Project #4

Multicore Computing

Problem 1

Date	Jun 20, 2020
Instructor	Bongsoo Sohn
Std. Name	Junhyuck Woo
Std. ID	20145337

INDEX

ENVIRONMENT

OpenMP

Hardware

- MacBook Pro (15-inch, 2017)
- Processor: 2.8 GHz Quad-Core Intel Core i7
- Memory: 16GB 2133 MHz LPDDR3

Operating System

- macOS Catalina, ver: 10.15.4

IDE (Integrated Development Environment)

- Visual Studio Code 1. 45.1
- gcc version 8.4.0 (Homebrew GCC 8.4.0_1)

Testing Environment

- iTerm2
- Build 3.3.9
- openjdk 14.0.1 2020-04-14
  OpenJDK Runtime Environment (build 14.0.1+7)
  OpenJDK 64-Bit Server VM (build 14.0.1+7, mixed mode, sharing)

CUDA

Hardware

- Desktop
- Processor: AMD Ryzen 5 2600X Six-Core Processor
- Memory: 16GB
- GPU: GeForce GTX 107

Operating System

- Ubuntu, ver: 20.04 LTS

IDE (Integrated Development Environment)

- Visual Studio Code 1. 45.1
- CUDA ver: 10.2
- gcc (Ubuntu 7.5.0-6ubuntu2) 7.5.0
- g++ (Ubuntu 7.5.0-6ubuntu2) 7.5.0
- nvcc ver 10.2.89

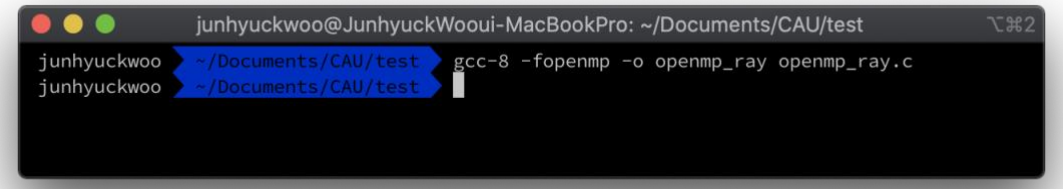
Testing Environment

- iTerm2 (Used for terminal)
- Build 3.3.9

HOW TO COMPILE

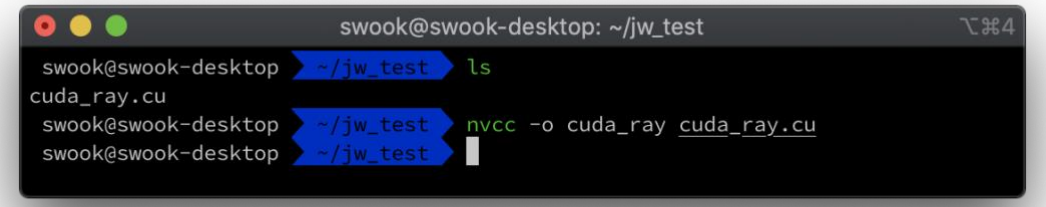
OpenMP

- gcc-8 -fopenmp -o openmp_ray openmp_ray.c



CUDA

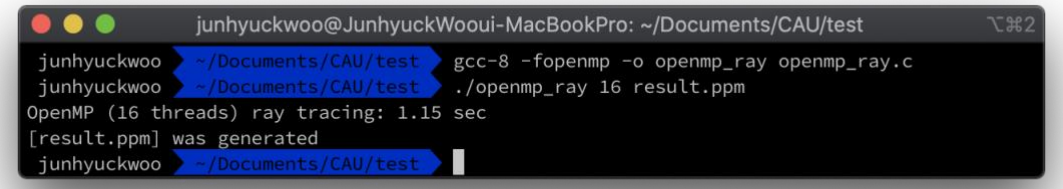
- nvcc -o cuda_ray cuda_ray.cu



HOW TO EXECUTE

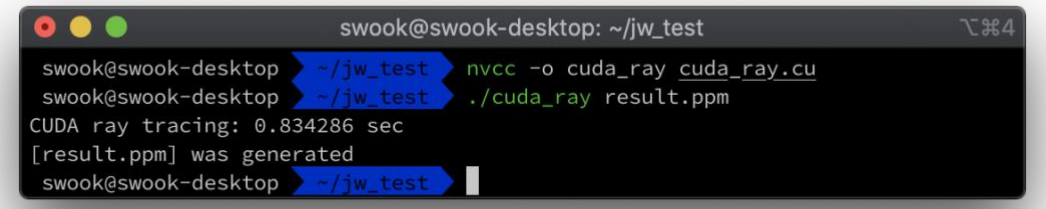
OpenMP

- ./openmp_ray [number of threads] [output filename]



CUDA

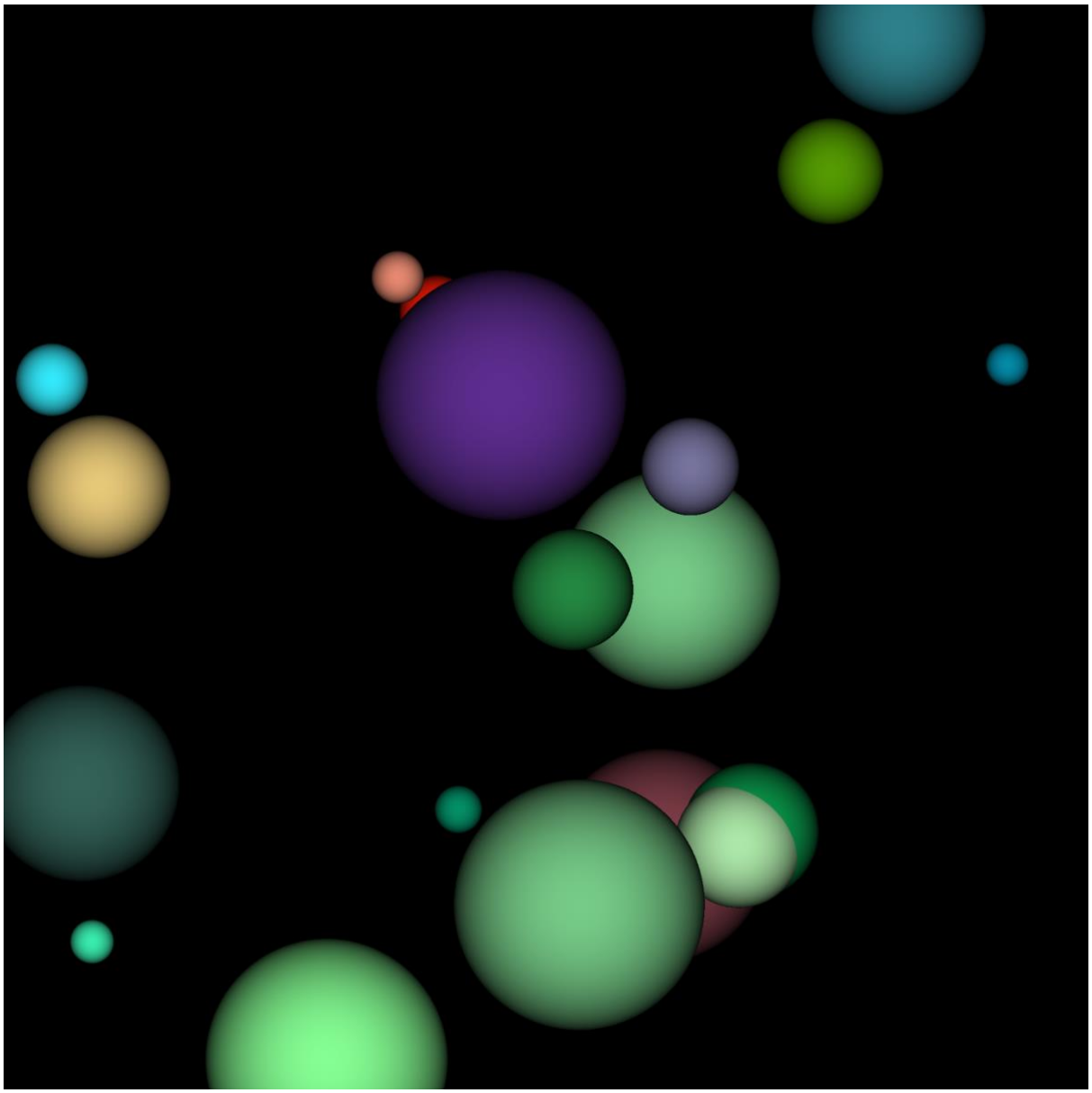
- ./cuda_ray [output filename]



RESULT PICTURE

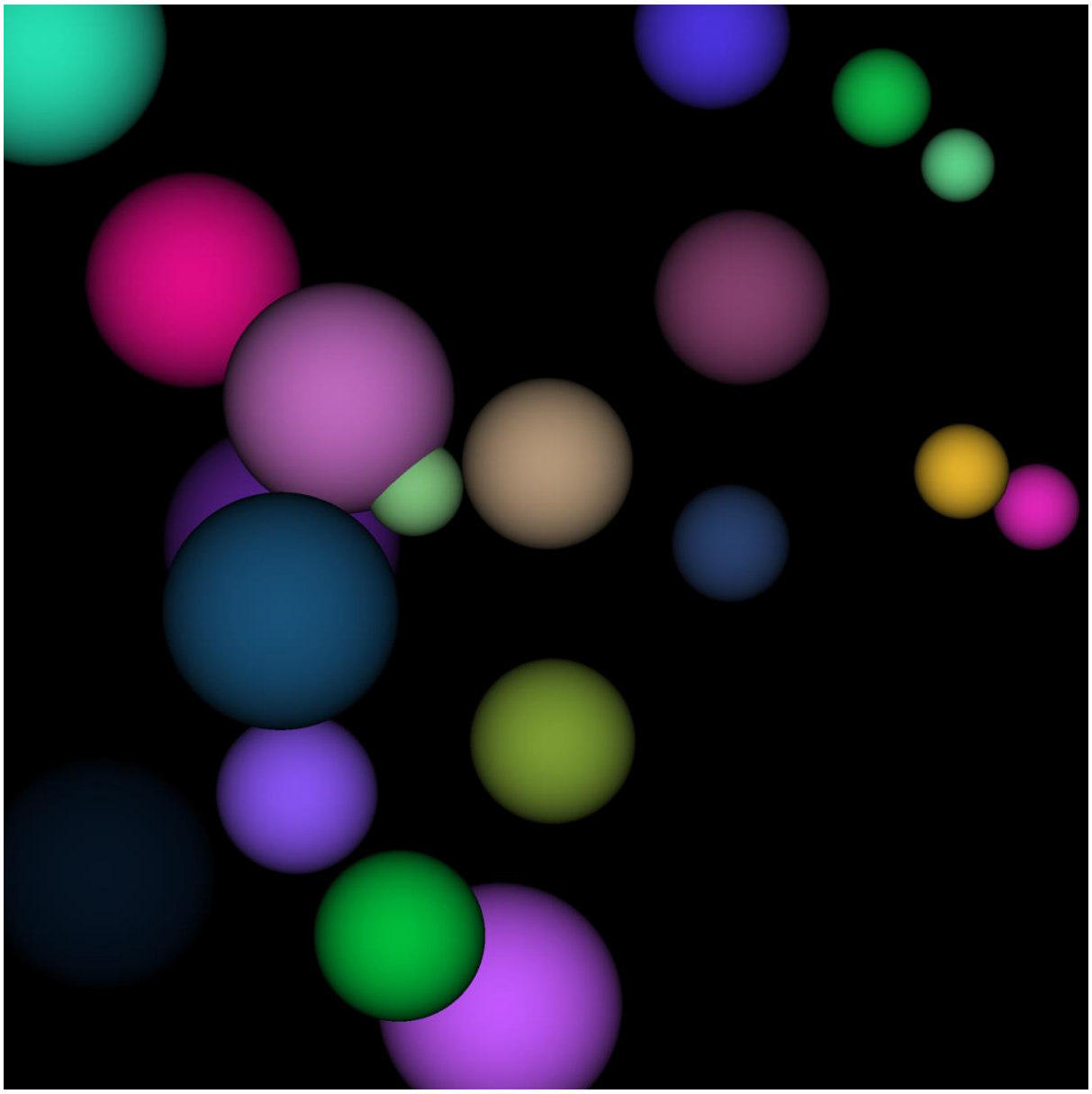
OpenMP

This image is generated by using 8 threads.



▲ Fig. 1. Ray-Tracing result picture using OpenMP

CUDA

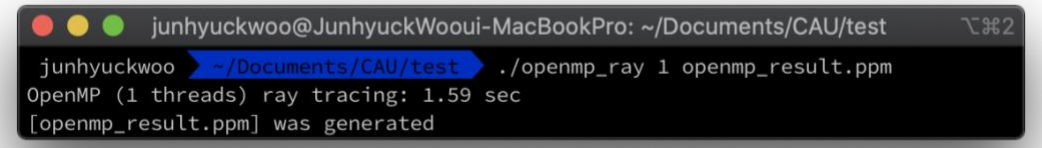


▲ Fig. 2. Ray-Tracing result picture using CUDA

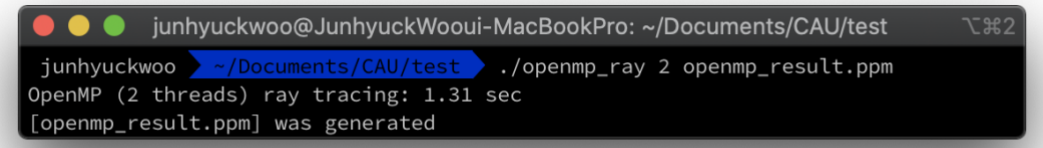
OUTPUT

OpenMP

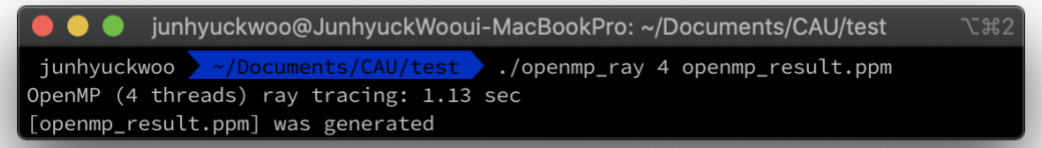
- Thread #1



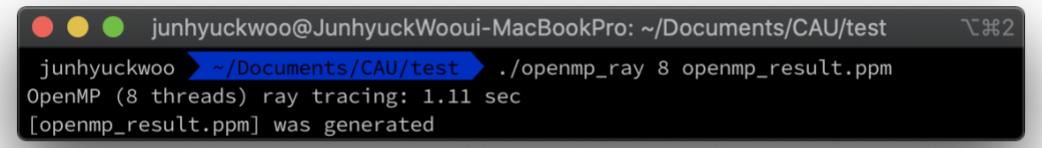
- Thread #2



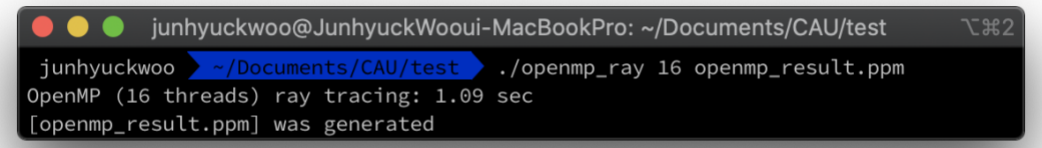
- Thread #4



- Thread #8

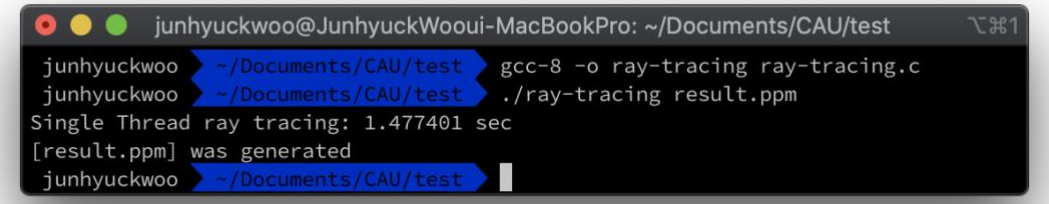


- Thread #16

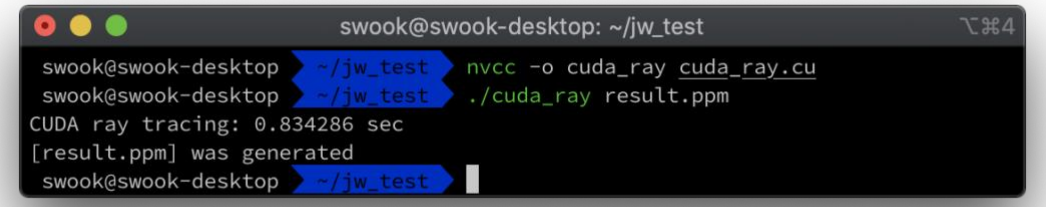


CUDA

- Thread #1

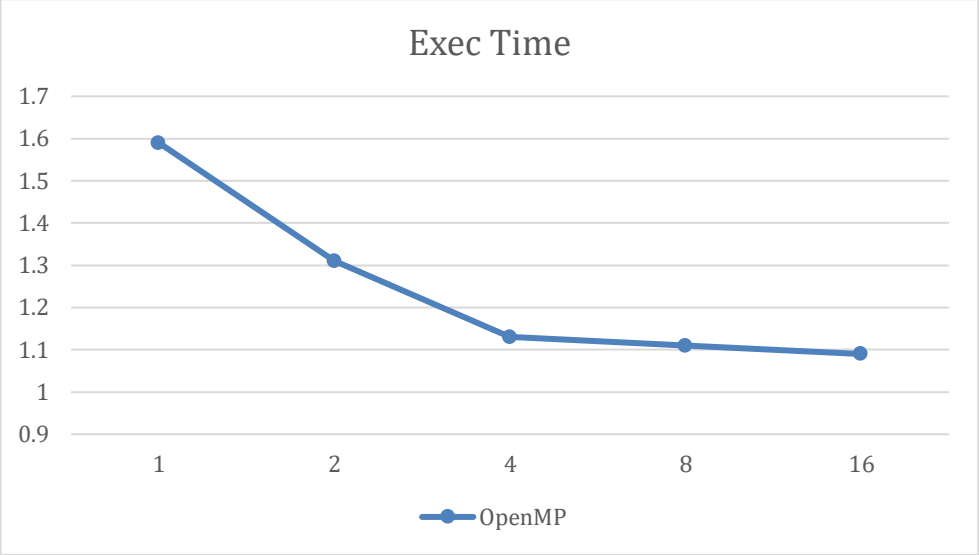


- CUDA



EXPERIMENTAL RESULT

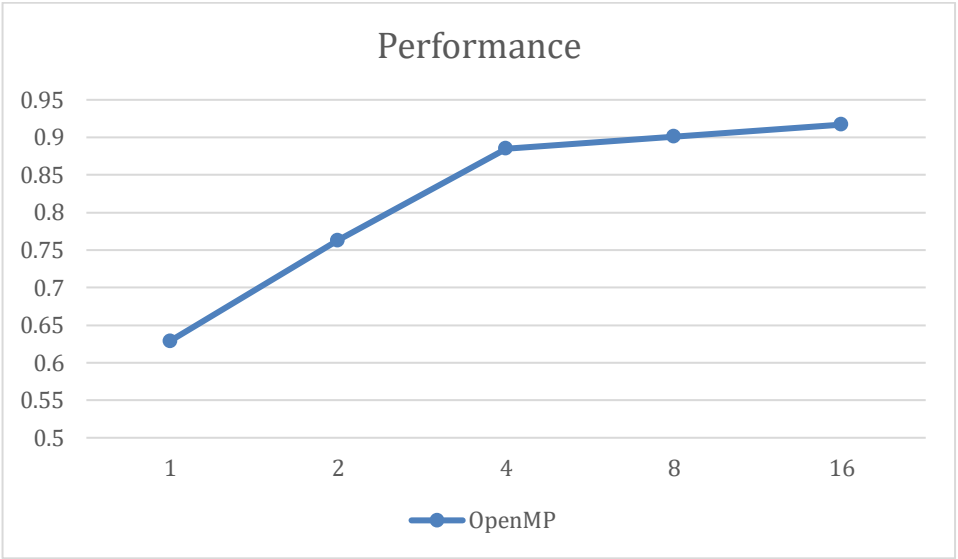
OpenMP



▲ Fig. 3. Exec Time of ray-tracing using OpenMP

Exec Time (Sec)	1	2	4	8	16
OpemMP	1.59	1.31	1.13	1.11	1.09

▲ TABLE 1

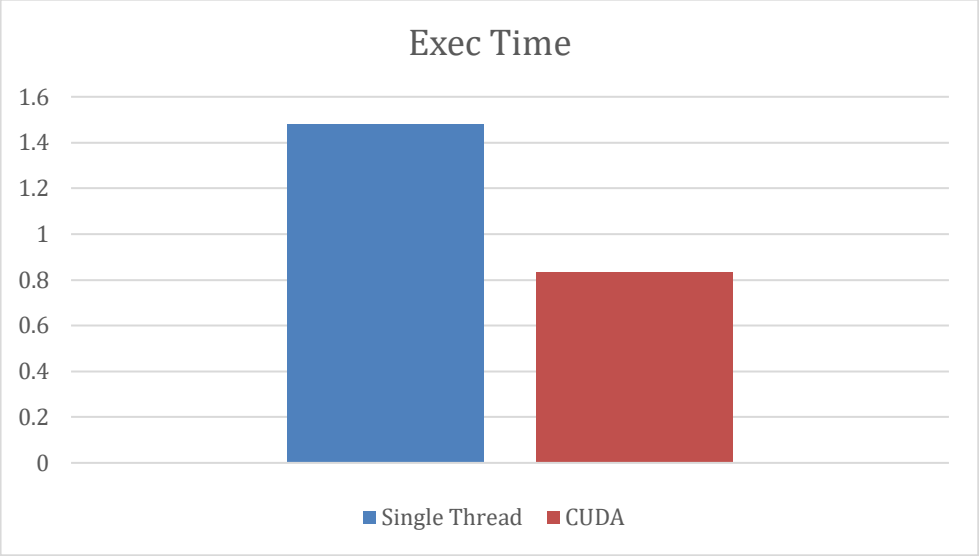


▲ Fig. 4. Performance of ray-tracing using OpenMP

Performance (1/exec time)	1	2	4	8	16
OpenMP	0.629	0.763	0.885	0.901	0.917

▲ TABLE 2

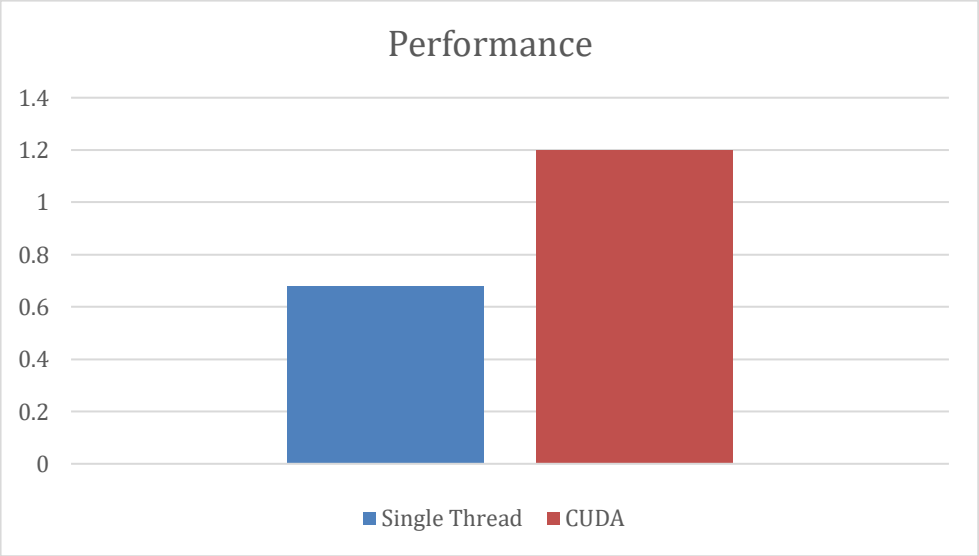
CUDA



▲ Fig. 5. Exec Time of ray-tracing using CUDA

Type	Exec Time (Sec)
Single Thread	1.477
CUDA	0.834

▲ TABLE 3



▲ Fig. 6. . Performance of ray-tracing using CUDA

Type	Performance (1/exec time)
Single Thread	0.677
CUDA	1.199

▲ TABLE 4

SOURCE CODE

OpenMP

```
#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 x, float y, float z, float ox, float oy, float *n,  float
radius ) {
   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, struct 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 = hit( s[i].x, s[i].y, s[i].z, ox, oy, &n, s[i].radius );
      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 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 {
      // Set the thread number
      option=OPENMP;
      no_threads=atoi(argv[1]);
      omp_set_num_threads(no_threads);
   }

   // Set the timer
   start_time = omp_get_wtime( );
   struct Sphere *temp_s = (struct Sphere*)malloc( sizeof(struct
Sphere) * SPHERES );
   bitmap=(unsigned char*)malloc(sizeof(unsigned
char)*DIM*DIM*4);
   #pragma omp parallel private(x) private(y)
   {
      #pragma omp for
      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;
      }

      #pragma omp for
      for (x=0;x<DIM;x++)
         for (y=0;y<DIM;y++)
            kernel(x,y,temp_s,bitmap);
   }

   ppm_write(bitmap,DIM,DIM,fp);
   end_time = (omp_get_wtime() - start_time);

   printf("OpenMP (%d threads) ray tracing: %.2lf sec\n", no_threads,
end_time);
   printf("[%s] was generated\n", argv[2]);

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

   return 0;
}
```

CUDA

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

__device__ float hit( float x, float y, float z, float ox, float oy, float *n,
float radius ) {
   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;
}

__global__ void kernel(struct Sphere* s, unsigned char* ptr)
{
   int x = blockIdx.x;
   int y = blockIdx.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].x, s[i].y, s[i].z, ox, oy, &n, s[i].radius );
      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[])
{
   double exe_time;
   clock_t start_time, end_time;
   struct Sphere *temp_s;
   unsigned char* bitmap;
   struct Sphere *d_temp_s;
   unsigned char* d_bitmap;
   dim3 blocks(DIM,DIM,1);

   // Error detection code
   if (argc!=2) {
      printf("> a.out [filename.ppm]\n");
      printf("for example, '> a.out result.ppm' means executing
CUDA\n");
      exit(0);
   }

   // Start Timer
   srand(time(NULL));
   start_time = clock();

   // Allocate the memory on host
   bitmap = (unsigned char*)malloc(sizeof(unsigned
char)*DIM*DIM*4);
   temp_s = (struct Sphere*)malloc(sizeof(struct Sphere) *
SPHERES);
   // Allocate the memory on device
   cudaMalloc( (void**)&d_temp_s, sizeof(struct Sphere) *
SPHERES );
   cudaMalloc( (void**)&d_bitmap, sizeof(unsigned
char)*DIM*DIM*4 );

   // Generate the 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;
   }

   //  Move data to device
   cudaMemcpy ( d_temp_s, temp_s, sizeof(struct Sphere) *
SPHERES, cudaMemcpyHostToDevice );

   // Calculate the ray
   kernel<<<blocks, 1>>>(d_temp_s, d_bitmap);
   cudaDeviceSynchronize();
   cudaMemcpy ( bitmap, d_bitmap, sizeof(unsigned
char)*DIM*DIM*4, cudaMemcpyDeviceToHost );

   // open the file
   FILE* fp = fopen(argv[1],"w");
   ppm_write(bitmap,DIM,DIM,fp);   // Write the image

   // Stop Timer
   end_time = clock();
   exe_time = ((double)(end_time - start_time)) /
CLOCKS_PER_SEC;

   // Print the result
   printf("CUDA ray tracing: %f sec\n", exe_time);
   printf("[%s] was generated\n", argv[1]);

   // Close the file and free the memory
   fclose(fp);
   free(bitmap);
   free(temp_s);
   cudaFree(d_bitmap);
   cudaFree(d_temp_s);
   return 0;
}
```