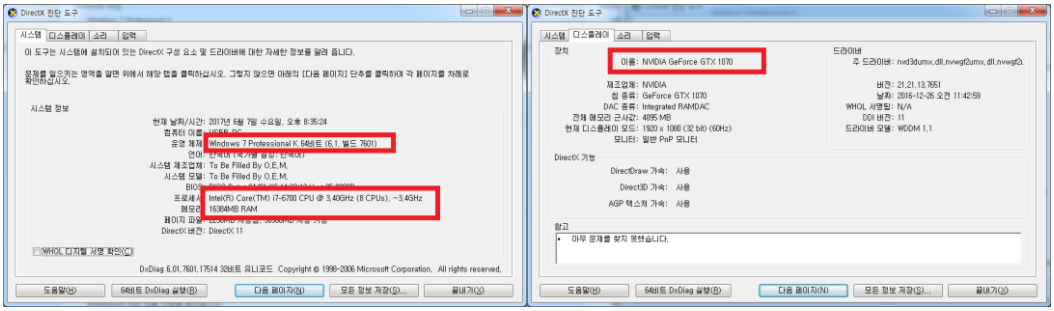
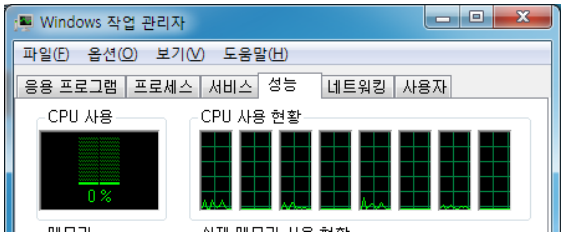
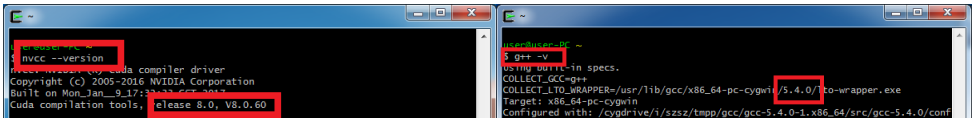
CUDA/OpenMP Ray-Tracing

1. Experiment Environment







1. OS Type: Windows 7 Professional K
2. CPU Type: Intel Core i7-6700
3. GPU Type: NVIDIA GeForce GTX 1070
4. RAM: 16GB
5. Clock Speed: 3.4 GHz
6. Core#: Octa-core
7. nvcc version: 8.0.6
8. g++ version: 5.4.0

2. How to Compile & Execute

Linux 또는 Cygwin에서 실행할 수 있습니다.

1. openmp_ray.c

- Compile: g++ openmp_ray.cpp -o openmp_ray -fopenmp
- Execute: ./openmp_ray.exe [thread_num] [result_file_name.ppm]

  Ex) ./openmp_ray.exe 8 result8.ppm

2. cuda_ray.cu

- CUDA 실행을 위해 먼저 실행 OS환경에 해당하는 CUDA를 다운받아야 한다. 설치 전에는 Visual Studio 2015 이하 버전이 설치되어 있어야 한다.
  → https://developer.nvidia.com/cuda-downloads
  "C:\ProgramData\NVIDIA Corporation\CUDA Samples\v.8.0.6\0_Simple\matrixMul"에서 사용하는 Visual Studio 버전에 맞는 Solution파일을 실행한다. 프로젝트를 Build하여 Setting을 완료한다.
- Compile: nvcc cuda_ray.cu -o cuda_ray

  'nvcc fatal : Cannot find compiler 'cl.exe' in PATH' 경고가 뜨는 경우, 환경 변수에 cl.exe의 위치 (C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\bin)를 추가하거나 컴파일 할 때 다음과 같은 명령어를 사용합니다.

  nvcc cuda_ray.cu -o cuda_ray -ccbin "C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\bin"
- Execute: ./cuda_ray.exe [result_file_name.ppm]

  Ex) ./cuda_ray.exe result.ppm

3. Source Code and Explanation

(기술한 것 외의 것들은 주석으로 상세히 설명해두었습니다)

1. openmp_ray.c

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

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;
		}
	}
	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 no_threads;
	int option;
	int x,y;
	unsigned char* bitmap;
	double timeDiff = 0.0, start_time, end_time;

	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]);
	}

	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;
	}

	omp_set_num_threads(no_threads);
	start_time = omp_get_wtime();

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

	#pragma omp parallel for schedule(guided, 2048)
	for (x=0;x<DIM;x++)
		for (y=0;y<DIM;y++)
			kernel(x,y,temp_s,bitmap);
```

```
        end_time = omp_get_wtime();
        timeDiff = (end_time - start_time);
        printf("OpenMP (%d threads) ray tracing: %lf sec \n", no_threads, timeDiff);

        ppm_write(bitmap,DIM,DIM,fp);
        printf("[%s] was generated. \n", argv[2]);

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

        return 0;
}
```

기존 raytracing.cpp코드에서 바뀐 부분은 파란색으로 표시했다.

실행 매개변수로 받은 스레드 수를 omp_set_num_threads를 통해 설정한다.

반복문 맨 위에 #pragma omp parallel for schedule 명령어를 써서 스케줄링을 할 부분을 kernel 함수로 지정했다. 어떤 부분은 sphere가 많이 모여서 오래 걸리고, 어떤 부분은 하나도 없어서 금방 끝날 것을 고려하여 guided 스케줄 타입으로 정하였고, 2048*2048 크기의 그림임을 고려하여 chunk size는 2048로 정했다.

2. cuda_ray.cu

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <math.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;
        // delete hit function

};

/* Added __global__ variable to run kernel function in GPU */
__global__ void kernel(const Sphere* s, unsigned char* ptr)
{
        /* Each kernel function uses thread/block index and block dimension to determine a unique number to process a particular pixel (x, y) */
        int x = threadIdx.x + blockIdx.x * blockDim.x;
        int y = threadIdx.y + blockIdx.y * blockDim.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;

                /* Moved the 'hit' function from the 'Sphere' structure of the existing code */
                float dx = ox - s[i].x;
                float dy = oy - s[i].y;

                if (dx*dx + dy*dy < s[i].radius*s[i].radius) {
                        float dz = sqrtf(s[i].radius*s[i].radius - dx*dx - dy*dy);
                        n = dz / sqrtf(s[i].radius * s[i].radius);
                        t = dz + s[i].z;
                }
                else t = -INF;

                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;
        cudaEvent_t start, stop;   // for time measurement
        float timeDiff;
```

```
	/* time variables event create */
	cudaEventCreate(&start);
	cudaEventCreate(&stop);
	srand(time(NULL));

	if (argc != 2) {
		printf("> a.out [filename.ppm]\n");
		exit(0);
	}
	FILE* fp = fopen(argv[1], "w");

	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);

	/* device_s and device_bitmap is to be assigned to device */
	Sphere *device_s;
	unsigned char* device_bitmap;

	/* Allocate space on GPU to copy the temp_s */
	cudaMalloc((void**)&device_s, sizeof(Sphere)*SPHERES);
	cudaMalloc((void**)&device_bitmap, sizeof(unsigned char)*DIM*DIM * 4);

	/* Copy temp_s to device_s to run the function in GPU */
	cudaMemcpy(device_s, temp_s, sizeof(Sphere)*SPHERES, cudaMemcpyHostToDevice);

	/* Start the recoding */
	cudaEventRecord(start, 0);

	/* 768 thread per block */
	dim3 dimBlock(32, 24);
	/* (Dimension/blockDimension) block per grid */
	dim3 dimGrid(DIM / dimBlock.x, DIM / dimBlock.y);

	kernel << <dimGrid, dimBlock >> > (device_s, device_bitmap);
	cudaDeviceSynchronize();

	/* End the recording */
	cudaEventRecord(stop, 0);
	cudaEventSynchronize(stop);

	/* Copy the result stored in the device back to the host */
	cudaMemcpy(bitmap, device_bitmap, sizeof(unsigned char)*DIM*DIM * 4,
cudaMemcpyDeviceToHost);

	/* Execution time checking */
	cudaEventElapsedTime(&timeDiff, start, stop);
	printf("CUDA ray tracing: %f sec \n", timeDiff / CLOCKS_PER_SEC);

	ppm_write(bitmap, DIM, DIM, fp);
```

```
        printf("[%s] was generated. \n", argv[1]);

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

        cudaFree(device_s);
        cudaFree(device_bitmap);

        return 0;
}
```

기존 raytracing.cpp코드에서 바뀐 부분은 파란색으로 표시했다.

CUDA를 쓰기 위해 헤더파일을 include했다.

kernel 함수를 device에서 쓰기 위해 __global__변수를 통해 설정했다.

kernel 함수의 매개변수로 넘어오던 x, y를 없애고 thread와 block의 index, dimension을 이용하여 고유 index를 함수 내부에서 정의하였다.

기존 코드에서 kernel함수 안에 쓰이던 hit함수를 Sphere 구조체에서 kernel 함수 안으로 옮겼다. (hit 함수는 host에 있고, kernel함수는 device에 있기 때문에 사용할 수가 없어서 옮겼다.)

시간 측정은 cudaEvent_t 변수를 사용해 측정하였다.

device에서 사용할 device_s Sphere와 device_bitmap을 정의하고 cudaMalloc을 통해 device에 메모리를 할당하였다. 그 후 device_s에 기존 temp_s 정보를 복사하고 kernel << <dimGrid, dimBlock >> >을 통해 함수를 실행하였다. cudaDeviceSynchronize를 걸어주고 계산한 결과값을 host의 bitmap변수에 복사했다. 실행 매개변수로 받은 스레드 수를 omp_set_num_threads를 통해 설정한다.

반복문 맨 위에 #pragma omp parallel for schedule 명령어를 써서 스케줄링을 할 부분을 kernel 함수로 지정했다. 어떤 부분은 sphere가 많이 모여서 오래 걸리고, 어떤 부분은 하나도 없어서 금방 끝날 것을 고려하여 guided 스케줄 타입으로 정하였고, 2048*2048 크기의 그림임을 고려하여 chunk size는 2048로 정했다.

4. Other implementation issues

1. Determine Block Size in CUDA

CUDA 프로그래밍을 하면서 가장 깊게 고민했던 것은 block size 설정이었다.

처음에는 하단의 표를 참고해서 사용하는 GPU의 최대 스레드 용량에 맞춰 block dimension을 32*32=1024로 설정했다.

Compute capability (version)	Micro-architecture	GPUs	GeForce
6.0	Pascal	GP100	
6.1		GP102, GP104, GP106, GP107, GP108	Nvidia TITAN Xp, Titan X, GeForce GTX 1080 Ti, GTX 1080, GTX 1070, GTX 1060, GTX 1050 Ti, GTX 1050, GT 1030

Technical specifications	Compute capability (version)														
	1.0	1.1	1.2	1.3	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2
Maximum dimensionality of grid of thread blocks	2				3										
Maximum x-dimension of a grid of thread blocks	65535				$2^{31}-1$										
Maximum y-, or z-dimension of a grid of thread blocks	65535														
Maximum dimensionality of thread block	3														
Maximum x- or y-dimension of a block	512				1024										
Maximum z-dimension of a block	64														
Maximum number of threads per block	512				1024										

```
dim3 dimBlock(32, 32);
dim3 dimGrid(DIM / dimBlock.x, DIM / dimBlock.y);
```

하지만 1개의 SM에 8개의 SP, 1개의 SP에 768개의 스레드가 32개씩 24개의 warp에 묶여 있음을 알게 되고 나서 다음과 같이 수정하였다.

```
dim3 dimBlock(32, 24);
dim3 dimGrid(DIM / dimBlock.x, DIM / dimBlock.y);
```

2. 'hit' function in struct Sphere in CUDA

Sphere 구조체 안에 있는 hit function을 kernel함수에서 사용하기 위해서는 코드 수정이 필요했다. Sphere를 device에 선언하면 main함수에서 temp_s선언을 못하고, hit만 __global__선언을 할 수가 없어서 어떻게 할지 고민했다.

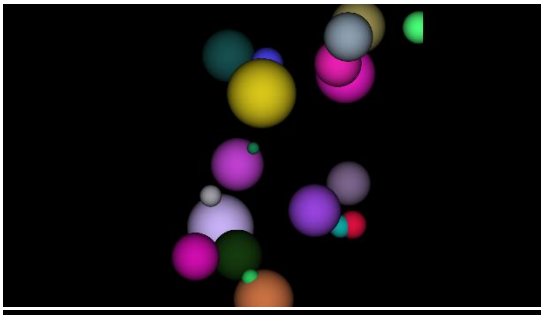
처음에는 hit함수를 __global__함수 안에서 호출하기 위해서 __device__ 선언을 하여 하나의 함수로 정의하였는데, 괜히 함수간 이동을 하면서 시간을 낭비하는 것보단 kernel함수 안에 바로 써주는 게 낫다고 판단하여 kernel함수 안에 바로 정의하였다.

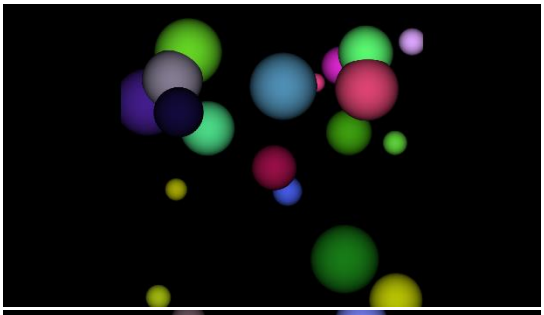
5. Results including Screen Shot

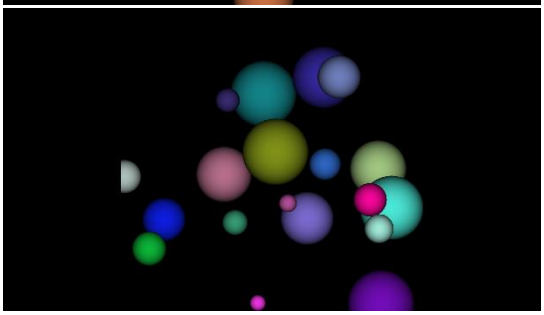
1. openmp_ray.c

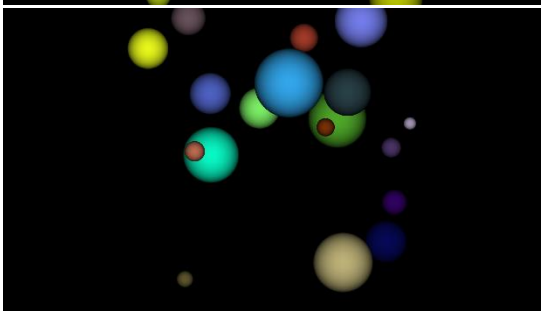
1. Thread num: 1

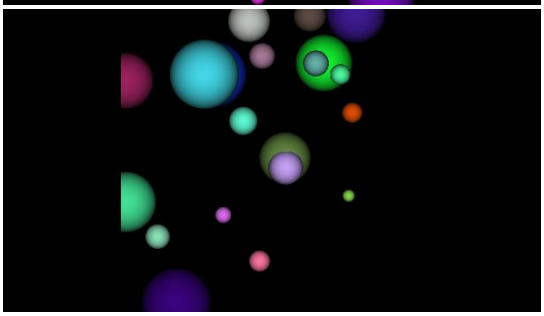
```
user@user-PC ~
$ ./openmp_ray.exe 1 01_openmp_01.ppm
OpenMP (1 threads) ray tracing: 0.675590 sec
[01_openmp_01.ppm] was generated.

user@user-PC ~
$ ./openmp_ray.exe 1 01_openmp_02.ppm
OpenMP (1 threads) ray tracing: 0.717781 sec
[01_openmp_02.ppm] was generated.

user@user-PC ~
$ ./openmp_ray.exe 1 01_openmp_03.ppm
OpenMP (1 threads) ray tracing: 0.745277 sec
[01_openmp_03.ppm] was generated.

user@user-PC ~
$ ./openmp_ray.exe 1 01_openmp_04.ppm
OpenMP (1 threads) ray tracing: 0.703028 sec
[01_openmp_04.ppm] was generated.

user@user-PC ~
$ ./openmp_ray.exe 1 01_openmp_05.ppm
OpenMP (1 threads) ray tracing: 0.665764 sec
[01_openmp_05.ppm] was generated.
```





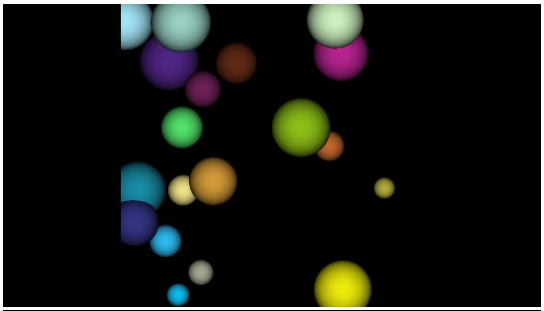
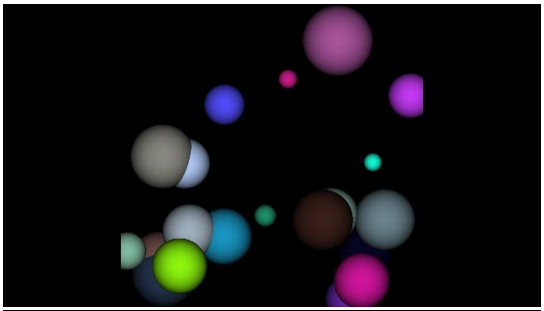
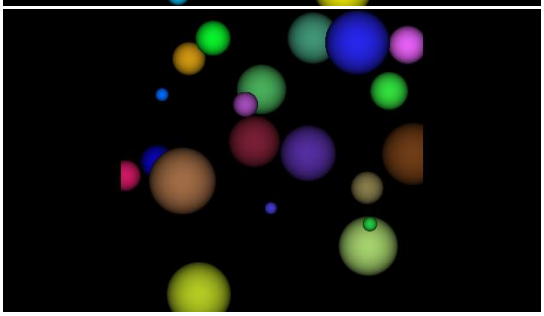
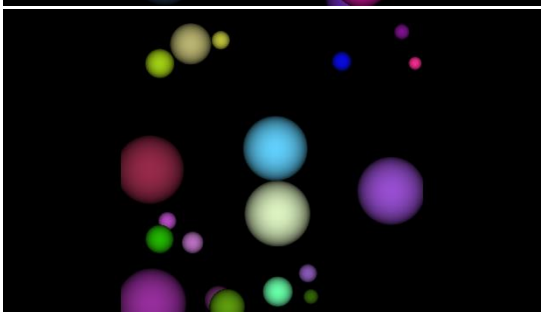
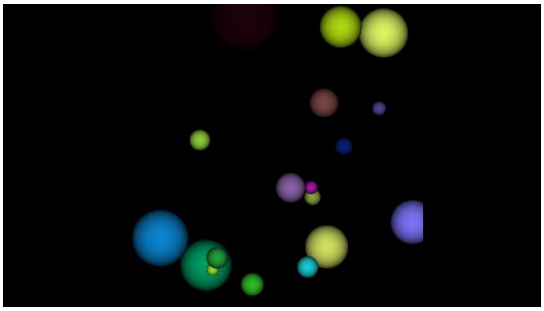






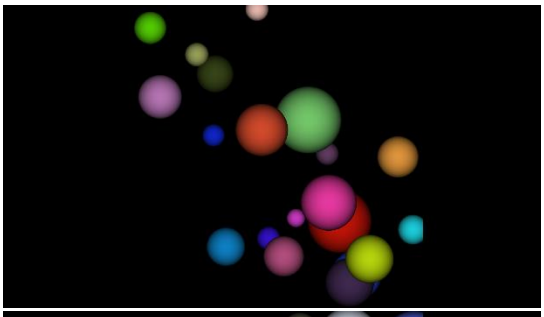
2. Thread num: 2

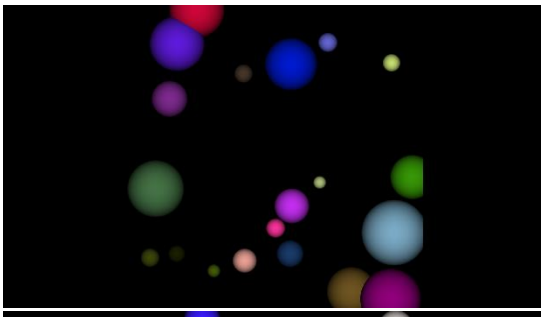
```
user@user-PC ~
$ ./openmp_ray.exe 2 02_openmp_01.ppm
OpenMP (2 threads) ray tracing: 0.682003 sec
[02_openmp_01.ppm] was generated.

user@user-PC ~
$ ./openmp_ray.exe 2 02_openmp_02.ppm
OpenMP (2 threads) ray tracing: 0.655602 sec
[02_openmp_02.ppm] was generated.

user@user-PC ~
$ ./openmp_ray.exe 2 02_openmp_03.ppm
OpenMP (2 threads) ray tracing: 0.707710 sec
[02_openmp_03.ppm] was generated.

user@user-PC ~
$ ./openmp_ray.exe 2 02_openmp_04.ppm
OpenMP (2 threads) ray tracing: 0.762912 sec
[02_openmp_04.ppm] was generated.

user@user-PC ~
$ ./openmp_ray.exe 2 02_openmp_05.ppm
OpenMP (2 threads) ray tracing: 0.752953 sec
[02_openmp_05.ppm] was generated.
```

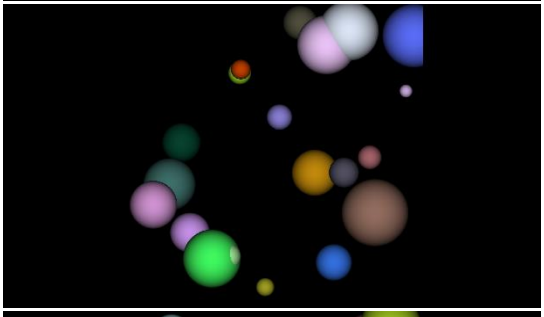






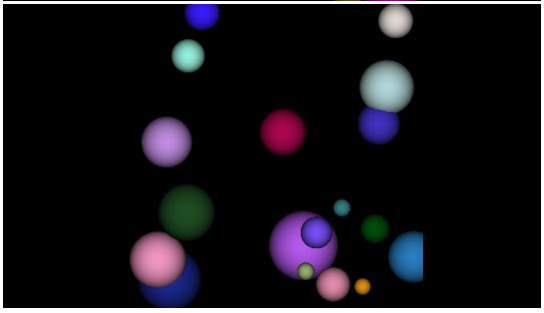
3. Thread num: 4

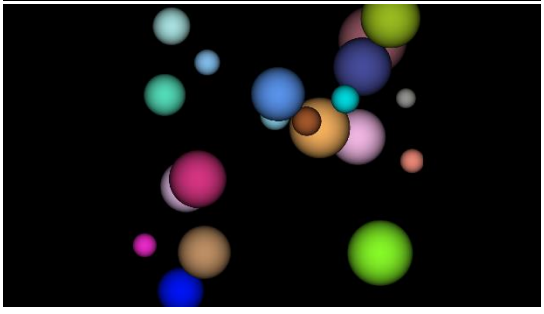
```
user@user-PC ~
$ ./openmp_ray.exe 2 02_openmp_01.ppm
OpenMP (2 threads) ray tracing: 0.682003 sec
[02_openmp_01.ppm] was generated.

user@user-PC ~
$ ./openmp_ray.exe 2 02_openmp_02.ppm
OpenMP (2 threads) ray tracing: 0.655602 sec
[02_openmp_02.ppm] was generated.

user@user-PC ~
$ ./openmp_ray.exe 2 02_openmp_03.ppm
OpenMP (2 threads) ray tracing: 0.707710 sec
[02_openmp_03.ppm] was generated.

user@user-PC ~
$ ./openmp_ray.exe 2 02_openmp_04.ppm
OpenMP (2 threads) ray tracing: 0.762912 sec
[02_openmp_04.ppm] was generated.

user@user-PC ~
$ ./openmp_ray.exe 2 02_openmp_05.ppm
OpenMP (2 threads) ray tracing: 0.752953 sec
[02_openmp_05.ppm] was generated.

user@user-PC ~
```





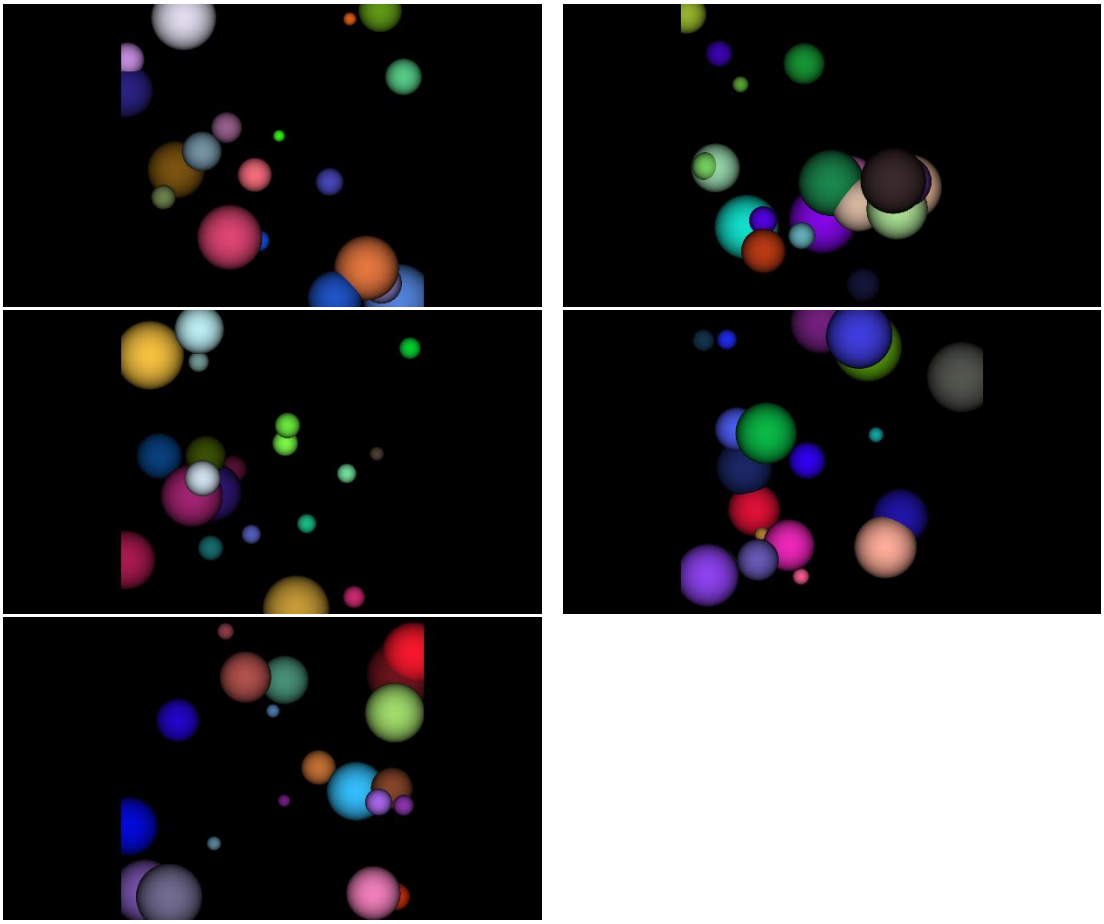






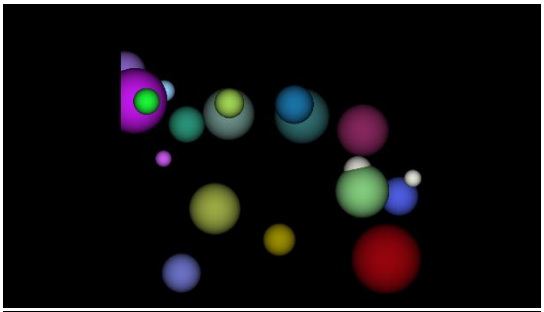
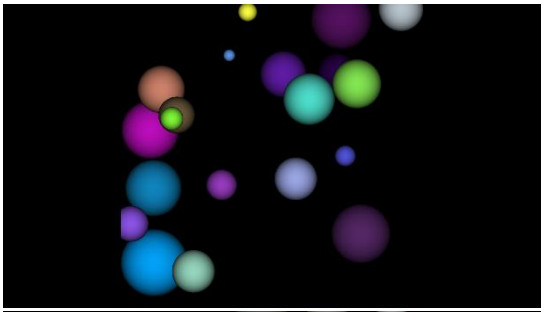
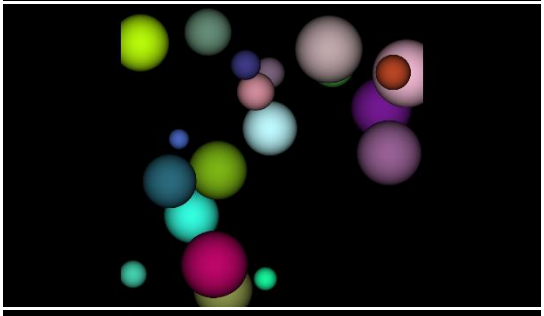
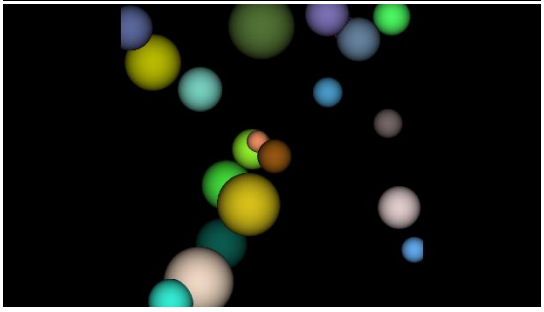
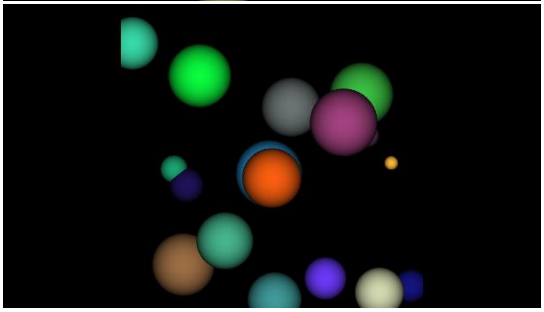
4. Thread num: 8

```
user@user-PC ~
$ ./openmp_ray.exe 8 08_openmp_01.ppm
OpenMP (8 threads) ray tracing: 0.745172 sec
[08_openmp_01.ppm] was generated.

user@user-PC ~
$ ./openmp_ray.exe 8 08_openmp_02.ppm
3OpenMP (8 threads) ray tracing: 0.686353 sec
[08_openmp_02.ppm] was generated.

user@user-PC ~
$ ./openmp_ray.exe 8 08_openmp_03.ppm
OpenMP (8 threads) ray tracing: 0.747182 sec
[08_openmp_03.ppm] was generated.

user@user-PC ~
$ ./openmp_ray.exe 8 08_openmp_04.ppm
OpenMP (8 threads) ray tracing: 0.717667 sec
[08_openmp_04.ppm] was generated.

user@user-PC ~
$ ./openmp_ray.exe 8 08_openmp_05.ppm
OpenMP (8 threads) ray tracing: 0.694808 sec
[08_openmp_05.ppm] was generated.

user@user-PC ~
```



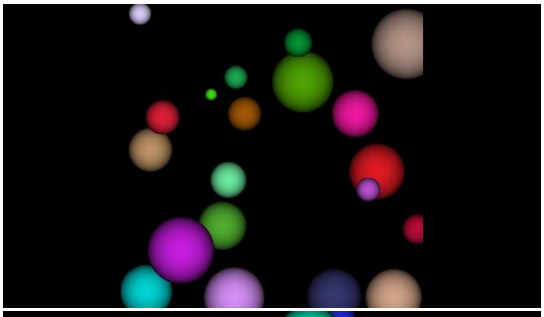
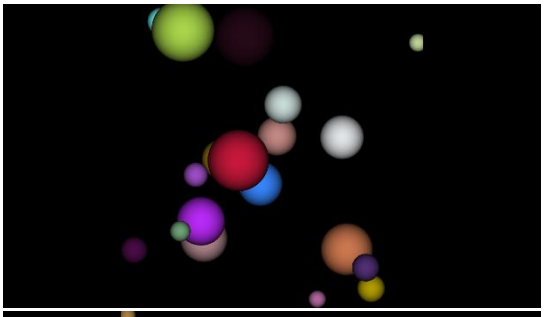
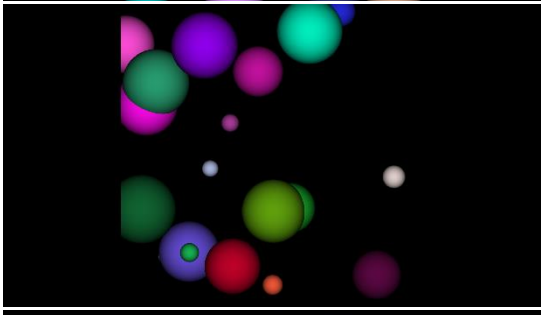
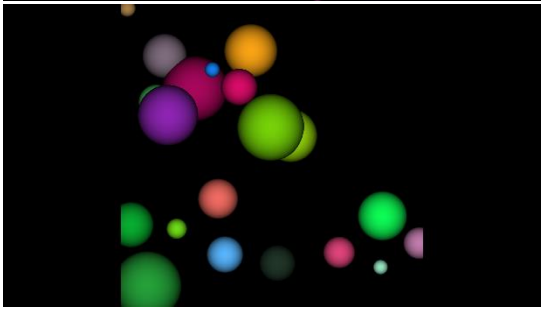
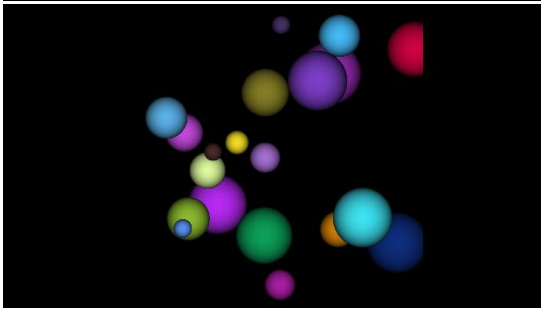
5. Thread num: 16

```
user@user-PC ~
$ ./openmp_ray.exe 16 16_openmp_01.ppm
OpenMP (16 threads) ray tracing: 0.705176 sec
[16_openmp_01.ppm] was generated.

user@user-PC ~
$ ./openmp_ray.exe 16 16_openmp_02.ppm
OpenMP (16 threads) ray tracing: 0.780748 sec
[16_openmp_02.ppm] was generated.

user@user-PC ~
$ ./openmp_ray.exe 16 16_openmp_03.ppm
OpenMP (16 threads) ray tracing: 0.734976 sec
[16_openmp_03.ppm] was generated.

user@user-PC ~
$ ./openmp_ray.exe 16 16_openmp_04.ppm
OpenMP (16 threads) ray tracing: 0.744655 sec
[16_openmp_04.ppm] was generated.

user@user-PC ~
$ ./openmp_ray.exe 16 16_openmp_05.ppm
OpenMP (16 threads) ray tracing: 0.726294 sec
[16_openmp_05.ppm] was generated.
```

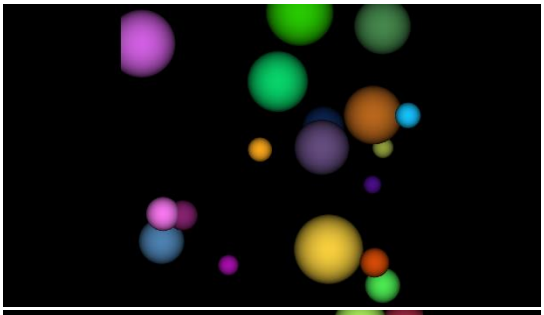
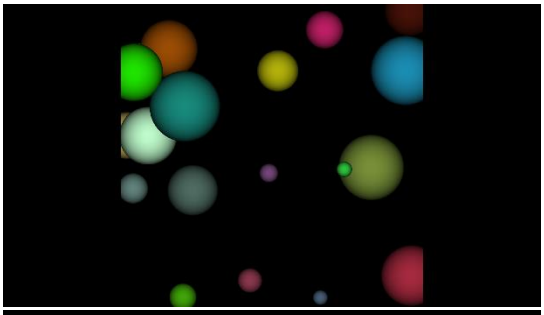
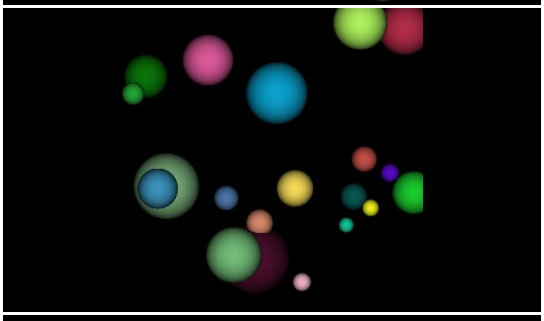
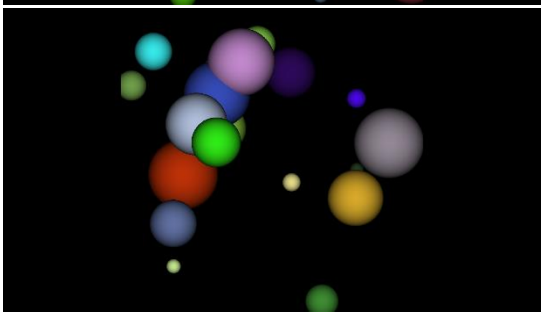
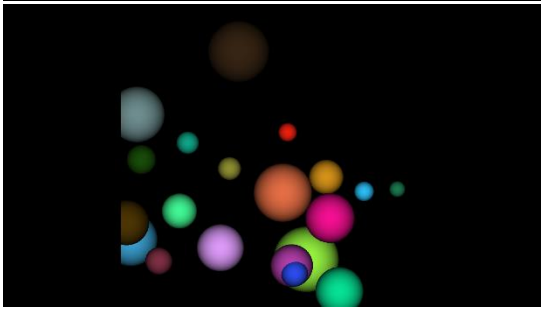
6. Thread num: 32

```
user@user-PC ~
$ ./openmp_ray.exe 32 32_openmp_01.ppm
OpenMP (32 threads) ray tracing: 0.690167 sec
[32_openmp_01.ppm] was generated.

user@user-PC ~
$ ./openmp_ray.exe 32 32_openmp_02.ppm
OpenMP (32 threads) ray tracing: 0.705519 sec
[32_openmp_02.ppm] was generated.

user@user-PC ~
$ ./openmp_ray.exe 32 32_openmp_03.ppm
OpenMP (32 threads) ray tracing: 0.715925 sec
[32_openmp_03.ppm] was generated.

user@user-PC ~
$ ./openmp_ray.exe 32 32_openmp_04.ppm
OpenMP (32 threads) ray tracing: 0.700543 sec
[32_openmp_04.ppm] was generated.

user@user-PC ~
$ ./openmp_ray.exe 32 32_openmp_05.ppm
OpenMP (32 threads) ray tracing: 0.705842 sec
[32_openmp_05.ppm] was generated.
```

7. Thread num: 64

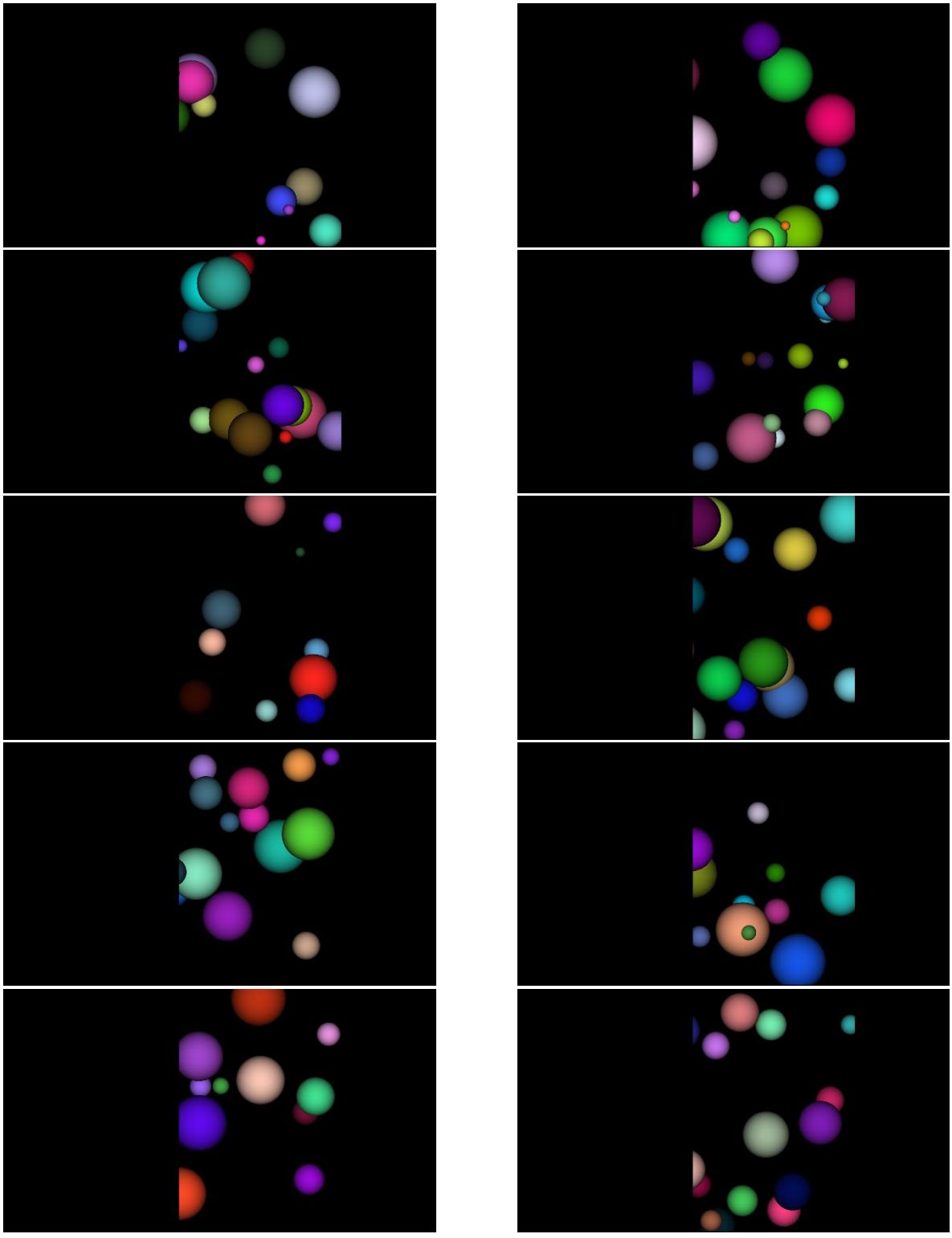
```
user@user-PC ~
$ ./openmp_ray.exe 64 64_openmp_01.ppm
OpenMP (64 threads) ray tracing: 0.695522 sec
[64_openmp_01.ppm] was generated.

user@user-PC ~
$ ./openmp_ray.exe 64 64_openmp_02.ppm
OpenMP (64 threads) ray tracing: 0.724169 sec
[64_openmp_02.ppm] was generated.

user@user-PC ~
$ ./openmp_ray.exe 64 64_openmp_03.ppm
OpenMP (64 threads) ray tracing: 0.723583 sec
[64_openmp_03.ppm] was generated.

user@user-PC ~
$ ./openmp_ray.exe 64 64_openmp_04.ppm
OpenMP (64 threads) ray tracing: 0.777130 sec
[64_openmp_04.ppm] was generated.

user@user-PC ~
$ ./openmp_ray.exe 64 64_openmp_05.ppm
OpenMP (64 threads) ray tracing: 0.707554 sec
[64_openmp_05.ppm] was generated.
```

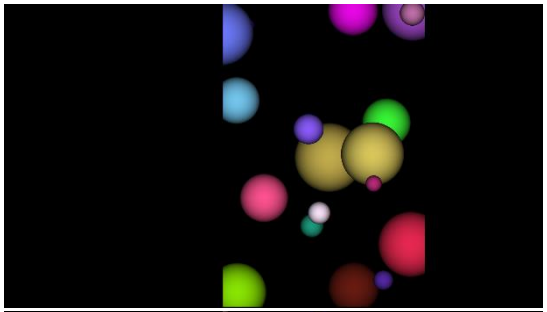
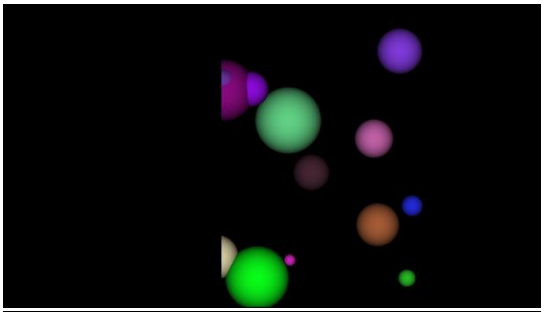
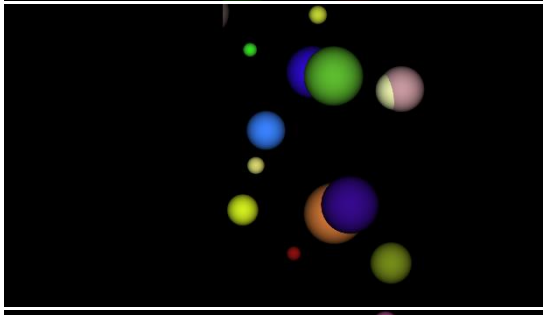
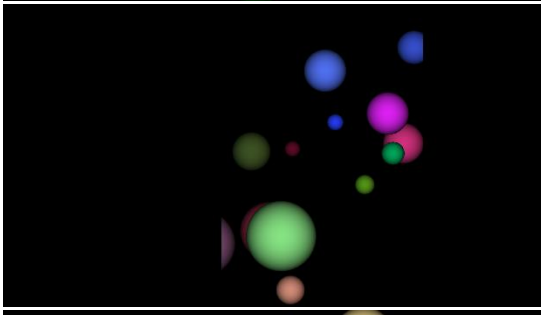
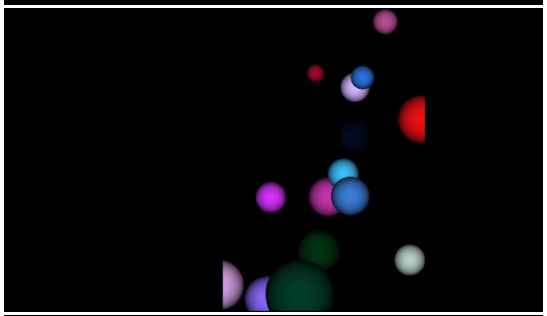
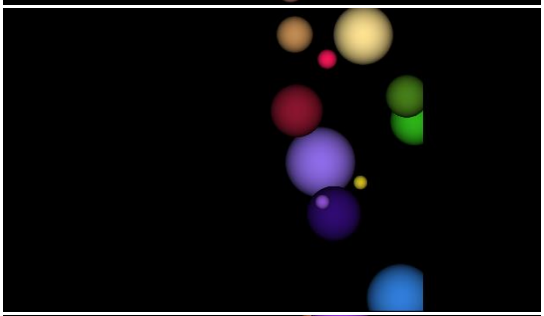
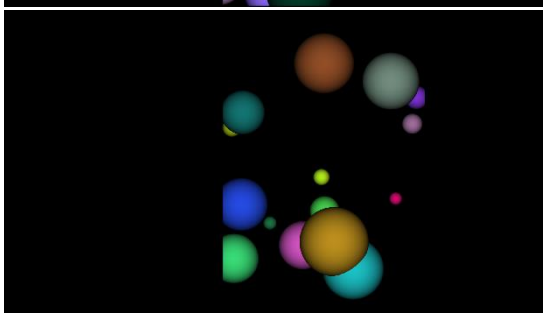
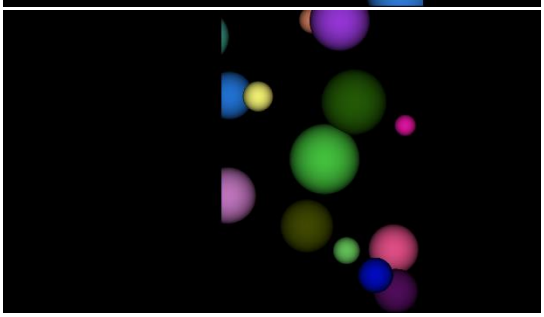
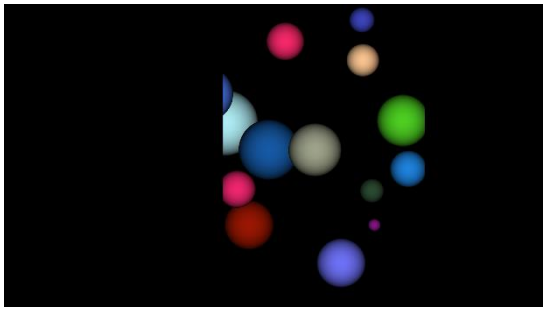
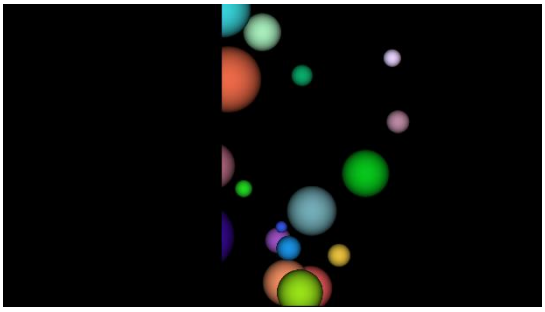






2. cuda_ray.cu

```
user@user-PC ~
$ ./cuda_ray.exe result_01.ppm
CUDA ray tracing: 0.003309 sec
[result_01.ppm] was generated.

user@user-PC ~
$ ./cuda_ray.exe result_02.ppm
CUDA ray tracing: 0.003314 sec
[result_02.ppm] was generated.

user@user-PC ~
$ ./cuda_ray.exe result_03.ppm
CUDA ray tracing: 0.003266 sec
[result_03.ppm] was generated.

user@user-PC ~
$ ./cuda_ray.exe result_04.ppm
CUDA ray tracing: 0.003293 sec
[result_04.ppm] was generated.

user@user-PC ~
$ ./cuda_ray.exe result_05.ppm
CUDA ray tracing: 0.003284 sec
[result_05.ppm] was generated.

user@user-PC ~
$ ./cuda_ray.exe result_06.ppm
CUDA ray tracing: 0.003290 sec
[result_06.ppm] was generated.

user@user-PC ~
$ ./cuda_ray.exe result_07.ppm
CUDA ray tracing: 0.003303 sec
[result_07.ppm] was generated.

user@user-PC ~
$ ./cuda_ray.exe result_08.ppm
CUDA ray tracing: 0.003318 sec
[result_08.ppm] was generated.

user@user-PC ~
$ ./cuda_ray.exe result_09.ppm
CUDA ray tracing: 0.003319 sec
[result_09.ppm] was generated.

user@user-PC ~
$ ./cuda_ray.exe result_10.ppm
CUDA ray tracing: 0.003288 sec
[result_10.ppm] was generated.
```

```
user@user-PC ~
$ ./cuda_ray.exe result_11.ppm
CUDA ray tracing: 0.003295 sec
[result_11.ppm] was generated.

user@user-PC ~
$ ./cuda_ray.exe result_12.ppm
CUDA ray tracing: 0.003278 sec
[result_12.ppm] was generated.

user@user-PC ~
$ ./cuda_ray.exe result_13.ppm
CUDA ray tracing: 0.003270 sec
[result_13.ppm] was generated.

user@user-PC ~
$ ./cuda_ray.exe result_14.ppm
CUDA ray tracing: 0.003283 sec
[result_14.ppm] was generated.

user@user-PC ~
$ ./cuda_ray.exe result_15.ppm
CUDA ray tracing: 0.003300 sec
[result_15.ppm] was generated.

user@user-PC ~
$ ./cuda_ray.exe result_16.ppm
CUDA ray tracing: 0.003262 sec
[result_16.ppm] was generated.

user@user-PC ~
$ ./cuda_ray.exe result_17.ppm
CUDA ray tracing: 0.003273 sec
[result_17.ppm] was generated.

user@user-PC ~
$ ./cuda_ray.exe result_18.ppm
CUDA ray tracing: 0.003289 sec
[result_18.ppm] was generated.

user@user-PC ~
$ ./cuda_ray.exe result_19.ppm
CUDA ray tracing: 0.003270 sec
[result_19.ppm] was generated.

user@user-PC ~
$ ./cuda_ray.exe result_20.ppm
CUDA ray tracing: 0.003256 sec
[result_20.ppm] was generated.
```



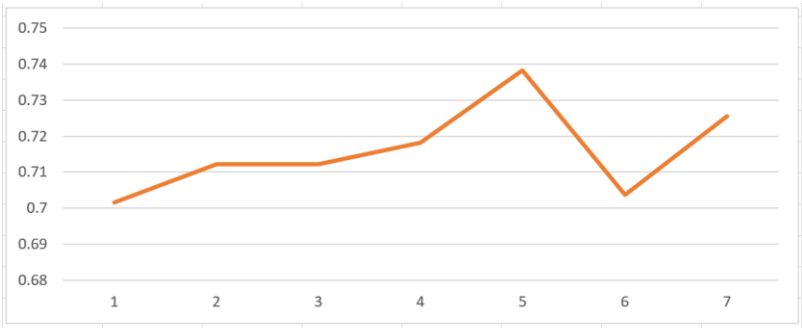
6. Experimental Results

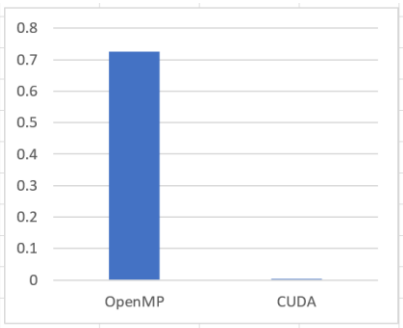
1. Performance (= execution time)

위의 result에서 실행 시간을 확인할 수 있다.

OpenMP는 각 스레드 개수마다 5번씩을 실행하였고, CUDA는 20번을 실행하였다.

각 실행시간 평균으로 성능을 비교해보면 아래와 같다.





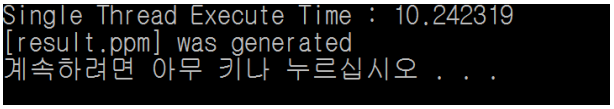
Thread#	1	2	4	8	16	32	64	CUDA1	CUDA2	CUDA3	CUDA4
	0.67559	0.682003	0.682003	0.745172	0.705176	0.690167	0.695522	0.003309	0.00329	0.003295	0.003262
	0.717781	0.655602	0.655602	0.686353	0.780748	0.705519	0.724169	0.003314	0.003303	0.003278	0.003273
	0.745277	0.70771	0.70771	0.747182	0.734976	0.715925	0.723583	0.003266	0.003318	0.00327	0.003289
	0.703028	0.762912	0.762912	0.717667	0.744655	0.700543	0.77713	0.003293	0.003319	0.003283	0.00327
	0.665764	0.752953	0.752953	0.694808	0.726294	0.705842	0.707554	0.003284	0.003288	0.0033	0.003256
평균	0.701488	0.712236	0.712236	0.718236	0.73837	0.703599	0.725592	0.003293	0.003304	0.003285	0.00327

왼쪽 그래프는 스레드 수가 적을수록 왼쪽에 위치한다. y축은 성능이고, 스레드가 32개일 때 잠시 감소했다가 다시 증가하는 추세이다.

오른쪽 그래프는 OpenMP와 CUDA의 실행 시간을 비교한 것이다. CUDA가 OpenMP에 비해 200배 정도 실행이 빠르게 됨을 확인할 수 있다.

7. Conclusion

OpenMP 없이 Single Thread로 실행했을 때는 다음과 같다.



앞서 진행한 프로젝트에서 OpenMP나 pthread를 사용해보았고, 이번 프로젝트에서 CUDA까지 해서 총 3개의 멀티코어 프로그래밍을 진행해보고 그 성능을 비교해보았다.

특히 이번 CUDA 프로그래밍을 통해 멀티코어를 넘어서 매니 코어 프로그래밍의 중요성을 느끼게 되었다. 멀티코어는 스레드를 많아야 100개정도 사용하는데, 매니 코어는 기본적으로 몇 천개의 스레드가 존재하기 때문이다. 처음에는 CUDA 프로그래밍을 위해 host → device로 값을 복사하고, 처리해서 device → host로 결과값을 가져오는 과정이 번거롭고, 시간이 많이 걸릴 것이라 예상하였다. 하지만 사용하는 코어 수가 압도적으로 많아서 그런지 메모리 할당이나 메모리 복사에 시간이 거의 들어가지 않는 것을 보고 놀랐다.

지금의 결과만 보아도, 멀티스레딩 없으면 10초, 멀티스레딩을 한다면 1초 혹은 0.01초 아래로 실행 시간이 현저히 떨어짐을 알 수 있다. 겨우 2048*2048 공간을 멀티스레드로 진행해도 성능이 10~100배 이상 좋아지는 것을 보며, 이제 하드웨어의 발전에는 한계가 있으니 멀티코어 프로그래밍이 점점 더 중요해질 것이라는 것을 직접 체감할 수 있었다.