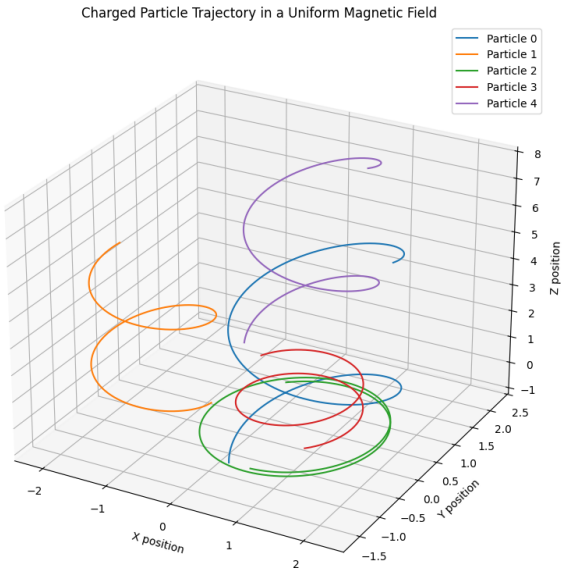
INTRODUCTION TO

High-performance test particle tracing code



Author:

El Ghaib, Adam

Contents

List of Figures

1 Introduction

1.1 Objective

This project aims to introduce the author into parallel ion tracing programming. To do so, three simple programs have been written. Two of them were written in CUDA language, where one of these uses Thrust library, to enhance development productivity. And the third one has been written in Python.

Moreover, the execution time of the three programs is compared to determine the optimal approach to build a high-performance test particle tracing simulator.

1.2 Considerations

To initially maintain the simulator simple:

- The electic **E(r,t)** and magnetic **B(r,t)** fields are considered constant through space and time.
- Elastic or inelactic collisions between particles are not taken into account.
- Only Lorentz force will act upon the particles and the 3D Newton's equation of motion is solved numerically using a fourth order Runge-Kutta algorithm.

1.3 Environment

The simulations have been executed in a Windows 11 operating system using Microsoft Visual Studio 2022. The GPU used was an Nvidia GeForce RTX 4070, the CPU was an Intel(R) Core(TM) i9-14900HX @ 2.20 GHz, and the RAM is of 32 GB.

2 Simulations

As mentioned before, to keep the programs as simple as possible, only the Lorentz force eq.1 will be considered.

$$m\frac{d\vec{\mathbf{v}}}{dt} = q(\vec{\mathbf{E}} + \vec{\mathbf{v}} \times \vec{\mathbf{B}}) \tag{1}$$

Furthermore, the mass is set to $\mathbf{m = 1\ kg}$, particles charge to $\mathbf{q = 1\ C}$, electric field to $\vec{\mathbf{E}} = \mathbf{0\ N/C}$ and magnetic field to $\vec{\mathbf{B}} = \mathbf{1\ T}$. Finally, the initial velocity components of each particle will be a random integer between -5 and 5 m/s.

2.1 Pure CUDA

The first program works as follows. First, basic CUDA and C/C++ libraries are included and some operator functions like addition (+), multiplication (∗), and others as (+ =), (− =) are defined, since in CUDA do not come by default at the device. Moreover, a Particle structure is created, only the position and velocity are fields of the structure as the other variables (q, m) will be the same for every particle.

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
#include <iostream>
#include <fstream>
#include <vector>

struct Particle {
    float3 position;
    float3 velocity;
};

__device__ float3 operator+(const float3 &a, const float3 &b) {
  return make_float3(a.x+b.x, a.y+b.y, a.z+b.z);
}
__device__ float3 operator*(const float3& a, const float3& b) {
    return make_float3(a.x * b.x, a.y * b.y, a.z * b.z);
}
```

Then a device function is written to calculate the Lorentz force:

```
__device__ float3 lorentz_force(float3 velocity, float3 position) {
    float3 E = make_float3(0.0f, 0.0f, 0.0f); // Electric field
    float3 B = make_float3(0.0f, 0.0f, 1.0f); // Magnetic field

    float3 v_cross_B = make_float3(
        velocity.y * B.z - velocity.z * B.y,
        velocity.z * B.x - velocity.x * B.z,
        velocity.x * B.y - velocity.y * B.x
    );

    float q = 1.0f; // Charge of the particle
    float m = 1.0f; // Mass of the particle
    return (q/m) * (E + v_cross_B);
}
```

The kernel function that will be executed at each thread is a fourth-order Runge-Kutta algorithm. Which gives a decent approximation to the solution of the differential equation of motion eq.1.

```
// CUDA kernel to update particle positions
__global__ void updateParticles(Particle* particles, int numParticles, float dt
    ) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < numParticles) {
        Particle p = particles[idx];

        float3 k1_v = lorentz_force(p.velocity, p.position);
        float3 k1_p = p.velocity;

        float3 temp_velocity = p.velocity + 0.5f * k1_v * dt;
        float3 temp_position = p.position + 0.5f * k1_p * dt;
        float3 k2_v = lorentz_force(p.velocity + 0.5f * k1_v * dt, p.position +
            0.5f * k1_p * dt);
        float3 k2_p = temp_velocity;

        temp_velocity = p.velocity + 0.5f * k2_v * dt;
        temp_position = p.position + 0.5f * k2_p * dt;
        float3 k3_v = lorentz_force(p.velocity + 0.5f * k2_v * dt, p.position +
            0.5f * k2_p * dt);
        float3 k3_p = temp_velocity;

        temp_velocity = p.velocity + k3_v * dt;
        temp_position = p.position + k3_p * dt;
```

```
        float3 k4_v = lorentz_force(p.velocity + k3_v * dt, p.position + k3_p *
            dt);
        float3 k4_p = temp_velocity;

        p.velocity += (k1_v + 2.0f * k2_v + 2.0f * k3_v + k4_v) * (dt / 6.0f);
        p.position += (k1_p + 2.0f * k2_p + 2.0f * k3_p + k4_p) * (dt / 6.0f);

        particles[idx] = p;
    }
}
```

In the main() function, the number of ions, simulation time, and time steps are set. Then two vectors with the structure particle are created, one is initialized with initial positions and velocities at the host (CPU) and the other copies this data and allocates it to the device (GPU) memory with cudaMalloc(). In addition, a thrid vector is created to store the xyz positions of the particles to later be exported as a binary file and plot it in Python using matplotlib.

```
    const int numParticles = 10;
    const int numTimesteps = 10000;
    const float dt = 0.001f;

    // Vector to store positions for plotting
    std::vector<float> positions;
    positions.reserve(numParticles * numTimesteps * 3);

    // Create host and device vectors
    Particle* h_particles = new Particle[numParticles];
    Particle* d_particles;
    cudaMalloc(&d_particles, numParticles * sizeof(Particle));

    // Initialize particles
    for (int i = 0; i < numParticles; ++i) {
        h_particles[i].position = make_float3(1.0f, 0.0f + i, 0.0f);
        h_particles[i].velocity = make_float3(i, 1.0f, 0.0f); // Constant
            velocity
    }

    // Copy particles to device
    cudaMemcpy(d_particles, h_particles, numParticles * sizeof(Particle),
        cudaMemcpyHostToDevice);
```

Next, the number of blocks and threads that are going to be executed parallelly in the device is defined according to the number of simulated particles. The kernel function will be executed "$numTimesteps = 10000$" times. For each time step, the result of the __global__ function is copied back to the host vector and for each particle their positions are saved in the *positions* vector. Finally, the memory is cleared and the positions file is exported.

```
    int blockSize = 256;
    int numBlocks = (numParticles + blockSize - 1) / blockSize;

    // Run simulation for numTimesteps
    for (int t = 0; t < numTimesteps; ++t) {
        updateParticles << < numBlocks, blockSize >> > (d_particles,
            numParticles, dt);
        cudaMemcpy(h_particles, d_particles, numParticles * sizeof(Particle),
            cudaMemcpyDeviceToHost);

        // Store positions for plotting
        for (int i = 0; i < numParticles; ++i) {
            positions.push_back(h_particles[i].position.x);
            positions.push_back(h_particles[i].position.y);
            positions.push_back(h_particles[i].position.z);
        }
    }

    // Free memory
    cudaFree(d_particles);
    delete[] h_particles;

    // Save positions to a file
    const char* path = "C:/Users/adame/OneDrive/Escritorio/CUDA/
        Particle_Tracing/.venv/particle_positions.bin";
    std::ofstream outFile(path, std::ios::binary);
    outFile.write(reinterpret_cast<char*>(positions.data()), positions.size() *
        sizeof(float));
    outFile.close();

    return 0;
```

The result of the simulation is shown in the next figure fig.1.

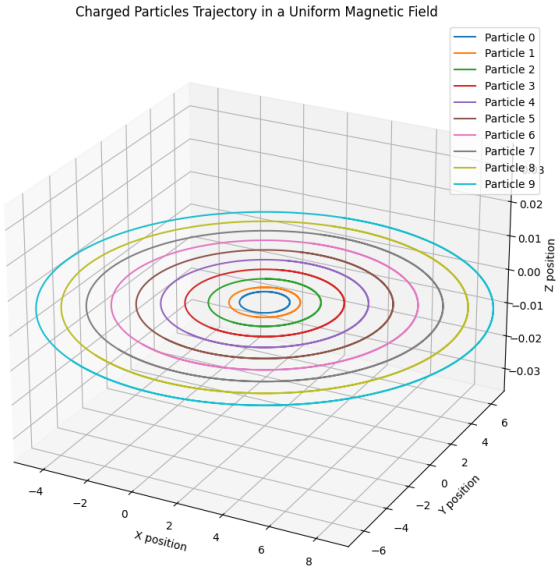


Figure 1: Trajectory of 10 particles simulated in CUDA.

2.2 Thrust library

The Thrust library is a high-level parallel algorithms library in CUDA, designed to resemble the C++ Standard Template Library (STL). Thrust enables developers to leverage GPU acceleration without needing to manage low-level details of CUDA programming explicitly [2].

Nonetheless, using Thrust can sometimes lead to longer compilation times due to the heavy use of templates and metaprogramming.

The main differences between this program and the previous one lie in main(). Here, the data translation between the host and device vectors can be done more intuitively. Furthermore, a built-in random number generator has been used to give random initial velocities to each particle.

```
    thrust::host_vector<Particle> h_particles(numParticles);
    // Initialize particles with random positions and velocities
    thrust::default_random_engine rng;
    thrust::uniform_real_distribution<float> dist(-5.0f, 5.0f);
    for (int i = 0; i < numParticles; ++i) {
        h_particles[i].position = make_float3(0.0f, 0.0f, 0.0f);
        h_particles[i].velocity = make_float3(dist(rng), dist(rng), dist(rng));
    }
    thrust::device_vector<Particle> d_particles = h_particles;
    for (int t = 0; t < numTimesteps; ++t) {
        thrust::for_each(thrust::device, d_particles.begin(), d_particles.end()
            , UpdateParticles(dt));
        thrust::copy(d_particles.begin(), d_particles.end(), h_particles.begin
            ());
        for (int i = 0; i < numParticles; ++i) {
            positions.push_back(h_particles[i].position.x);
            positions.push_back(h_particles[i].position.y);
            positions.push_back(h_particles[i].position.z);
        }
    }
```

The result of this simulation is shown in the next figure fig.2.

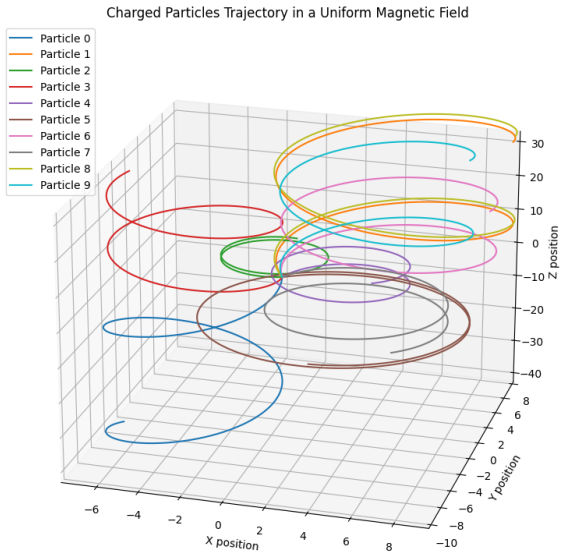


Figure 2: Trajectory of 10 particles simulated in CUDA using Thrust library.

2.3 Python

Finally, a Python simulation has also been written to compare its execution time, in section 3 with the last two simulations. The code will not be explained in this report, but it can be found in appendix A.1.

3 Performance

In this section, the execution times of the three proposed programs are compared. It is worth mentioning that the programs have been slightly modified to only measure the elapsed time that took for them to run the threads and simulate the particles state, that is to say, for the first program (2.1), the time between lines 4 and 8. Thus, the time used to initialize the vectors or copy the data and export it into a file has not been taken into account.

The results are shown in the next figure fig.3.

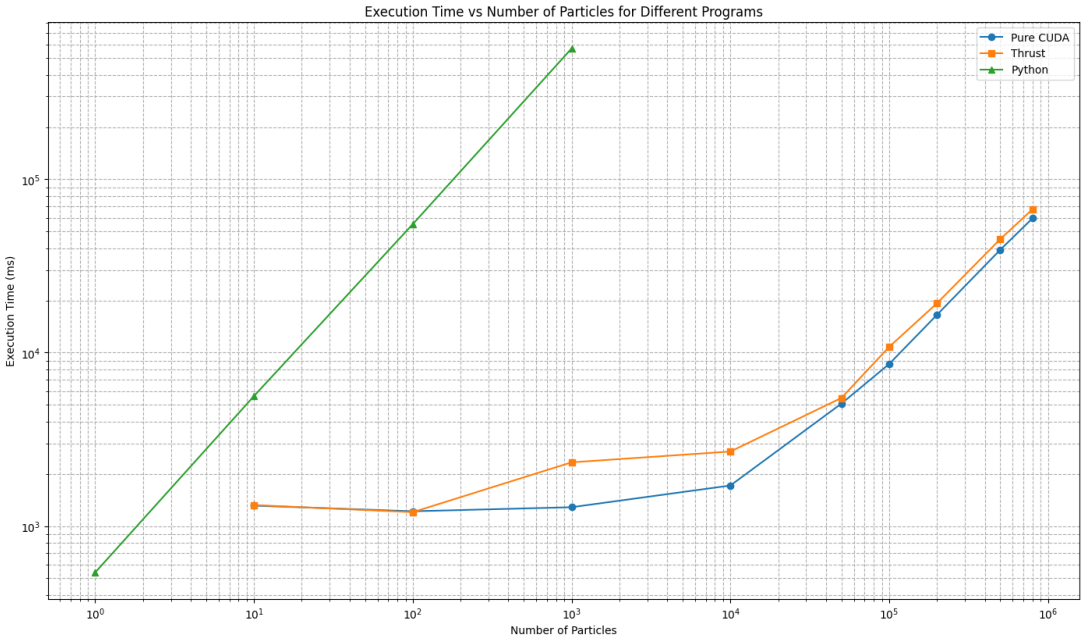


Figure 3: Execution time in *ms* vs the *number of particles* for each program.

It can be noticed that when the number of particles n is low, e.g $n \lessapprox 20$, the python program is the fastest as it uses the high-frequency cores of the CPU. However, the clear superiority of the GPU when the number of particles is increased does not take long to appear. And at $n = 1000$ the CUDA code is already more than **400x faster**.

On the other hand, the execution time difference between the pure CUDA code and the one using the Thrust library is not very remarkable. It indeed takes more time for the Thrust program to compile (5s over 2s of the pure CUDA) and to execute. Nevertheless, the simulations have omitted the effect of collisions and have been executed under constant fields, which greatly simplified the programs and shared memory processes between cores. For a more complex simulation, the slightly higher execution time cost over easier memory management for the developer that Thrust offers, could be worth it.

4 Conclusion

With all the data previously presented, it can be concluded that the programs written in CUDA perform remarkably better than the Python one. Moreover, for a higher complexity simulation, the Thrust library could facilitate the developer, memory management between cores and, make the program more understandable, at the expense of a slightly higher execution time.

Finally, as future steps for this didactic project. A variable magnetic field could be added, to do so and maintain the magnetic field divergence null, Chebyshev polynomials would be used. Furthermore, the Focker-Planck theory could also be used to implement the effect of elastic collisions. An elaborated CUDA particle simulator that would be followed as an example can be found in [1].

References

[1] C.F. Clauser, R. Farengo, H.E. Ferrari. Focus: A full-orbit cuda solver for particle simulations in magnetized plasmas. DOI: https://doi.org/10.1016/j.cpc.2018.07.018, year: 2018.

[2] Nvidia. Thrust library. https://developer.nvidia.com/thrust.

A Appendix

A.1 Python Code

Particle simulator written in Python which uses threads.

```
import numpy as np
import matplotlib.pyplot as plt
import threading
import time

# Constants
q = 1.0  # Charge of the particle
m = 1.0  # Mass of the particle
Bz = 1.0  # Magnetic field in the z-direction

# Time parameters
t0 = 0.0  # Initial time
tf = 10.0  # Final time
dt = 0.001  # Time step
num_steps = int((tf - t0) / dt)  # Number of time steps

def lorentz_force(v, B):
    return q * np.cross(v, B)

def rk4_step(r, v, dt):
    B = np.array([0, 0, Bz])
    k1_r = v
    k1_v = lorentz_force(v, B) / m

    k2_r = v + 0.5 * dt * k1_v
    k2_v = lorentz_force(v + 0.5 * dt * k1_v, B) / m

    k3_r = v + 0.5 * dt * k2_v
    k3_v = lorentz_force(v + 0.5 * dt * k2_v, B) / m

    k4_r = v + dt * k3_v
    k4_v = lorentz_force(v + dt * k3_v, B) / m

    r_next = r + (dt / 6) * (k1_r + 2*k2_r + 2*k3_r + k4_r)
    v_next = v + (dt / 6) * (k1_v + 2*k2_v + 2*k3_v + k4_v)

    return r_next, v_next

def simulate_particle(index, r0, v0, positions_all, velocities_all):
    r = r0
    v = v0
```

```
    positions = np.zeros((num_steps, 3))
    velocities = np.zeros((num_steps, 3))

    for i in range(num_steps):
        positions[i] = r
        velocities[i] = v

        r, v = rk4_step(r, v, dt)

    positions_all[index] = positions
    velocities_all[index] = velocities

# Function to measure execution time for a given number of particles
def measure_execution_time(num_particles):
    positions_all = np.zeros((num_particles, num_steps, 3))
    velocities_all = np.zeros((num_particles, num_steps, 3))

    # Generate unique initial conditions for each particle
    initial_conditions = [(np.array([1.0, 0.0, 0.0]) + 0.1*np.random.randn(3),
                           np.array([0.0, 1.0, 0.0]) + 0.1*np.random.randn(3))
                              for _ in range(num_particles)]

    # Measure start time
    start_time = time.time()

    # Create threads for each particle
    threads = []
    for i in range(num_particles):
        r0, v0 = initial_conditions[i]
        thread = threading.Thread(target=simulate_particle, args=(i, r0, v0,
            positions_all, velocities_all))
        threads.append(thread)
        thread.start()

    # Wait for all threads to finish
    for thread in threads:
        thread.join()

    # Measure end time
    end_time = time.time()

    # Calculate and return the execution time
    execution_time = end_time - start_time
    return execution_time

# Number of particles to test
num_particles_list = [1, 10, 100, 1000, 10000]
```

```
87 execution_times = []
88
89 # Measure execution time for each number of particles
90 for num_particles in num_particles_list:
91     execution_time = measure_execution_time(num_particles)
92     execution_times.append(execution_time)
93     print(f"Execution Time for {num_particles} particles: {execution_time:.2f}
           seconds")
94
95 # Plotting the results
96 plt.figure(figsize=(10, 5))
97 plt.plot(num_particles_list, execution_times, marker='o')
98 plt.xlabel('Number of Particles')
99 plt.ylabel('Execution Time (seconds)')
100 plt.title('Execution Time vs Number of Particles')
101 plt.grid(True)
102 plt.show()
```