OpenMP (4 threads) ray tracing: 0.200 sec
jeong-uichan ~/Test ./openmp_ray 6
> OpenMP (6 threads) ray tracing: 0.169 sec
jeong-uichan ~/Test ./openmp_ray 8
> OpenMP (8 threads) ray tracing: 0.158 sec
jeong-uichan ~/Test ./openmp_ray 10
> OpenMP (10 threads) ray tracing: 0.142 sec
jeong-uichan ~/Test ./openmp_ray 12
> OpenMP (12 threads) ray tracing: 0.140 sec
jeong-uichan ~/Test ./openmp_ray 14
> OpenMP (14 threads) ray tracing: 0.136 sec
jeong-uichan ~/Test ./openmp_ray 16
> OpenMP (16 threads) ray tracing: 0.141 sec
jeong-uichan ~/Test ./openmp_ray 32
> OpenMP (32 threads) ray tracing: 0.136 sec
jeong-uichan ~/Test ./openmp_ray 64
> OpenMP (64 threads) ray tracing: 0.127 sec
jeong-uichan ~/Test ./openmp_ray 128
> OpenMP (128 threads) ray tracing: 0.132 sec
jeong-uichan ~/Test ./openmp_ray 192
> OpenMP (192 threads) ray tracing: 0.125 sec
jeong-uichan ~/Test ./openmp_ray 200
> OpenMP (200 threads) ray tracing: 0.137 sec
```

<Entire source code - openmp_ray.c>

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

#define SPHERES 20 // 생성할 구의 수
#define rnd(x) (x * rand() / RAND_MAX) // 난수를 생성하기 위한 함수
#define INF 2e10f // 무한대를 나타내는 상수
#define DIM 2048 // 이미지의 가로, 세로 픽셀 수

// 구를 나타내는 구조체
struct Sphere {
    float r, b, g; // 구의 색상 (RGB)
    float radius; // 구의 반지름
    float x, y, z; // 구의 위치
};

// 구와의 충돌 여부와, 충돌지점의 거리를 반환하는 함수
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) { // 거리가 구의 반지름보다 작다면, 즉
구와 충돌한다면
        float dz = sqrtf(s.radius*s.radius - dx*dx - dy*dy);
        *n = dz / sqrtf(s.radius * s.radius); // 충돌 지점의 위치를 계산
        return dz + s.z; // 충돌 지점의 z 좌표 반환
    }
    return -INF; // 충돌하지 않는다면 -INF 반환
}

// 각 픽셀에 대한 색상을 계산하는 함수
void kernel(int x, int y, struct Sphere* s, unsigned char* ptr) {
    int offset = x + y * DIM; // 이미지 데이터에서의 현재 픽셀 위치
    float ox = (x - DIM / 2); // 광선의 x좌표
    float oy = (y - DIM / 2); // 광선의 y좌표

    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); // 광선과 구와의 충돌 여부 확인
        if (t > maxz) { // 만약 가장 가까운 구와 충돌했다면
            float fscale = n;
            r = s[i].r * fscale; // 색상의 r 요소 계산
            g = s[i].g * fscale; // 색상의 g 요소 계산
```

```

        b = s[i].b * fscale; // 색상의 b 요소 계산
        maxz = t;
    }
}

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

// PPM 형식의 이미지 파일을 작성하는 함수
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); // 이미지의 가로, 세로 크기를 파일에 기록
    fprintf(fp, "255\n"); // 색상의 최대값을 파일에 기록
    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]
); // 각 픽셀의 RGB값을 파일에 기록
        }
        fprintf(fp, "\n");
    }
}

int main(int argc, char* argv[]) {
    int no_threads; // 사용할 스레드의 개수
    int x, y;
    unsigned char* bitmap; // 이미지 데이터를 저장할 포인터

    srand(time(NULL)); // 난수 생성기 초기화

    // 명령줄 인자로 스레드의 개수가 제공되지 않았다면 프로그램 종료
    if (argc != 2) {
        printf("> openmp_ray.exe [number of threads]\n");
        exit(0);
    }

    // 명령줄 인자로 받은 스레드의 개수를 정수로 변환
    no_threads = atoi(argv[1]);

    // 구의 정보를 저장할 메모리를 할당하고, 각 구의 속성을 랜덤하게 설정
    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;
    }

    // 이미지 데이터를 저장할 메모리를 할당
    bitmap = (unsigned char*)malloc(sizeof(unsigned char) * DIM * DIM * 4);

    double start_time, end_time; // 측정을 위한 변수
    start_time = omp_get_wtime(); // 시간 측정 시작

    omp_set_num_threads(no_threads); // OpenMP에 사용할 스레드의 개수 설정
    #pragma omp parallel for private(y) // 병렬 처리 시작
    for (x = 0; x < DIM; x++)
        for (y = 0; y < DIM; y++)
            kernel(x, y, temp_s, bitmap); // 각 픽셀의 색상 계산

    end_time = omp_get_wtime(); // 시간 측정 종료
    printf("> OpenMP (%d threads) ray tracing: %.3f sec\n", no_threads, end_t
ime - start_time); // 결과 출력

    // PPM 형식의 이미지 파일 작성
    FILE* fp = fopen("result.ppm", "w");
    ppm_write(bitmap, DIM, DIM, fp);
    fclose(fp);

    // 할당했던 메모리 해제
    free(bitmap);
    free(temp_s);

    return 0;
}

```

<program output results – openmp_ray.c>

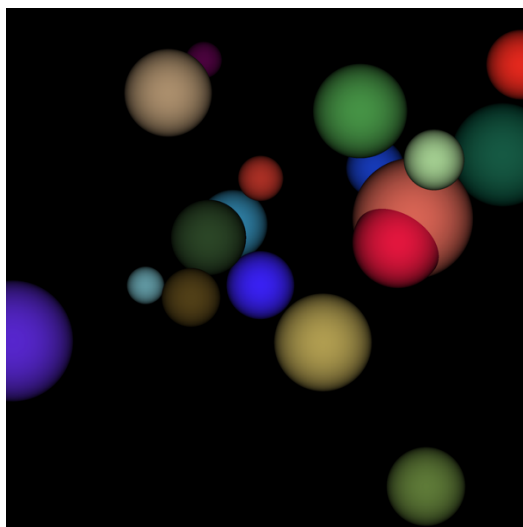
I ran the file multiple times by specifying the number of threads as 1 2 4 6 8 10 12 14 16 32 64 128 192 200. We were able to confirm the change in processing time according to the number of threads.

```
jeong-uichan ~/Test > ./openmp_ray 1
> OpenMP (1 threads) ray tracing: 0.667 sec
jeong-uichan ~/Test > ./openmp_ray 2
> OpenMP (2 threads) ray tracing: 0.343 sec
jeong-uichan ~/Test > ./openmp_ray 4
> OpenMP (4 threads) ray tracing: 0.200 sec
jeong-uichan ~/Test > ./openmp_ray 6
> OpenMP (6 threads) ray tracing: 0.169 sec
jeong-uichan ~/Test > ./openmp_ray 8
> OpenMP (8 threads) ray tracing: 0.158 sec
jeong-uichan ~/Test > ./openmp_ray 10
> OpenMP (10 threads) ray tracing: 0.142 sec
jeong-uichan ~/Test > ./openmp_ray 12
> OpenMP (12 threads) ray tracing: 0.140 sec
jeong-uichan ~/Test > ./openmp_ray 14
> OpenMP (14 threads) ray tracing: 0.136 sec
jeong-uichan ~/Test > ./openmp_ray 16
> OpenMP (16 threads) ray tracing: 0.141 sec
jeong-uichan ~/Test > ./openmp_ray 32
> OpenMP (32 threads) ray tracing: 0.136 sec
jeong-uichan ~/Test > ./openmp_ray 64
> OpenMP (64 threads) ray tracing: 0.127 sec
jeong-uichan ~/Test > ./openmp_ray 128
> OpenMP (128 threads) ray tracing: 0.132 sec
jeong-uichan ~/Test > ./openmp_ray 192
> OpenMP (192 threads) ray tracing: 0.125 sec
jeong-uichan ~/Test > ./openmp_ray 200
> OpenMP (200 threads) ray tracing: 0.137 sec
```

<ray-tracing result pictures - openmp_ray.c >

While executing the code, you can check that the resulting picture is created, and if you check the image file, you can see that the sphere was created randomly.

The following picture is the first result picture.



<experimental results - openmp_ray.c>

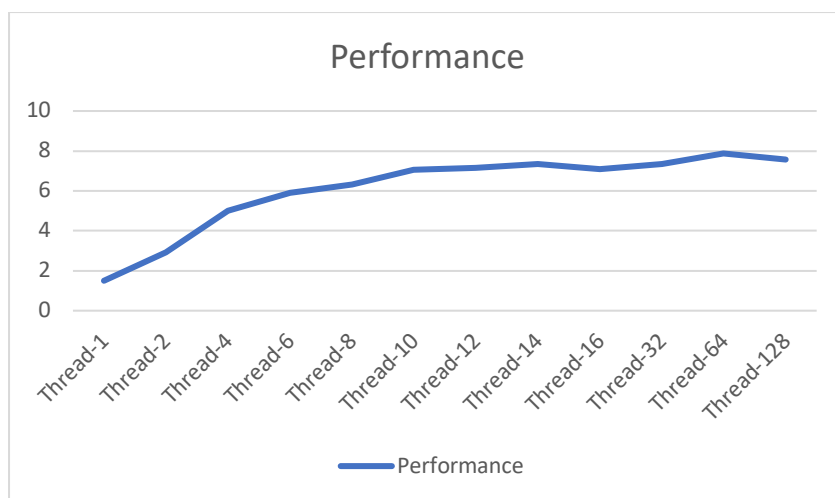
Exec Time

ThreadNum	1	2	4	6	8	10	12	14	16	32	64	128
Exec Time	0.667	0.343	0.200	0.169	0.158	0.142	0.140	0.136	0.141	0.136	0.127	0.132

Performance Time

ThreadNum	1	2	4	6	8	10	12	14	16	32	64	128
Exec Time	1.499	2.915	5	5.917	6.329	7.042	7.143	7.353	7.092	7.353	7.874	7.576

Performance Graph



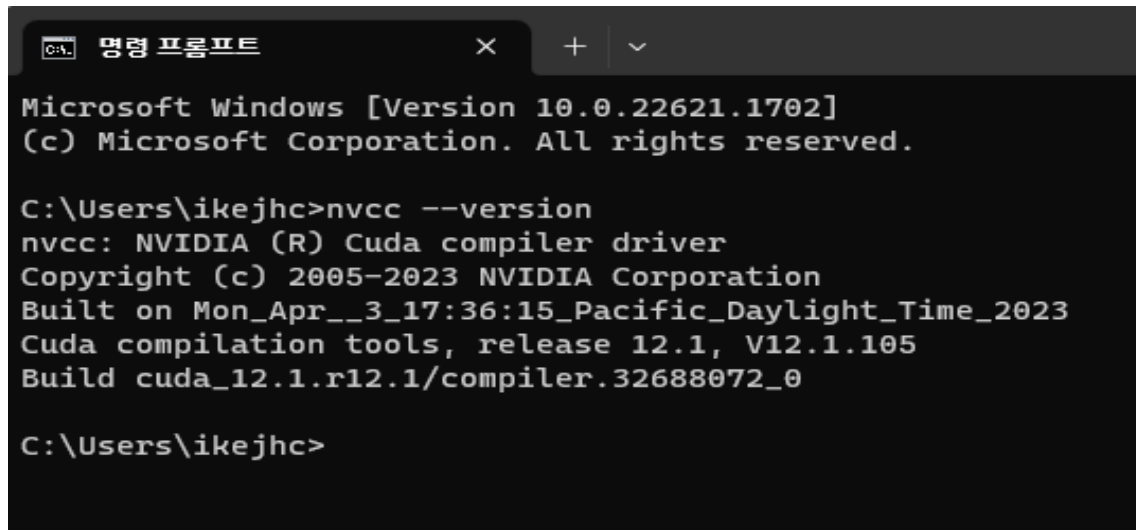
These results suggest that the program effectively halves its computation time each time the number of threads is doubled. The reason the performance improves as the number of threads increases in this code is because it leverages parallelism. This is because the work performed by the programs can be efficiently distributed across multiple processing cores without interfering with each other. In ray tracing, each ray can be computed independently from the others. Thus, increasing the number of threads disperses the workload over more processing units, allowing for more concurrent computations.

However, it's noticeable that this scalability has its limits. It can be observed that when the number of threads exceeds the number of physical cores in the CPU, there is no further improvement, but rather, there can be a performance degradation due to overhead.

Exceeding the number of physical cores in the CPU leads the operating system to perform context switching among the threads, which incurs additional overhead and can lead to performance degradation.

<How to compile – cuda_ray.cu>

In the case of the cuda_ray.cu file, compilation was performed on a Windows desktop equipped with an NVIDIA graphics card, and after installing nvcc, compilation was performed in Visual Studio 2022. As shown in the picture below, you can confirm that nvcc is installed.



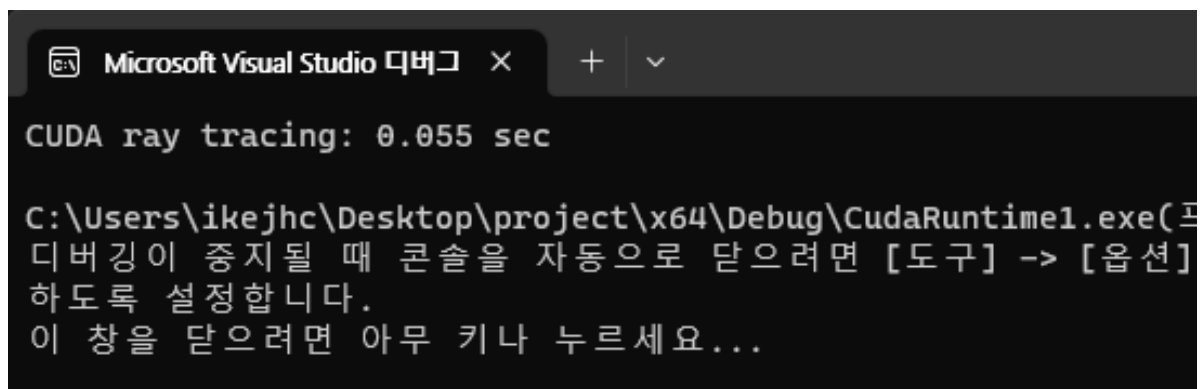
```
Microsoft Windows [Version 10.0.22621.1702]
(c) Microsoft Corporation. All rights reserved.

C:\Users\ikejhc>nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2023 NVIDIA Corporation
Built on Mon_Apr__3_17:36:15_Pacific_Daylight_Time_2023
Cuda compilation tools, release 12.1, V12.1.105
Build cuda_12.1.r12.1/compiler.32688072_0

C:\Users\ikejhc>
```

<How to execute – cuda_ray.cu>

In the case of the cuda_ray.cu file, it was executed using Visual Studio on a Windows desktop equipped with an NVIDIA graphics card, and you can see the execution result as shown in the picture below.



```
Microsoft Visual Studio 디버그
CUDA ray tracing: 0.055 sec

C:\Users\ikejhc\Desktop\project\x64\Debug\CudaRuntime1.exe(프
디버깅이 중지될 때 콘솔을 자동으로 닫으려면 [도구] -> [옵션]
하도록 설정합니다.
이 창을 닫으려면 아무 키나 누르세요...
```

<Entire source code – cuda_ray.cu>

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

#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;
};

// 레이가 구체에 충돌하는지 확인하고, 충돌하면 그 위치의 깊이(z값)과 표면의 단위 벡터를
// 반환하는 함수
__device__ float hit(Sphere s, float ox, float oy, float* n) {
    // 수평(x,y) 거리 계산
    float dx = ox - s.x;
    float dy = oy - s.y;
    // 레이가 구체에 충돌하는지 확인
    if (dx * dx + dy * dy < s.radius * s.radius) {
        // 충돌한 경우, z 값을 계산하고, 단위 벡터를 반환
        float dz = sqrtf(s.radius * s.radius - dx * dx - dy * dy);
        *n = dz / sqrtf(s.radius * s.radius);
        return dz + s.z;
    }
    // 충돌하지 않은 경우, -INF 반환
    return -INF;
}

__global__ void kernel(Sphere* s, unsigned char* ptr) {
    // 각 스레드에 대해 연산할 픽셀의 위치를 계산
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    // 픽셀의 오프셋 계산
    int offset = x + y * blockDim.x * gridDim.x;
    // 픽셀의 중심 위치 계산
    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");
    }
}

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

    srand(time(NULL)); // 랜덤 시드를 초기화

    Sphere* temp_s = (Sphere*)malloc(sizeof(Sphere) * SPHERES); // 구체 구조체
    를 위한 메모리 할당

    Sphere* dev_temp_s; // 디바이스(즉, GPU)에서 사용될 구체 구조체 포인터
    cudaMalloc((void**)&dev_temp_s, sizeof(Sphere) * SPHERES); // 디바이스에
    서 사용할 메모리를 할당

    // 모든 구체에 대해 랜덤한 속성을 설정
    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);
    // 비트맵 이미지를 위한 메모리 할당

    unsigned char* dev_bitmap; // 디바이스에서 사용될 비트맵 이미지 포인터
    cudaMalloc((void**)&dev_bitmap, sizeof(unsigned char) * DIM * DIM * 4);
    // 디바이스에서 사용할 메모리를 할당

```

```

    cudaMemcpy(dev_temp_s, temp_s, sizeof(Sphere) * SPHERES, cudaMemcpyHostToDevice); // 호스트(즉, CPU)에서 디바이스로 구체 정보를 복사

    dim3 grids(DIM / 16, DIM / 16); // 그리드의 크기 설정
    dim3 threads(16, 16); // 블록당 스레드의 수 설정

    cudaEvent_t start, stop; // 이벤트 변수 선언 (시간 측정을 위해 사용됨)
    cudaEventCreate(&start); // 시작 이벤트 생성
    cudaEventCreate(&stop); // 종료 이벤트 생성
    cudaEventRecord(start, 0); // 레이 트레이싱 시작 시간 기록

    kernel << <grids, threads >> > (dev_temp_s, dev_bitmap); // CUDA 커널 호출

    cudaEventRecord(stop, 0); // 레이 트레이싱 종료 시간 기록
    cudaEventSynchronize(stop); // 모든 CUDA 연산이 종료될 때까지 대기
    float elapsedTime; // 경과 시간을 저장할 변수
    cudaEventElapsedTime(&elapsedTime, start, stop); // 시작 이벤트와 종료 이벤트 사이의 시간 측정

    printf("CUDA ray tracing: %.3f sec\n", elapsedTime / 1000.0f); // 측정된 시간을 출력

    cudaEventDestroy(start); // 이벤트 파괴
    cudaEventDestroy(stop); // 이벤트 파괴

    cudaMemcpy(bitmap, dev_bitmap, sizeof(unsigned char) * DIM * DIM * 4, cudaMemcpyDeviceToHost); // 디바이스에서 호스트로 비트맵 이미지 복사

    FILE* fp = fopen("result_CUDA.ppm", "w");
    ppm_write(bitmap, DIM, DIM, fp);

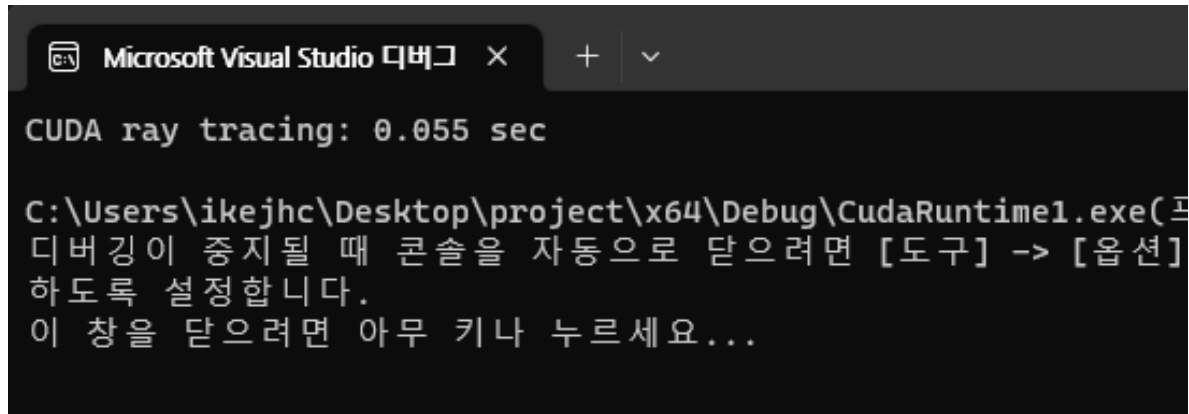
    fclose(fp);
    free(bitmap);
    free(temp_s);
    cudaFree(dev_temp_s); // 디바이스에서 할당한 메모리 해제
    cudaFree(dev_bitmap); // 디바이스에서 할당한 메모리 해제

    return 0;
}

```

<program output results – cuda_ray.cu>

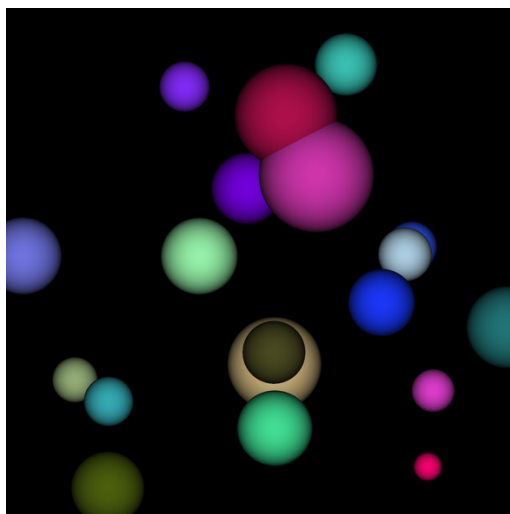
The following picture is the result of running through Visual Studio 2022.

A screenshot of the Visual Studio 2022 debug console. The title bar shows 'Microsoft Visual Studio 디버그' with a close button. The console output displays 'CUDA ray tracing: 0.055 sec' followed by a Korean message: 'C:\Users\ikejhc\Desktop\project\x64\Debug\CudaRuntime1.exe(프로젝트 디버깅이 중지될 때 콘솔을 자동으로 닫으려면 [도구] -> [옵션] 하도록 설정합니다. 이 창을 닫으려면 아무 키나 누르세요...'.

```
Microsoft Visual Studio 디버그 × + v
CUDA ray tracing: 0.055 sec
C:\Users\ikejhc\Desktop\project\x64\Debug\CudaRuntime1.exe(프로젝트 디버깅이 중지될 때 콘솔을 자동으로 닫으려면 [도구] -> [옵션] 하도록 설정합니다.
이 창을 닫으려면 아무 키나 누르세요...
```

<ray-tracing result pictures - cuda_ray.cu >

While executing the code, you can check that the resulting picture is created, and if you check the image file, you can see that the sphere was created randomly.



<experimental result>

It's evident that the code written in CUDA demonstrates faster performance. The reason is that GPUs execute thousands of threads concurrently, leveraging massive parallelism. CPUs are more suitable for tasks that require fewer cores and single-threaded performance. However, GPUs excel in the opposite scenario. That's why GPUs with CUDA outperform CPUs when tasks can be divided and run in parallel, such as matrix operations, image processing, and machine learning.