INDEX(20200283 소프트웨어학부 조범희)

1. OS/Execution Environment

OS : Window11

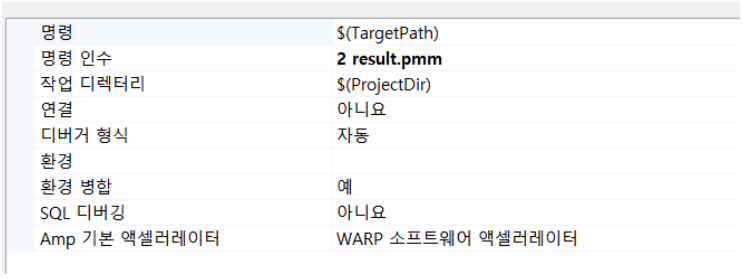
Processor : 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz

Number of Core : 4

Memory : Ram 8GB

IDE : Visual Studio 2022, Google Colab

Prob1 실행방법

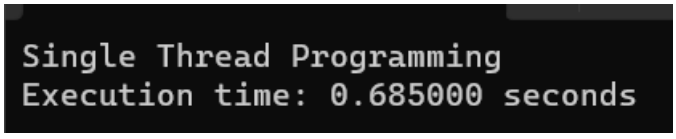


<You can execute command-line arguments in Visual Studio.>

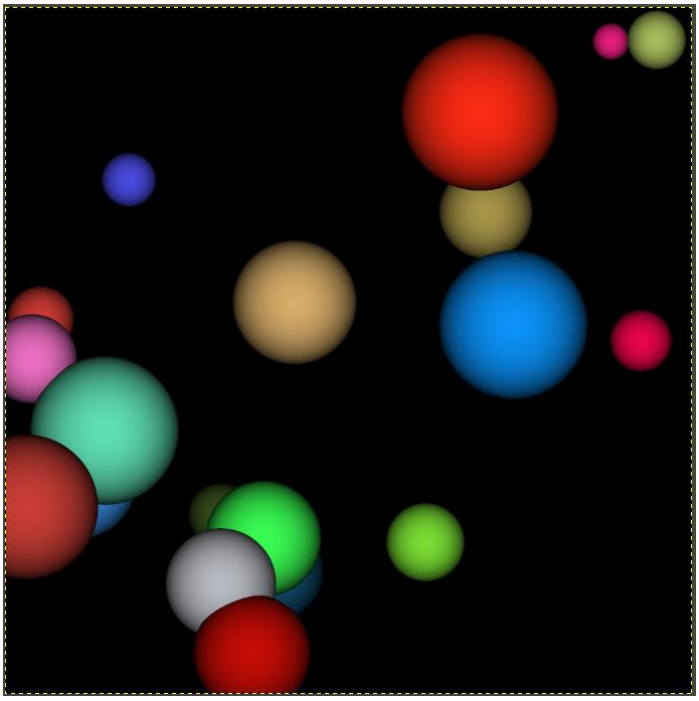
2. Ray Tracing

- Ray Tracing

Ray tracing is a rendering technique used in graphics to simulate the characteristics of real light rays and create 3D scenes. Ray tracing involves computationally intensive and complex algorithms. In this project, we will parallelize ray tracing using OpenMP and CUDA to explore which approach is better and what the differences are.



<Execution Time>



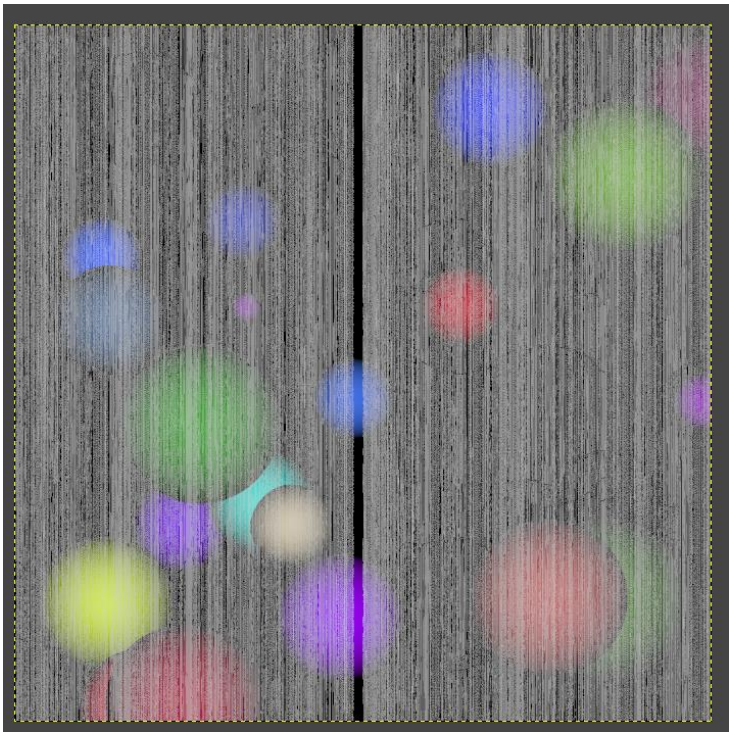
<No Parallel Programming Result>

3. openMP

- Code

```
    omp_set_num_threads(no_threads);    // 쓰레드 수 결정
#pragma omp parallel for private(x), private(y), shared(bitmap) // x, y는 독립변수이므로 private 처리
    for (x = 0; x < DIM; x++)
        for (y = 0; y < DIM; y++) kernel(x, y, temp_s, bitmap);
```

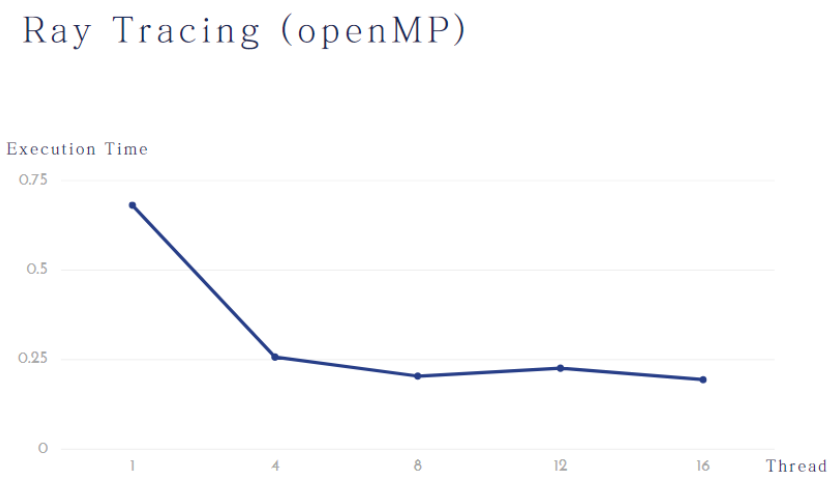
Parallel processing begins in this section. The most computationally intensive part of the program is where the kernel is executed, with x and y values iterating. Here, the number of threads is determined, and OpenMP is used to parallelize the processing. Additionally, the variables x and y must be treated individually. Otherwise, accurate results cannot be obtained. If not properly marked as private, the same x and y values will be processed in parallel, leading to incorrect results as shown below.



<Wrong Result>

The reason for this issue seems to be that during parallel processing, the overlapping of x and y values prevents the bitmap from being properly drawn. Additionally, the kernel continuously stores random values in the bitmap, so it requires shared processing. When everything is appropriately adjusted, improved processing time and clean images are output correctly.

- Graph / Analysis



```
OpenMP (1 threads) ray tracing: 0.681 sec
[result.pmm] was generated.
```

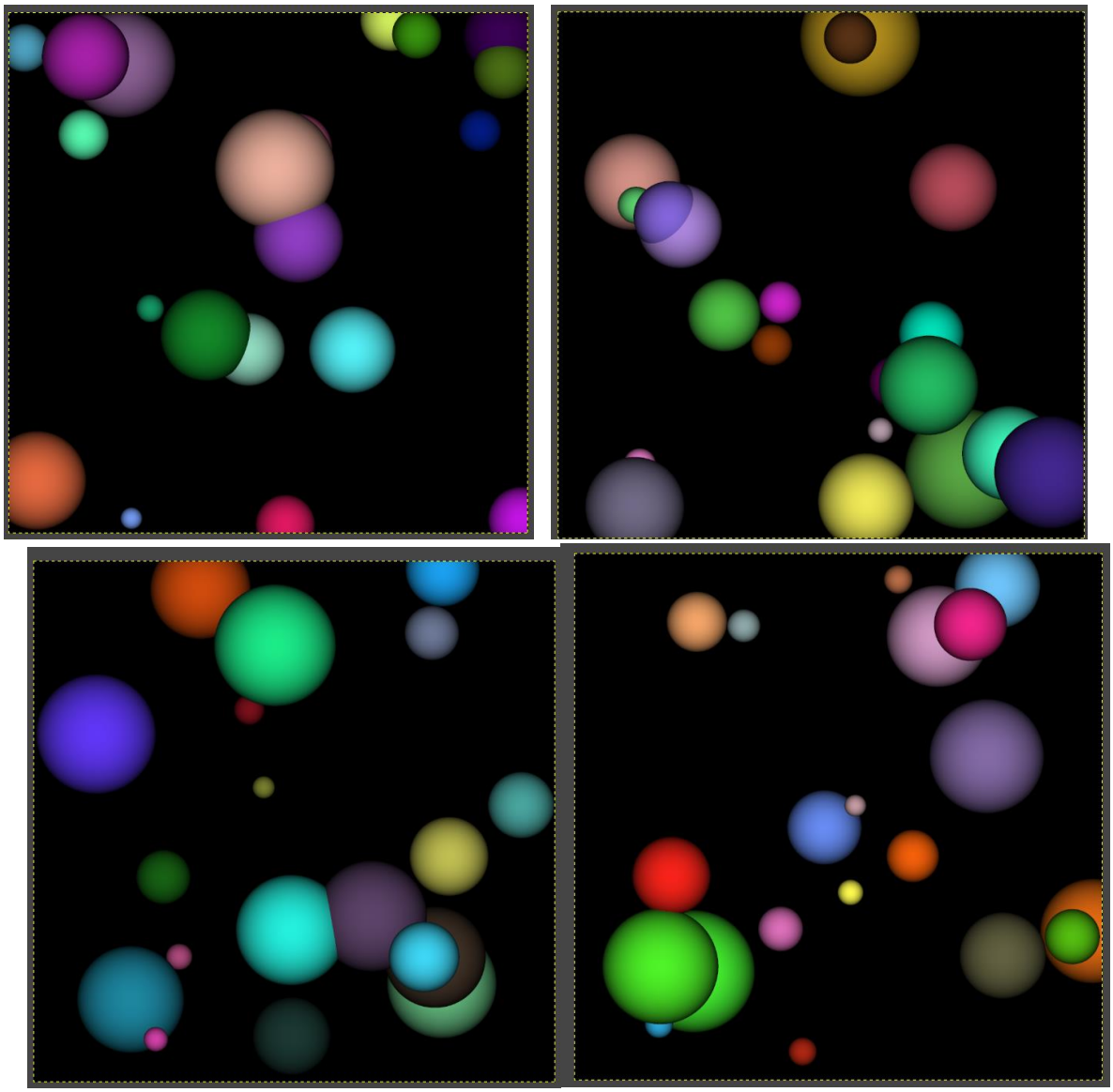
```
OpenMP (4 threads) ray tracing: 0.257 sec
[result.pmm] was generated.
```

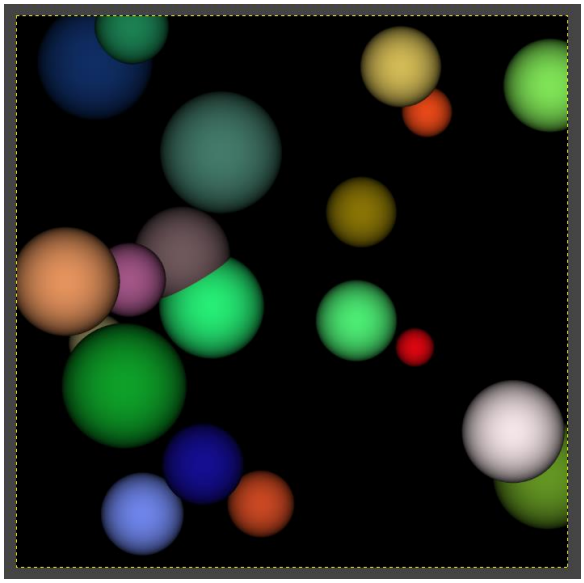
```
OpenMP (8 threads) ray tracing: 0.204 sec
[result.pmm] was generated.
```

```
OpenMP (12 threads) ray tracing: 0.226 sec
[result.pmm] was generated.
```

```
OpenMP (16 threads) ray tracing: 0.194 sec
[result.pmm] was generated.
```

When running with only one thread, similar results were obtained compared to the single-threaded execution. However, from four threads to sixteen threads, the execution time remained relatively constant. It can be observed that the performance improved by approximately three times when using multi-threading compared to a single thread.





<1,4,8,12,16 순서대로 결과 이미지>

All 20 spheres with random colors and sizes were neatly printed. Since the code generates them randomly in real-time, it is expected that different images will be produced each time. Additionally, there may be a slight clustering effect, which can be attributed to the nature of srand where random values can be the same at a certain point in time if many threads are working simultaneously. However, even in the case of a single thread, there may be a perception of clustering on one side, so there is also a possibility of simple coincidence.

4. CUDA

- Code/Analysis

```
__global__ void kernel(Sphere* s, unsigned char* ptr) {
    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 = -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;
        }
    }
```

<CUDA Kernel 함수>

The kernel function has been modified. In the previous version, x and y values were received through a loop, but in this version, the values are adjusted sequentially using thread and block indices to generate the spheres.

```
cudaEvent_t start, stop;
float elapsedTime;
cudaEventCreate(&start);
cudaEventCreate(&stop);
```

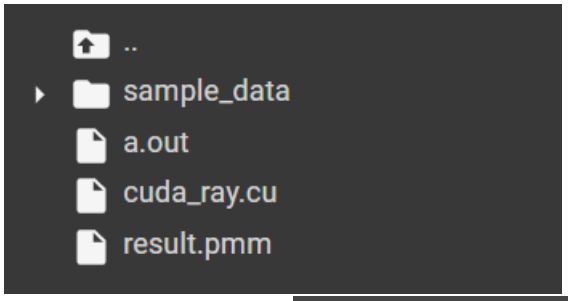
< Variables for calculating the running time of a program>

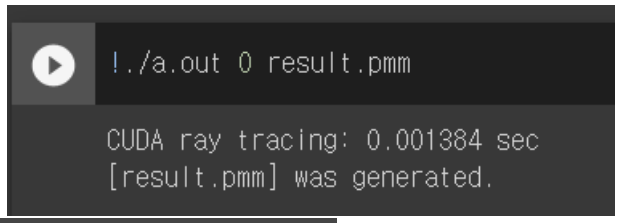
```
bitmap = (unsigned char*)malloc(sizeof(unsigned char) * DIM * DIM * 4);
unsigned char* d_bitmap;
Sphere* d_temp_s;
cudaMalloc((void**)&d_bitmap, sizeof(unsigned char) * DIM * DIM * 4);
cudaMalloc((void**)&d_temp_s, sizeof(Sphere) * SPHERES);

cudaMemcpy(d_temp_s, temp_s, sizeof(Sphere) * SPHERES, cudaMemcpyHostToDevice);
dim3 block(16, 16);
dim3 grid(DIM / block.x, DIM / block.y);

cudaEventRecord(start, 0);
kernel<<<grid, block>>>(d_temp_s, d_bitmap);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&elapsedTime, start, stop);

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

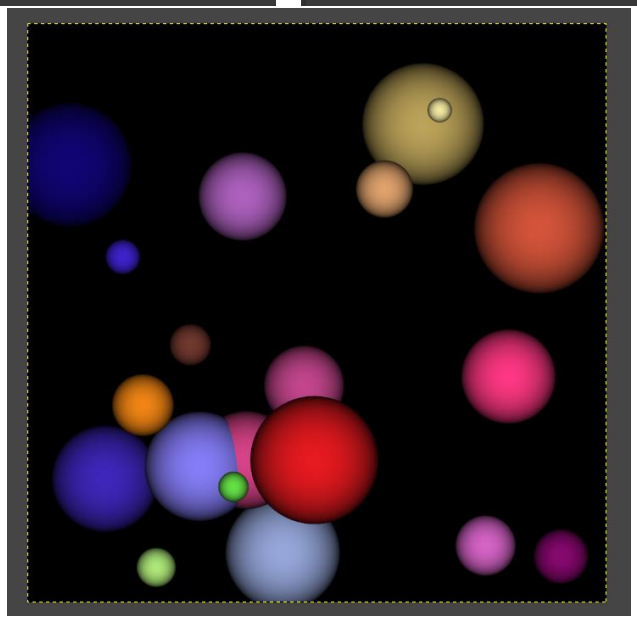
In the GPU, it is not possible to directly access the memory of the CPU (host). Therefore, it requires allocating memory in the GPU space and transferring data between the host and device. In this case, the bitmap for generating and storing the spheres is allocated in the GPU space and copied from the host to the device.

The line Dim3 block(16,16) creates blocks of size 16x16. The size of the grid is determined by DIM/16, which is 2048/16 = 128. Therefore, there are a total of 16,384 blocks (128 * 128), and each block uses 16x16 threads. Consequently, the total number of threads is 16,384 * 256, which is equal to the total number of pixels (2048 * 2048). Each thread is responsible for one pixel, allowing for fast calculations.

This is one of the advantages of GPUs, as they can utilize an enormous number of threads that would be impractical with multi-core programming on the CPU.





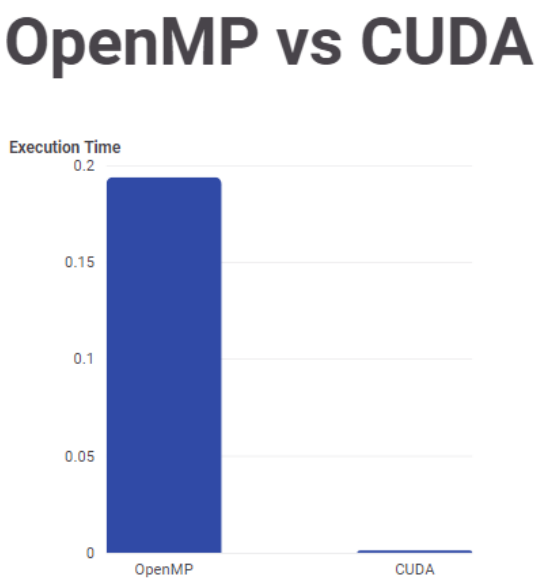


<Result>

It shows a speed that is incomparable to OpenMP.

5. openMP vs CUDA

- Execution Time



<openMP vs CUDA>

When implemented with CUDA, you can see a several-fold speedup compared to openMP. However, it cannot be always claimed that CUDA is faster. Ray tracing is a task that GPUs excel at, which is why we observe such contrasting results. Additionally, appropriate thread allocation is crucial. In this code, CUDA optimally allocates one thread per pixel, resulting in efficient utilization. If single-threaded execution was used in CUDA, significant results would not have been achieved.