Project 4, problem 2

Table of Contents

Installation

Requirements

- C++ compiler (g++, clang++, msvc++)
- OpenMp
- CUDA
- XMake
- Git

Installation

```
git clone git@github.com:GlassAlo/CAU_Multicore.git
cd CAU_Multicore/proj4
```

Compile with xmake (easier)

```
xmake f -m release && xmake -y
```

- **-m release** is used to compile the project in release mode, which is faster than debug mode.
- **xmake -y** is used to compile the project. The **-y** flag is used to skip the confirmation prompt.
- **xmake** will create a **bin** folder with the executables inside.

Compile manually

```
cd problem2
g++ -O3 -fopenmp omp_pi_one.cpp -o openmp_int
nvcc -O3 -arch=sm_75 thrust_ex.cu -o cuda_integral
```

Usage

```
./bin/openmp_int
./bin/cuda_integral
```

Test

🖥️ Hardware Specifications

- **OS**: Garuda Linux Broadwing x86_64
- **Kernel**: 6.13.8-zen1-1-zen
- **CPU**: AMD Ryzen 9 5900HS with Radeon Graphics (16) @ 4.680GHz
  - Cores 8
    - Uniform core design
  - Threads 16
  - Base clock 3.0GHz
  - Max boost clock up to 4.6GHz
  - L3 cache 16MB
  - Memory PCIe 3.0
  - Supports Simultaneous Multithreading (SMT), with each cores supporting two threads
- **Integrated GPU**: AMD ATI Radeon Vega Series / Radeon Vega Mobile Series
- **Discrete GPU**: NVIDIA GeForce RTX 3080 Mobile / Max-Q (8GB/16GB)
- **RAM**: 32GB
- **Disk**: 1TB SSD
- **Shell**: zsh

- Using Arch Linux comes with a cost, the CPU / GPU drivers might not be very efficient, stable or up to date.

Thrust_ex.cu

🧪 Source Code

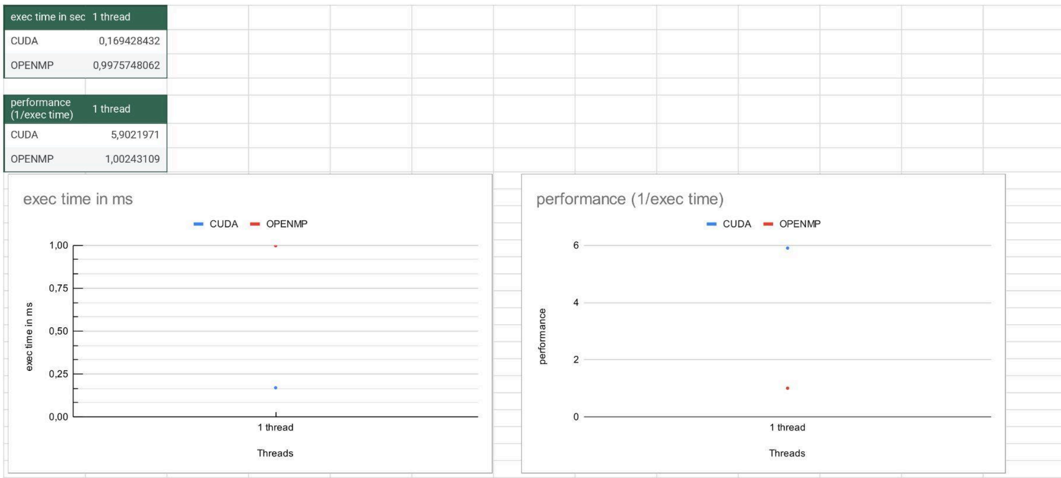
```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <cuda_runtime.h>

#define SPHERES 20
#define INF     2e10f
#define DIM     2048
#define rnd(x)  (x * rand() / RAND_MAX)

struct Sphere
{
       float r, g, b;
       float radius;
       float x, y, z;

       __device__ float hit(float ox, float oy, float *n) const
       {
           float dx = ox - x;
           float dy = oy - y;
           float rr = radius * radius;
           float d2 = dx * dx + dy * dy;
           if (d2 < rr) {
               float dz = sqrtf(rr - d2);
               *n = dz / radius;
               return dz + z;
           }
           return -INF;
       }
};

// Move spheres to constant memory for faster cached access
__constant__ Sphere d_spheres[SPHERES];

__global__ void render_kernel(unsigned char *ptr)
{
   int x = blockIdx.x * blockDim.x + threadIdx.x;
```

```
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    if (x >= DIM || y >= DIM)
        return;

    int offset = x + y * DIM;
    float ox = x - DIM * 0.5f;
    float oy = y - DIM * 0.5f;

    float r = 0, g = 0, b = 0;
    float maxz = -INF;

    for (int i = 0; i < SPHERES; i++) {
        float n;
        float t = d_spheres[i].hit(ox, oy, &n);
        if (t > maxz) {
            maxz = t;
            r = d_spheres[i].r * n;
            g = d_spheres[i].g * n;
            b = d_spheres[i].b * n;
        }
    }

    // Use __saturatef to clamp color values and avoid branching
    ptr[offset * 4 + 0] = (unsigned char) (__saturatef(r) * 255.0f);
    ptr[offset * 4 + 1] = (unsigned char) (__saturatef(g) * 255.0f);
    ptr[offset * 4 + 2] = (unsigned char) (__saturatef(b) * 255.0f);
    ptr[offset * 4 + 3] = 255;
}

void ppm_write(const unsigned char *bitmap, int xdim, int ydim, FILE
*fp)
{
    fprintf(fp, "P3\n%d %d\n255\n", xdim, ydim);
    for (int y = 0; y < ydim; y++) {
        for (int x = 0; x < xdim; x++) {
            int 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[])
{
    char *filename = "result.ppm";
    if (argc > 1) {
        filename = argv[1];
    }
    srand(time(NULL));

    Sphere spheres_h[SPHERES];
    for (int i = 0; i < SPHERES; i++) {
        spheres_h[i].r = rnd(1.0f);
        spheres_h[i].g = rnd(1.0f);
        spheres_h[i].b = rnd(1.0f);
        spheres_h[i].x = rnd(2000.0f) - 1000;
        spheres_h[i].y = rnd(2000.0f) - 1000;
        spheres_h[i].z = rnd(2000.0f) - 1000;
        spheres_h[i].radius = rnd(200.0f) + 40;
    }

    // Copy host spheres to device constant memory
    cudaMemcpyToSymbol(d_spheres, spheres_h, sizeof(Sphere) *
SPHERES);

    unsigned char *bitmap_h = nullptr;
    unsigned char *bitmap_d = nullptr;
    // Allocate device memory for image buffer
    cudaMalloc((void **) &bitmap_d, DIM * DIM * 4);
    // Allocate pinned (page-locked) host memory for image buffer
    cudaMallocHost((void **) &bitmap_h, DIM * DIM * 4);

    // Use 16x16 threads per block (256 threads)
    dim3 threads(16, 16);
    // Compute grid size to cover entire image
    dim3 blocks((DIM + threads.x - 1) / threads.x, (DIM + threads.y -
1) / threads.y);

    clock_t start = clock();

    render_kernel<<<blocks, threads>>>(bitmap_d);

    clock_t stop = clock();
```

```
    double duration = ((double) (stop - start)) / CLOCKS_PER_SEC;

    // Ensure all threads are finished
    cudaDeviceSynchronize();

    // Copy rendered image from device to host
    cudaMemcpy(bitmap_h, bitmap_d, DIM * DIM * 4,
cudaMemcpyDeviceToHost);

    FILE *fp = fopen(filename, "w");
    if (fp) {
        ppm_write(bitmap_h, DIM, DIM, fp);
        fclose(fp);
    }

    printf("CUDA ray tracing: %f sec\n", duration);
    printf("[%s] was generated.\n", filename);

    cudaFree(bitmap_d);
    cudaFreeHost(bitmap_h);
    return 0;
}
```

📊 Performance Chart



OpenMP output

```
~/Documents/CAU_Multi/proj4 git:(main)±9 (1.034s)
./bin/openmp_int
Execution Time : 0.9975748062sec
pi=3.1415926536
```

Thrust Result

```
~/Documents/CAU_Multi/proj4 git:(main)±9 (0.287s)
./bin/thrust_integral
Execution Time : 0.1693235500sec
pi=3.1415926536
```

🧠 Interpretation

Aspect	CUDA (Thrust)	OpenMP (1 thread)

Parallelism	Fully parallel on GPU (many threads)	No parallelism (pure serial)
API Used	`thrust::transform_reduce`	Manual for-loop (likely)
Execution Time (s)	0.169	0.998
Speedup	**~6× faster**	
Abstraction	High-level, STL-like (Thrust)	Low-level, manually controlled

- The execution time of **~0.169 s** reflects **highly efficient GPU parallelism** using **Thrust**.

- **OpenMP (1 thread)** does the same work sequentially, taking **~0.998 s**, because no parallelism is used.

🏁 Conclusion

- CUDA+Thrust showcases **ideal use of GPU parallelism for numerical integration**.
- Thrust handles automatically:
    - **Launches thousands of threads**
    - Handles **memory transfers, execution, and reduction**
- The performance is **~6× faster than a single-threaded CPU version**, confirming that for **data-parallel tasks**, GPU acceleration offers significant performance benefits with **minimal code complexity**.