

2023.1 Multicore Computing, Project #4

Problem 1

|  |  |
| --- | --- |
| Course / Class: | Multicore Computing / Class 01 |
| Instructor: | Bong-Soo Sohn |
| Date: | 2023. 06. 13 |
| Student ID: | 20184757 |
| Student Name: | Youngseok Joo |

Table of Contents

[1. Environment 3](#_Toc137523902)

[2. Compilation 3](#_Toc137523903)

[1) openmp\_ray.cpp 3](#_Toc137523904)

[2) cuda\_ray.cu 3](#_Toc137523905)

[3. Execution 3](#_Toc137523906)

[1) openmp\_ray.cpp 3](#_Toc137523907)

[2) cuda\_ray.cu 3](#_Toc137523908)

[4. Source Code 4](#_Toc137523909)

[1) openmp\_ray.cpp 4](#_Toc137523910)

[2) cuda\_ray.cu 6](#_Toc137523911)

[5. Output Results 8](#_Toc137523912)

[1) openmp\_ray 8](#_Toc137523913)

[2) cuda\_ray 9](#_Toc137523914)

[6. Experimental Results 10](#_Toc137523915)

[1) OpenMP 10](#_Toc137523916)

[2) CUDA 11](#_Toc137523917)

# **Environment**

* Hardware
* CPU: Intel® Core™ i5-10400 CPU @ 2.90GHz
* Memory: 16.0 GB, 2667MHz
* GPU: NVIDIA GeForce GTX 1660 SUPER (1408 CUDA Cores, 6144MiB Memory)
* Software
* Windows 10 Home
* WSL2 (Windows Subsystem for Linux) – Ubuntu 20.04.6 LTS
* CUDA Version: 12.1
* NVIDIA Driver Version: 531.79
* gcc: gcc (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0
* nvcc: V12.1.105, build cuda\_12.1.r12.1/compiler.32688072\_0

# **Compilation**

## openmp\_ray.cpp

$ gcc -o openmp\_ray openmp\_ray.cpp -fopenmp -lm

## cuda\_ray.cu

$ nvcc -o cuda\_ray cuda\_ray.cu -O2

# **Execution**

## openmp\_ray.cpp

$ ./openmp\_ray [number\_of\_threads]

## cuda\_ray.cu

$ ./cuda\_ray

# **Source Code**

## 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 option;

int x,y;

unsigned char\* bitmap;

double start\_time, end\_time;

srand(time(NULL));

if (argc!=2) {

printf("> a.out [option] \n");

printf("[option] 1~32: OpenMP using 1~32 threads\n");

printf("for example, '> a.out 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);

// Set number of threads

omp\_set\_num\_threads(no\_threads);

// Get start time

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

// Parallelize the nested for-loop

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

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

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

// Get end time

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

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

fclose(fp);

free(bitmap);

free(temp\_s);

// Print execution time

printf("OpenMP (%d threads) ray tracing: %.3lf sec\n", no\_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>

#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;

// Set hit method to be executed on device

\_\_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;

}

};

// Set kernel method to be executed on device

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

{

// Calculate x and y from built-in variables

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);

//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;

unsigned char \*device\_bitmap; // Bitmap on device

Sphere \*temp\_s;

Sphere \*device\_temp\_s; // Sphere on device

clock\_t start\_time, end\_time;

srand(time(NULL));

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

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

// Allocate sphere memory on device

cudaMalloc((void \*\*)&device\_temp\_s, 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);

// Allocate bitmap memory on device

cudaMalloc((void \*\*)&device\_bitmap, sizeof(unsigned char) \* DIM \* DIM \* 4);

// Copy generated spheres to device

cudaMemcpy(device\_temp\_s, temp\_s, sizeof(Sphere) \* SPHERES, cudaMemcpyHostToDevice);

// Set blocks and threads

dim3 blocks(DIM / 32, DIM / 32, 1);

dim3 threads(32, 32, 1);

start\_time = clock();

// Execute kernel function

kernel<<<blocks, threads>>>(device\_temp\_s, device\_bitmap);

// Wait for device

cudaDeviceSynchronize();

end\_time = clock();

// Copy calculated spheres from device

cudaMemcpy(bitmap, device\_bitmap, sizeof(unsigned char) \* DIM \* DIM \* 4, cudaMemcpyDeviceToHost);

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

// Free device memory

cudaFree(device\_temp\_s);

cudaFree(device\_bitmap);

free(temp\_s);

free(bitmap);

fclose(fp);

// Print execution time

printf("CUDA ray tracing: %.3lf sec\n", (end\_time-start\_time)/(double)1000);

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

return 0;

}

# **Output Results**

result.ppm was converted into result.png by ffmpeg ($ *ffmpeg -i result.ppm result.png*)

## openmp\_ray

A screenshot of a computer

Description automatically generatedA picture containing blur, light, colorfulness

Description automatically generated

## cuda\_ray

A screenshot of a computer

Description automatically generated

A picture containing blur, colorfulness, light, sphere

Description automatically generated

# **Experimental Results**

## OpenMP

A picture containing screenshot, line

Description automatically generated

Figure 1. Error Bar Graph showing Execution Time of OpenMP with Various Numbers of Threads (10-Fold).

Table 1. Table showing Average Execution Time of OpenMP with Various Numbers of Threads (10-Fold).

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Number of Threads | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 |
| Execution Time (sec) | 0.4987 | 0.2731 | 0.1463 | 0.1054 | 0.0931 | 0.0907 | 0.0889 | 0.0886 | 0.0888 |

OpenMP raytracing implementation was tested using 1, 2, 4, 6, 8, 10, 12, 14, and 16 threads. When the number of threads increases, execution time decreases significantly by utilizing multiple CPU cores. The execution time converges when the number of threads is about 10. This is because the physical core is limited to 12 cores.

## CUDA

A picture containing screenshot, rectangle, square, design

Description automatically generated

Figure 2. Bar Graph showing Execution Times of Single Thread and CUDA Implementation (10-Fold).

CUDA version of raytracing was implemented by utilizing CUDA functionalities. The core ‘kernel’ function was executed in multiple GPU cores parallelly. The number of threads in a block was set as 32 \* 32 \* 1, where the number of blocks in a grid was set as (DIM/32) \* (DIM/32) \* 1, dividing total DIM \* DIM calculations.

CUDA raytracing implementation was tested by comparing execution time with the single thread version. Compared to the single thread version, the execution time of the CUDA version was shorter, where the single thread version took 1.4707 seconds, and the CUDA version took 1.1994 seconds on average.