|  |  |
| --- | --- |
|  |  |
| Problem1 Report |  |
|  |  |
|  | Student NO: 20183784Student Name: 노현진 |
|  |  |

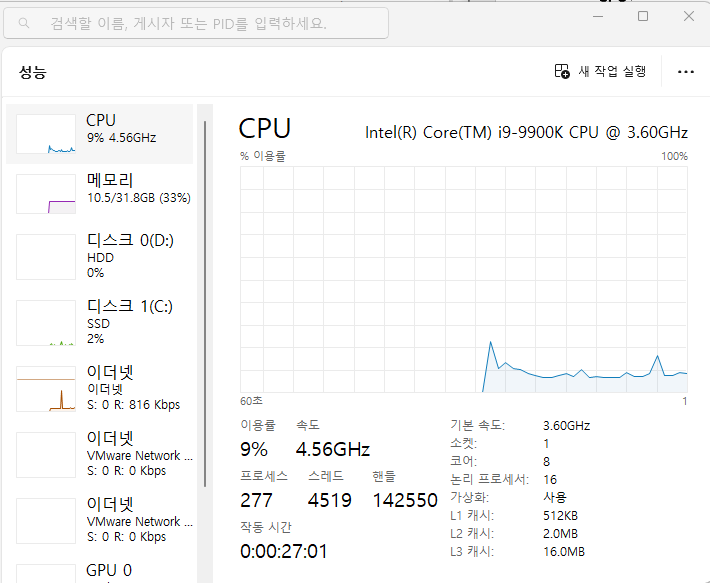
**[Execution environment]**

* **OS**



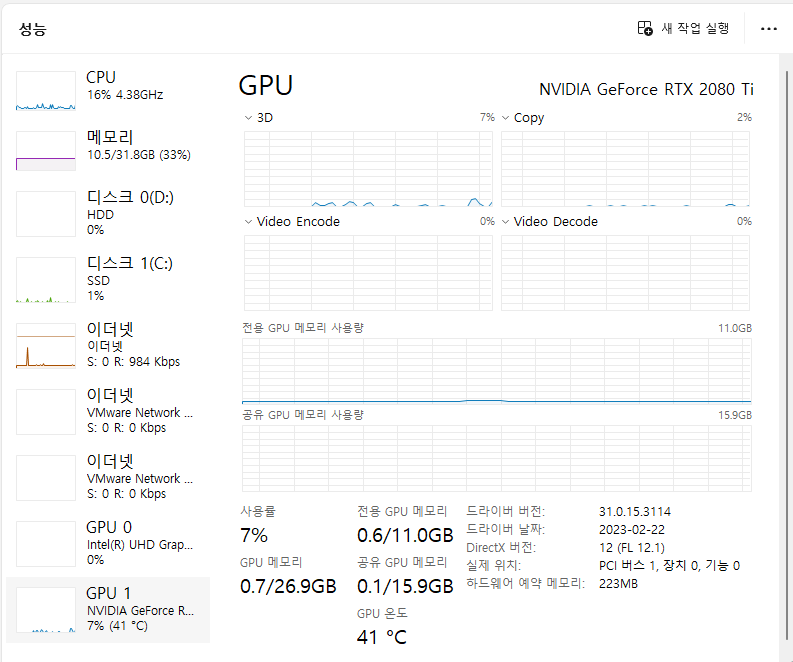
**OS type: Windows 11**

* **CPU**



**CPU type: Intel® Core™ i9-9900K CPU**

* **GPU**



**GPU type: NVIDIA GeForce RTX 2080 Ti**

**[How to compile and execute]**

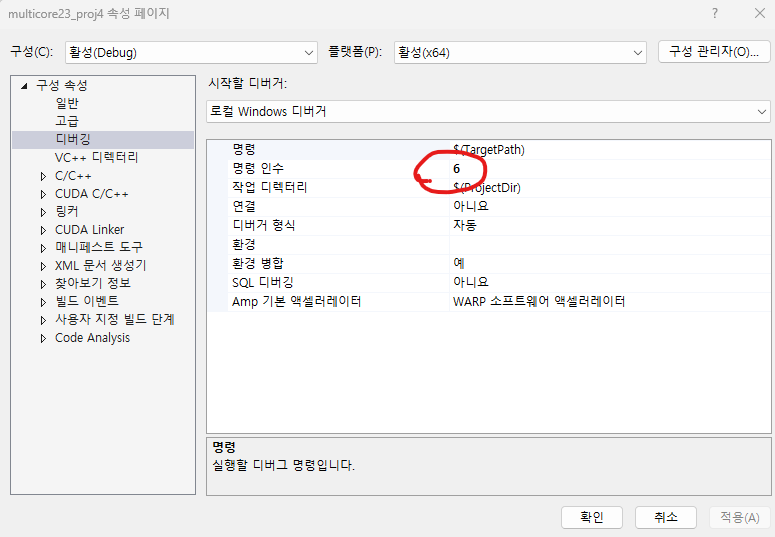
1. **Firstly, install Visual Studio 2022.**
2. **Secondly, install CUDA.**
3. **After installations, open the Visual Studio 2022.**

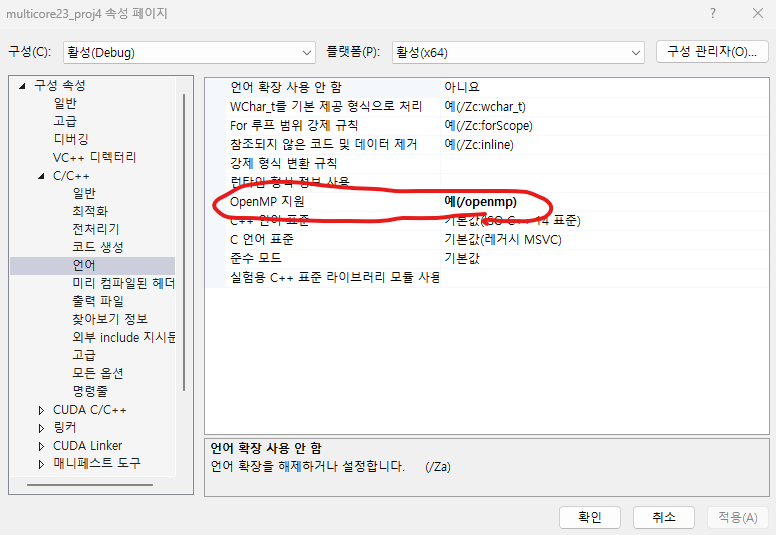


1. **Create new project. The project should be CUDA Run time project.**

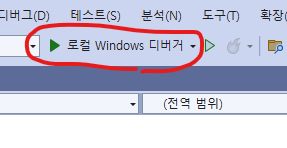
****

1. **Put the source code that you want to run into the project.**
2. **Before execution, set the parameter and make OpenMP available.**

****

****

1. **Finally, run the source code.**



**[Source Code]**

* **openmp\_ray.cpp**

#include <omp.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;

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 (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 i;

int num\_threads;

int x, y;

unsigned char\* bitmap;

double start\_time, end\_time;

Sphere\* temp\_s = (Sphere\*)malloc(sizeof(Sphere) \* SPHERES);

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

srand(time(NULL));

if (argc != 2) {

printf("> a.out [number of threads]\n");

printf("for example, '> a.out 8' means executing OpenMP with 8 threads\n");

exit(0);

}

else {

num\_threads = atoi(argv[1]);

}

omp\_set\_num\_threads(num\_threads);

start\_time = omp\_get\_wtime();

for (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);

#pragma omp parallel for default(shared) private(x, y)

for (x = 0; x < DIM; x++)

for (y = 0; y < DIM; y++) kernel(x, y, temp\_s, bitmap);

end\_time = omp\_get\_wtime();

ppm\_write(bitmap, DIM, DIM, fp);

fclose(fp);

free(bitmap);

free(temp\_s);

printf("OpenMP (%d threads) ray tracing: %lf sec\n", num\_threads, end\_time - start\_time);

printf("[result.ppm] was generated.\n");

return 0;

}

* **cuda\_ray.cu**

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <time.h>

#include <math.h>

#include <chrono>

#include <iostream>

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

using namespace std;

using namespace std::chrono;

#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;

}

};

\_\_global\_\_ void kernel(Sphere\* s, unsigned char\* ptr) {

int x = threadIdx.x + blockIdx.x \* blockDim.x;

int y = threadIdx.y + blockIdx.y \* blockDim.y;

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);

int i;

float r = 0, g = 0, b = 0;

float maxz = -INF;

for (i = 0; i < SPHERES; i++) {

float n;

//float t = s[i].hit(ox, oy, &n);

float t;

float dx = ox - s[i].x;

float dy = oy - s[i].y;

float radius = s[i].radius;

if (dx \* dx + dy \* dy < radius \* radius) {

float dz = sqrtf(radius \* radius - dx \* dx - dy \* dy);

n = dz / sqrtf(radius \* 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 i;

Sphere\* temp\_s;

Sphere\* d\_temp\_s;

unsigned char\* bitmap;

unsigned char\* d\_bitmap;

int size1 = sizeof(Sphere) \* SPHERES;

int size2 = sizeof(unsigned char) \* DIM \* DIM \* 4;

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

srand(time(NULL));

// Allocate space for device copies of temp\_s and bitmap

cudaMalloc((void\*\*)&d\_temp\_s, size1);

cudaMalloc((void\*\*)&d\_bitmap, size2);

// Allocate space for host copies of temp\_s and bitmap

temp\_s = (Sphere\*)malloc(size1);

bitmap = (unsigned char\*)malloc(size2);

// Setup initial values

for (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;

}

auto start\_time = high\_resolution\_clock::now();

// Copy values to device

cudaMemcpy(d\_temp\_s, temp\_s, size1, cudaMemcpyHostToDevice);

// Setup the execution configuration

dim3 dimBlock(32, 32, 1);

dim3 dimGrid(64, 64, 1);

kernel<<<dimGrid, dimBlock>>>(d\_temp\_s, d\_bitmap);

// Copy result back to host

cudaMemcpy(bitmap, d\_bitmap, size2, cudaMemcpyDeviceToHost);

auto end\_time = high\_resolution\_clock::now();

auto duration = duration\_cast<microseconds>(end\_time - start\_time);

ppm\_write(bitmap, DIM, DIM, fp);

fclose(fp);

free(bitmap);

free(temp\_s);

cudaFree(d\_bitmap);

cudaFree(d\_temp\_s);

cout << "CUDA ray tracing: " << duration.count() / 1000000.0 << " sec" << endl;

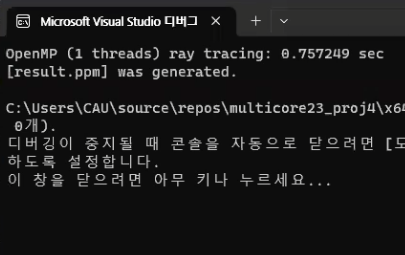
cout << "[result.ppm] was generated." << endl;

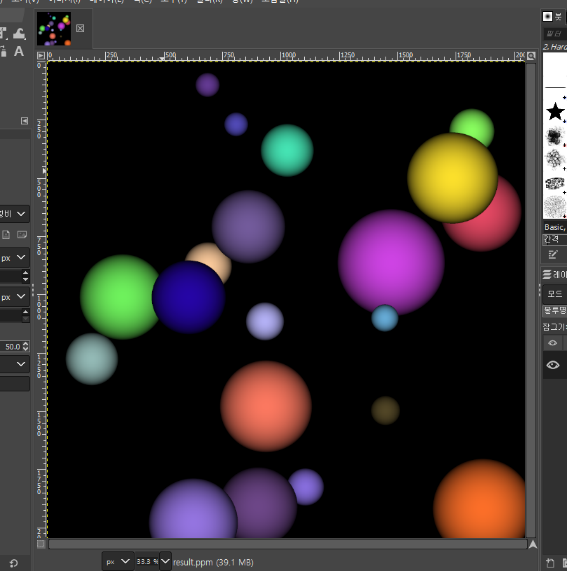
return 0;

}

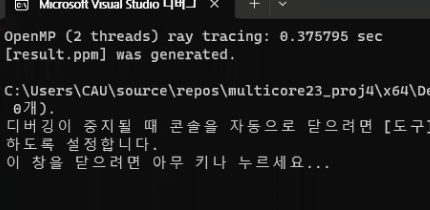
**[Results]**

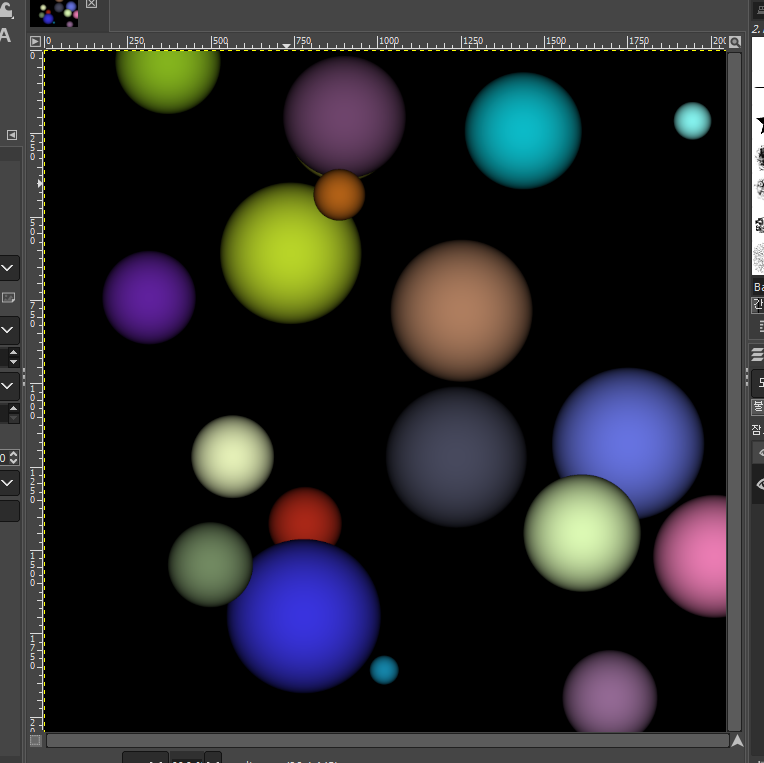
* **openmp\_ray.cpp**
* 1 thread



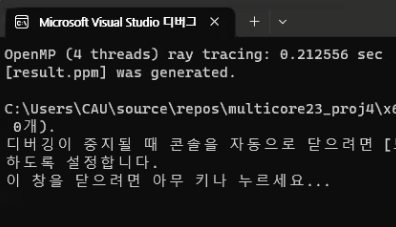


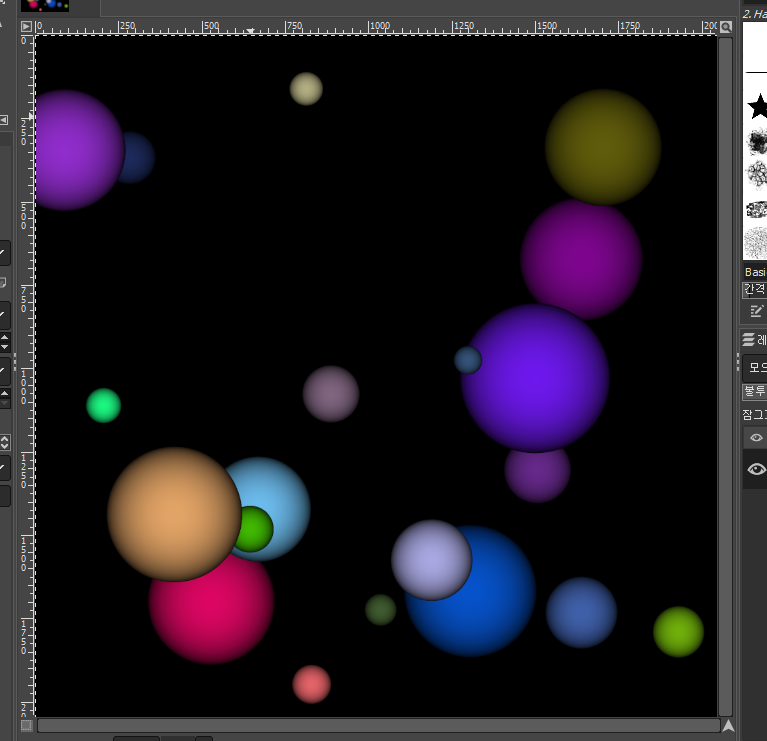
* 2 threads



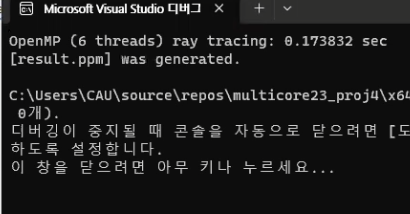


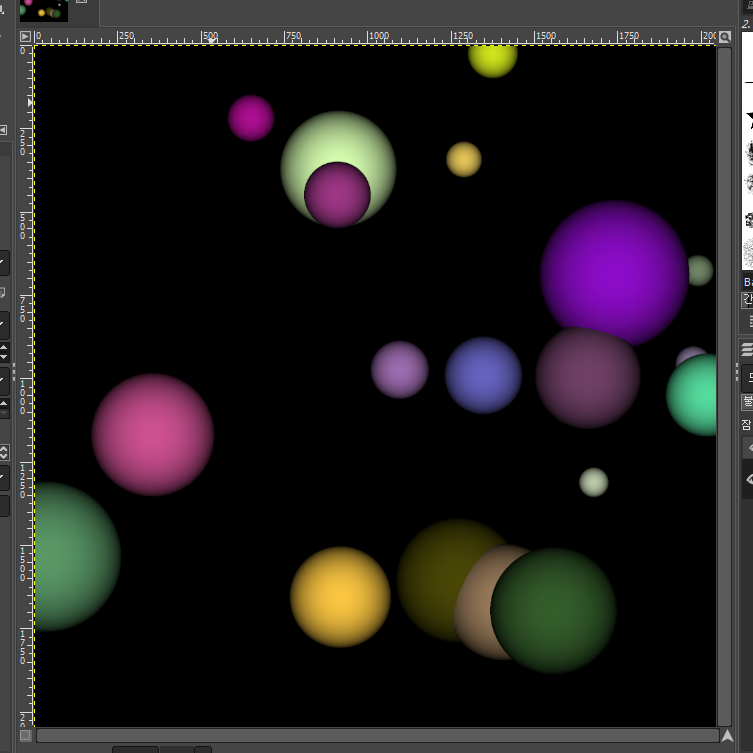
* 4 threads



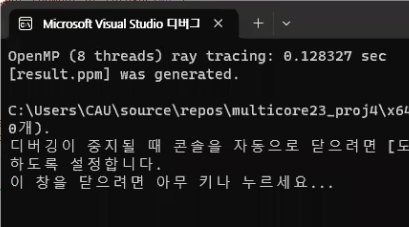


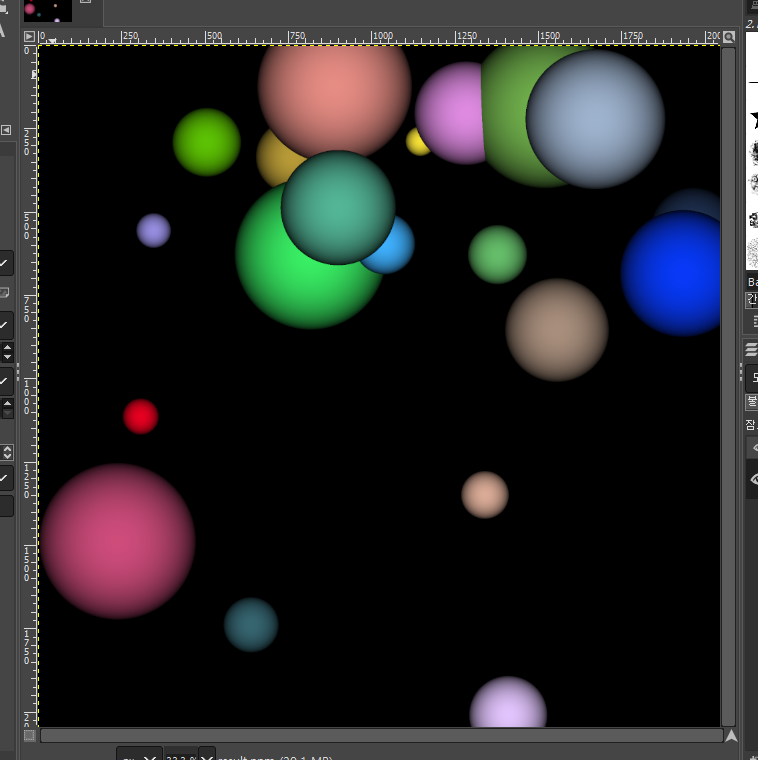
* 6 threads



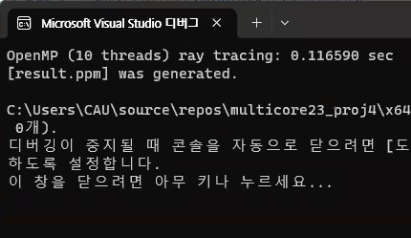


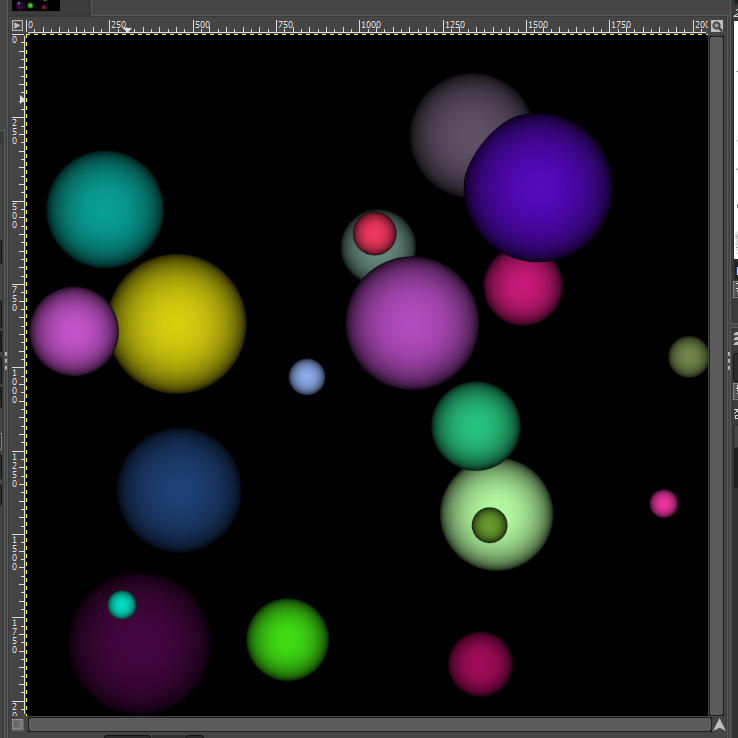
* 8 threads



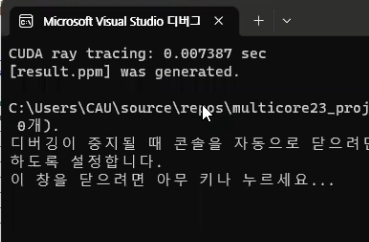


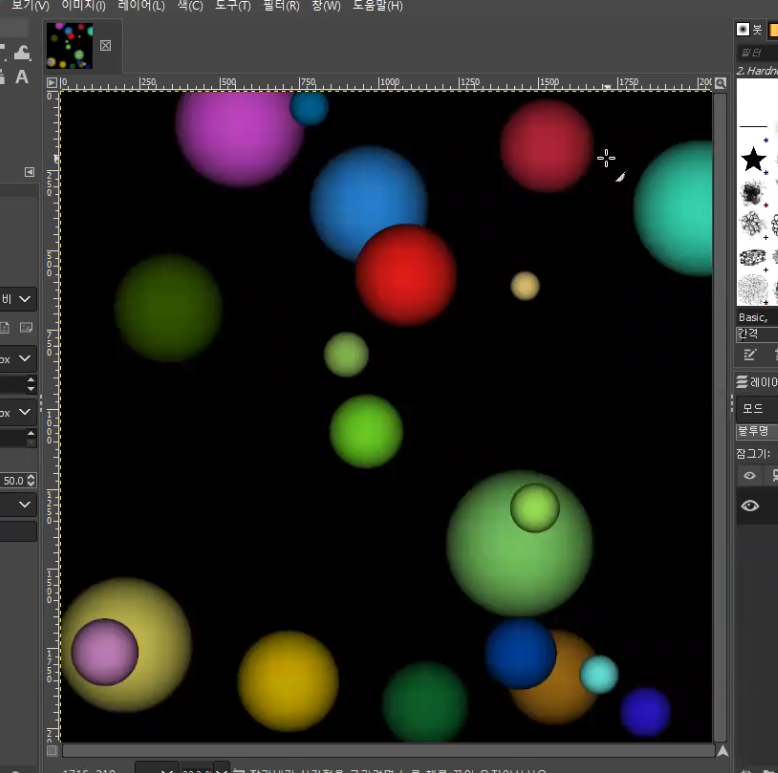
* 10 threads



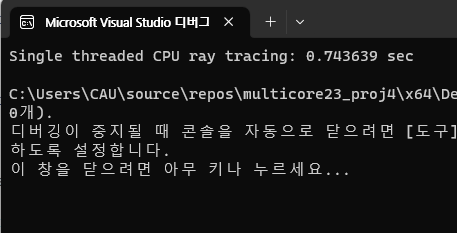


* **cuda\_ray.cu**





* **raytracing.cpp (Given source code)**



* **Execution Time**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| OpenMp | 1 | 2 | 4 | 6 | 8 | 10 |
| Exec time | 0.757249 sec | 0.375795 sec | 0.212556 sec | 0.173832 sec | 0.128327 sec | 0.116590 sec |

|  |  |
| --- | --- |
| CUDA |  |
| Exec time | 0.007387 sec |

|  |  |
| --- | --- |
| Single threaded CPU |  |
| Exec time | 0.743639 sec |

* **Performance**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| OpenMp | 1 | 2 | 4 | 6 | 8 | 10 |
| Performance (1/exec time) | 1.3205 | 2.6610 | 4.7046 | 5.7526 | 7.7925 | 8.5770 |

|  |  |
| --- | --- |
| CUDA |  |
| Performance (1/exec time) | 153.3729 |

|  |  |
| --- | --- |
| Single threaded CPU |  |
| Performance (1/exec time) | 1.3447 |

**[Interpretation/Explanation on the Results]**

The above results shows that the more number of threads, the better performance. When the number of threads is small (ex. number of threads = 1), it takes 0.7 seconds. But, when the number of threads is large, program performance is improved dramatically. In other words, using OpenMp with large number of threads has good performance. Also, using CUDA library has much better performance than OpenMp. Using CUDA library is 20 times faster than OpenMp. It’s because GPU has better performance in highly parallel applications than CPU.