2023-1 Multicore Computing, Project #4

Problem 1

2023-05-24

CAU SW 20184286

LEE DONGHWA

Environment

Google Colab (GPU type : T4)

Just run the Jupyter file in Colab.

Compile > !nvcc {filename}

Execute > !./a.out

Source code

openmp_ray.cpp

```
%%writefile openmp_ray.cpp
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <omp.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 no_threads;
    int x,y;
    unsigned char* bitmap;

    srand(time(NULL));
    if (argc!=2) {
        printf("> openmp_ray [option]\n");
        printf("[option] 1~16: OpenMP using 1~16 threads\n");
        printf("for example, '> openmp_ray 8' means executing OpenMP
with 8 threads\n");
        exit(0);
    }
    FILE* fp = fopen("result.ppm", "w");

    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;
    }

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

    // check timer with clock_k
    clock_t start_time = clock();

    // using openMP multithreading
    #pragma omp parallel for schedule(dynamic) num_threads(no_threads)
    for (y = 0; y < DIM; y++) {
        for (x=0;x<DIM;x++) {
```

```
            kernel(x, y, temp_s, bitmap);
        }
    }

    clock_t end_time = clock();
    clock_t diff_time = end_time - start_time;
    printf("OpenMP (%d threads) ray tracing: %.3lf sec. \n",
no_threads, (double)diff_time/CLOCKS_PER_SEC);

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

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

    return 0;
}
```

Cuda_ray.cu

```
%%writefile cuda_ray.cu
#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;
    __device__ 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;
```

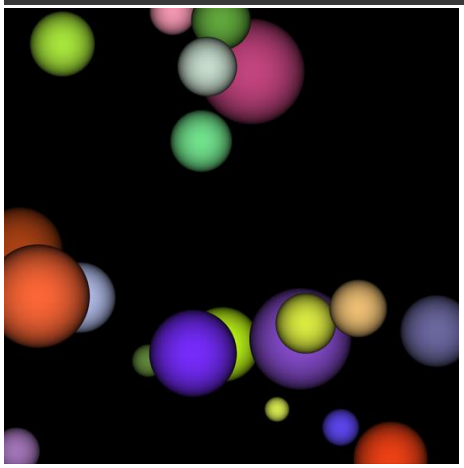
```
        }
        return -INF;
    }
};

// using CUDA
__global__ void          (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);

    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]);
```

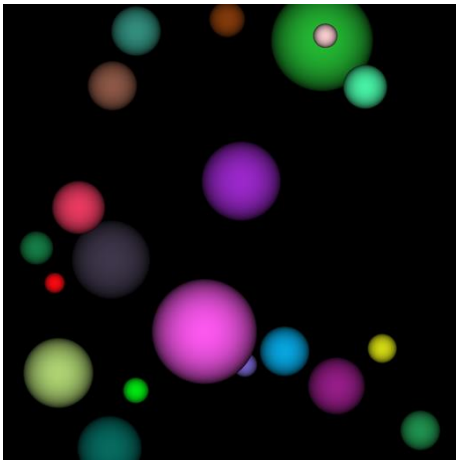
```
        }
        fprintf(fp, "\n");
    }
}


int main(int argc, char* argv[])
{
    unsigned char* bitmap;
    Sphere* dev_s;
    unsigned char* dev_b;

    srand(time(NULL));
    FILE* fp = fopen("result.ppm", "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;
    }

    cudaMalloc((void**)&dev_s, SPHERES * sizeof(Sphere));
    cudaMalloc((void**)&dev_b, sizeof(unsigned char) * DIM * DIM * 4);

    cudaMemcpy(dev_s, temp_s, sizeof(Sphere) * SPHERES,
cudaMemcpyHostToDevice);

    clock_t start_time = clock();


    // number of blocks, threads per block
    // it computes using CUDA and get output of ppm image
    kernel<<<dim3((DIM + 15) / 16, (DIM + 15) / 16), dim3(16,
16)>>>(dev_s, dev_b);

    bitmap = (unsigned char*)malloc(sizeof(unsigned char) * DIM * DIM *
4);
    cudaMemcpy(bitmap, dev_b, sizeof(unsigned char) * DIM * DIM * 4,
cudaMemcpyDeviceToHost);

    clock_t end_time = clock();
    clock_t diff_time = end_time - start_time;
```

```
    printf("CUDA ray tracing: %f sec. \n",
(double)diff_time/CLOCKS_PER_SEC);

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

    fclose(fp);
    free(bitmap);
    free(temp_s);
    cudaFree(dev_s); cudaFree(dev_b);
    return 0;
}
```

Result

```
!./a.out 16

OpenMP (16 threads) ray tracing: 0.624 sec.
[result.ppm] was generated.
```

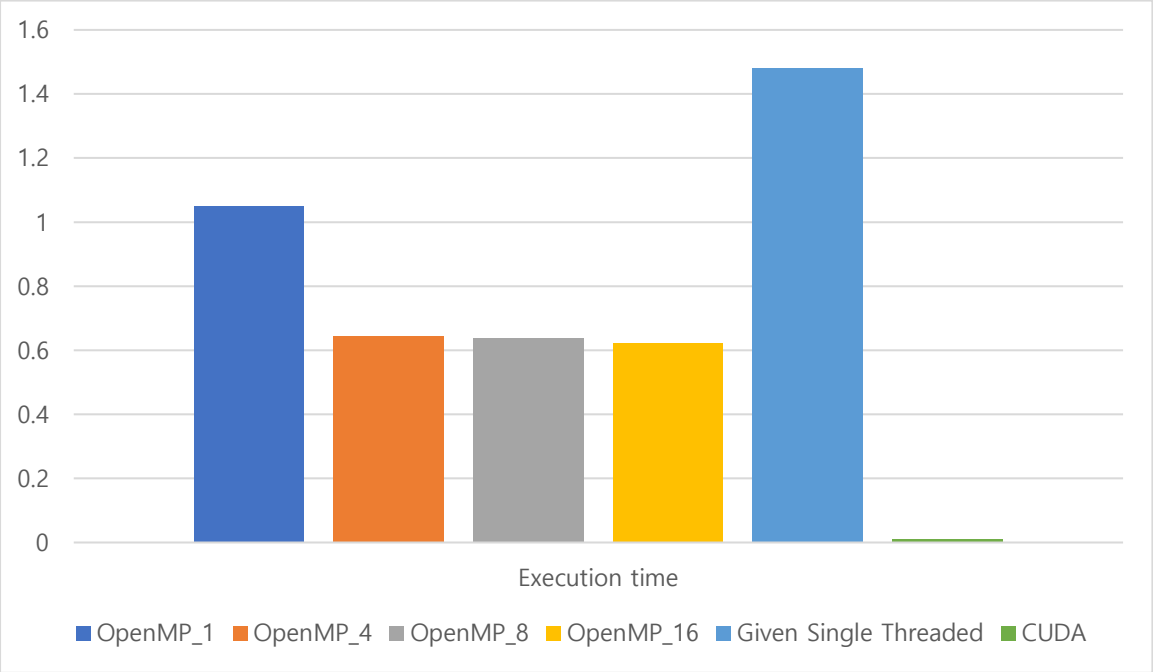




```
!./a.out

CUDA ray tracing: 0.012876 sec.
[result.ppm] was generated.
```

Tables_ (unit : sec)

	1	4	8	16
OpenMP	1.052	0.645	0.639	0.624
Given single threaded	1.481			
CUDA	0.012593			



Explanation / Analysis_

As the number of threads increased, I was able to observe a slight improvement in performance, thanks to the changes in the number of threads in OpenMP. I also noticed a slight difference compared to the case of using a single thread. Furthermore, I found that using CUDA with GPU resulted in significantly faster performance. Therefore, through experimentation, I confirmed that processing ppm files is more advantageous on the GPU compared to the CPU.