Parallel Implementation of N Body Simulation
Sequential, OpenMP, and CUDA

Ali K. Rahimian

November 30, 2025

1 Introduction

In this project an N body simulation is implemented in three variants:

- a sequential C++ version,
- an OpenMP based CPU version, and
- a CUDA based GPU version.

Bodies interact by Newtonian gravitation in a two dimensional unit box with reflective boundaries. At each time step all pairwise forces are computed, velocities and positions are updated, and wall reflections are applied.

2 Methodology

2.1 Physical model

For bodies i and j,

$$\mathbf{F}_{ij} = G\frac{m_i m_j}{(r_{ij}^2 + \varepsilon)^{3/2}}\mathbf{r}_{ij},$$

with $G = 1.0$, softening $\varepsilon = 10^{-4}$, and $\mathbf{r}_{ij} = \mathbf{x}_j - \mathbf{x}_i$. The integration scheme is

$$\begin{aligned}\mathbf{v}_i^{t+\Delta t} &= \mathbf{v}_i^t + \mathbf{a}_i^t\Delta t,\\ \mathbf{x}_i^{t+\Delta t} &= \mathbf{x}_i^t + \mathbf{v}_i^{t+\Delta t}\Delta t,\end{aligned}$$

with $\Delta t = 0.01$. Positions are clamped to $[0, 1]$ and the corresponding velocity component is flipped on boundary hits.

2.2 Data structures

All versions use a `Body` structure with position `x, y`, velocity `vx, vy` and mass `m`. The sequential and OpenMP versions use an array of structs. The CUDA version uses separate arrays for positions, velocities and masses for better memory access on the device.

2.3 Sequential implementation

The sequential version uses a nested loop with $\mathcal{O}(N^2)$ force computation per step. For each body i all bodies j are visited to accumulate the force, then all bodies are updated in a second loop.

2.4 OpenMP implementation

The OpenMP version keeps the same physics and data layout. The outer loop over bodies i is parallelized with

```
#pragma omp parallel for schedule(static)
```

Each thread computes the force for its own body index and writes to its own entries in `fx` and `fy`. The integration loop remains serial but is inexpensive compared to the force calculation. The number of threads is controlled using a command line argument and `omp_set_num_threads`.

2.5 CUDA implementation

In the CUDA version one CUDA thread is mapped to one body. Each kernel call

- loads the position and mass of body i,
- loops over all bodies in device memory and accumulates the force,
- updates velocity and position and applies wall reflections.

On the host, arrays are allocated and initialized, copied to device memory, and then used in a time stepping loop that repeatedly invokes the kernel. After the simulation, the final state is copied back to the host and written to a file. A block size of 256 threads and grid size $(N + 255)/256$ are used.

3 Performance report

3.1 Setup

Performance was measured for

- $N \in \{64, 512, 1024, 2048, 4096\}$,
- 1000 time steps for these scaling tests.

The sequential and OpenMP codes are compiled with `-O3`. The OpenMP code also uses `-fopenmp`. The CUDA code is compiled with `nvcc -O3`. All CPU results use double precision, while the CUDA version uses single precision.

Table 1: Execution time (seconds) for sequential, OpenMP (8 threads) and CUDA, 1000 steps.

N	Sequential (1 thread)	OpenMP (8 threads)	CUDA
64	0.02365	0.01760	0.01145
512	1.25299	0.20392	0.03489
1024	4.99316	0.70256	0.06615
2048	19.93950	2.62902	0.12617
4096	79.74770	10.24720	0.22465

3.2 Execution time comparison

Table 1 shows the execution time in seconds for the three implementations and 1000 steps. To explore larger problem sizes and to estimate what fits within one hour on the GPU, several larger CUDA cases were also run. These are summarized in Table 2. The OpenMP implementation provides a clear speedup over the sequential version, and the CUDA version is significantly faster than either CPU version for all tested sizes.

Table 2: CUDA execution time for larger simulations.

N	Steps	CUDA time (s)	Notes
10 000	1 000	0.573022	exploratory scaling run
20 000	1 000	1.38926	exploratory scaling run
20 000	2 000	2.77037	step scaling test
30 000	2 000	4.50866	larger scaling test
100 000	5 000	126.952	initial large run
100 000	140 000	3575.43	largest run under one hour

3.3 Scaling with threads and bodies

To study thread scaling the OpenMP implementation was run with 1, 2, 4 and 8 threads. For each N the runtime drops rapidly from 1 to 2 and to 4 threads, with diminishing returns from 4 to 8 threads due to overhead and memory bandwidth limits. Figure 1 shows execution time versus number of threads for several body counts.

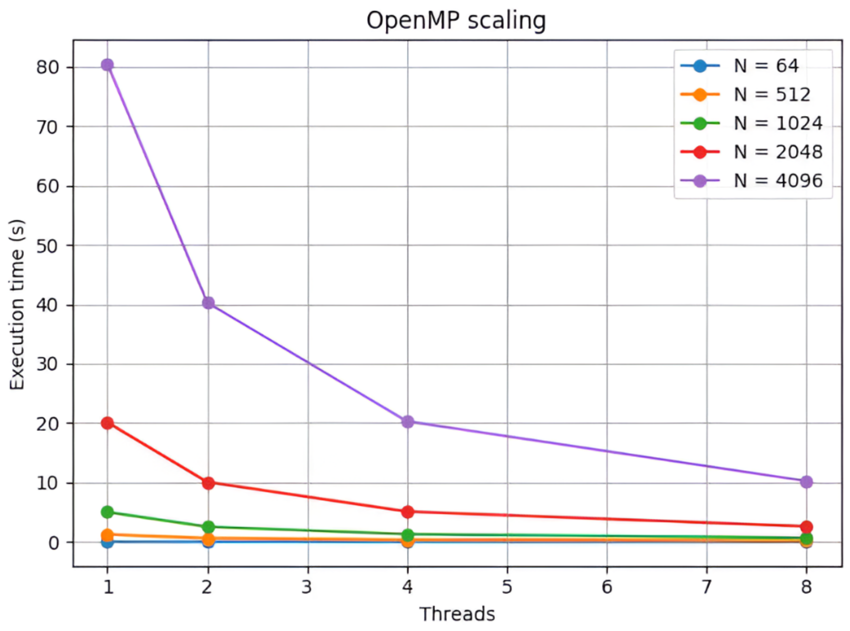

Figure 1: OpenMP scaling: execution time vs threads for different N.

Figure 2 shows execution time versus N for the three implementations. All curves grow roughly quadratically in N, as expected, but the CUDA curve has a much smaller constant factor.

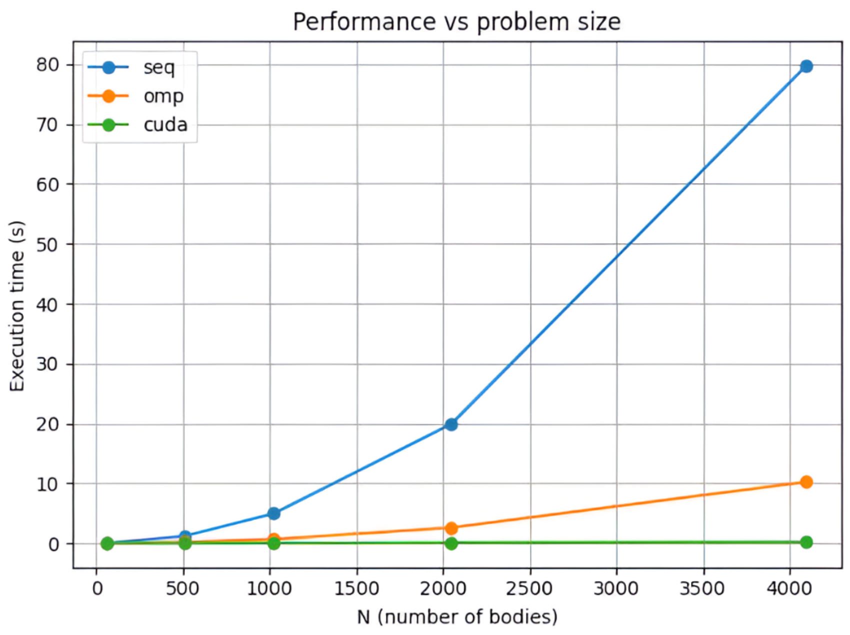

Figure 2: Execution time vs number of bodies N for sequential, OpenMP and CUDA.

3.4 Largest simulation within one hour

For the largest simulation that can run within one hour starting from at least 5000 bodies, the CUDA implementation was used and both N and the number of time steps were increased.

Based on the exploratory runs in Table 2, a final configuration with

- $N = 100\,000$ bodies,
- $140\,000$ time steps

was selected.

This run completed in

$$\text{CUDA time} = 3575.43 \text{ s}$$

which is about 59 minutes and 35 seconds, slightly below one hour. The total work is

$$100\,000 \times 140\,000 = 1.4 \times 10^{10} \text{ body steps.}$$

3.5 Summary of performance

In summary:

- The sequential implementation becomes slow as N grows, consistent with the $\mathcal{O}(N^2)$ force calculation.
- The OpenMP version exploits multi core CPUs and reduces runtime by roughly an order of magnitude at 8 threads for the largest tested N.
- The CUDA version achieves the lowest runtimes and enables very large simulations, including $100\,000$ bodies with $140\,000$ steps within one hour.

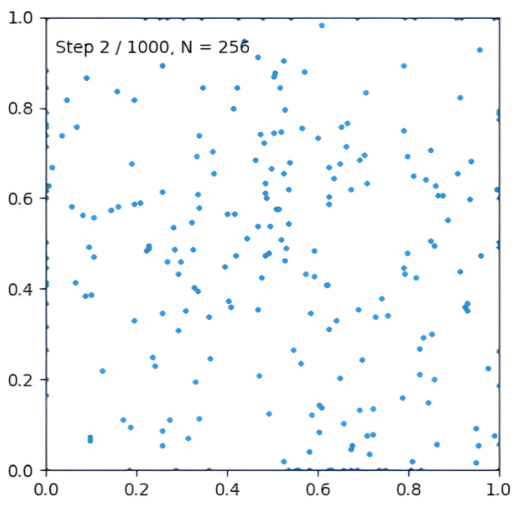

Figure 3: Snapshot of the visualization for 256 bodies and 1000 steps (animated GIF submitted separately).

4 Visualization

For visualization the sequential implementation was run with $N = 256$ bodies and 1000 steps, with trajectory output enabled. A Python script using Matplotlib reads the trajectory file and produces an animated GIF.

The animated GIF file

- `outputs/vis/nbody_N256_steps1000.gif`

is submitted separately as the visualization artifact. A static snapshot is shown in Figure 3 for reference.

5 Conclusion

An N body simulation has been implemented in three variants: sequential C++, OpenMP with shared memory parallelism and CUDA on the GPU. The measured results show that OpenMP provides solid speedups over the sequential baseline and that the CUDA version offers the best performance and scales to much larger problems.

A Sequential C++ implementation

Listing 1: Sequential N body implementation (nbody_seq.cpp)

```cpp
#include <cmath>
#include <chrono>
#include <filesystem>
#include <fstream>
#include <iostream>
#include <random>
#include <string>
#include <vector>

namespace fs = std::filesystem;

struct Body {
    double x, y;
    double vx, vy;
    double m;
};

const double G = 1.0;
const double DT = 0.01;
const double SOFTENING = 1e-4;
const double BOX_SIZE = 1.0;

void init_bodies(std::vector<Body> &bodies, unsigned int seed = 42) {
    std::mt19937 gen(seed);
    std::uniform_real_distribution<double> pos_dist(0.0, BOX_SIZE);
    std::uniform_real_distribution<double> vel_dist(-0.01, 0.01);
    std::uniform_real_distribution<double> mass_dist(0.5, 1.5);

    for (auto &b : bodies) {
        b.x = pos_dist(gen);
        b.y = pos_dist(gen);
        b.vx = vel_dist(gen);
        b.vy = vel_dist(gen);
        b.m = mass_dist(gen);
    }
}

void step_sequential(std::vector<Body> &bodies) {
    int n = static_cast<int>(bodies.size());
    std::vector<double> fx(n, 0.0), fy(n, 0.0);

    // Compute forces
    for (int i = 0; i < n; ++i) {
        double xi = bodies[i].x;
        double yi = bodies[i].y;
        double mi = bodies[i].m;
        double fxi = 0.0;
        double fyi = 0.0;

        for (int j = 0; j < n; ++j) {
            if (i == j) continue;
            double dx = bodies[j].x - xi;
            double dy = bodies[j].y - yi;
            double dist2 = dx * dx + dy * dy + SOFTENING;
            double invDist = 1.0 / std::sqrt(dist2);
            double invDist3 = invDist * invDist * invDist;
            double f = G * mi * bodies[j].m * invDist3;
            fxi += f * dx;
            fyi += f * dy;
        }
        fx[i] = fxi;
        fy[i] = fyi;
    }

    // Update velocities and positions, apply boundary reflections
    for (int i = 0; i < n; ++i) {
        double ax = fx[i] / bodies[i].m;
        double ay = fy[i] / bodies[i].m;

        bodies[i].vx += ax * DT;
        bodies[i].vy += ay * DT;
        bodies[i].x += bodies[i].vx * DT;
        bodies[i].y += bodies[i].vy * DT;

        if (bodies[i].x < 0.0) {
            bodies[i].x = 0.0;
            bodies[i].vx *= -1.0;
        } else if (bodies[i].x > BOX_SIZE) {
            bodies[i].x = BOX_SIZE;
            bodies[i].vx *= -1.0;
        }

        if (bodies[i].y < 0.0) {
            bodies[i].y = 0.0;
            bodies[i].vy *= -1.0;
        } else if (bodies[i].y > BOX_SIZE) {
            bodies[i].y = BOX_SIZE;
            bodies[i].vy *= -1.0;
        }
    }
}

int main(int argc, char **argv) {
    if (argc < 5) {
        std::cerr << "Usage: " << argv[0]
                  << " N num_steps output_dir write_trajectories(0 or 1)\n
                     ";
        return 1;
    }

    int N = std::stoi(argv[1]);
    int num_steps = std::stoi(argv[2]);
    std::string output_dir = argv[3];
    int write_traj = std::stoi(argv[4]);

    std::vector<Body> bodies(N);
    init_bodies(bodies);

    fs::path out_dir(output_dir);
    try {
        fs::create_directories(out_dir);
    } catch (const std::exception &e) {
        std::cerr << "Failed to create output directory: " << e.what() <<
           "\n";
        return 1;
    }

    std::string suffix = write_traj ? "traj" : "final";
    fs::path out_path =
        out_dir / ("seq_N" + std::to_string(N) +
                   "_steps" + std::to_string(num_steps) +
                   "_" + suffix + ".txt");

    std::ofstream ofs(out_path);
    if (!ofs) {
        std::cerr << "Error opening output file: " << out_path << "\n";
        return 1;
    }

    auto start = std::chrono::high_resolution_clock::now();

    for (int step = 0; step < num_steps; ++step) {
        step_sequential(bodies);

        if (write_traj) {
            for (int i = 0; i < N; ++i) {
                ofs << step << " " << i << " "
                    << bodies[i].x << " " << bodies[i].y << " "
                    << bodies[i].vx << " " << bodies[i].vy << "\n";
            }
        }
    }

    if (!write_traj) {
        for (int i = 0; i < N; ++i) {
            ofs << i << " "
                << bodies[i].x << " " << bodies[i].y << " "
                << bodies[i].vx << " " << bodies[i].vy << "\n";
        }
    }

    ofs.close();

    auto end = std::chrono::high_resolution_clock::now();
    double elapsed = std::chrono::duration<double>(end - start).count();
    std::cout << "Sequential time: " << elapsed << " s\n";
    std::cout << "Sequential output written to: " << out_path << "\n";

    return 0;
}
```

B OpenMP C++ implementation

Listing 2: OpenMP N body implementation (nbody_omp.cpp)

```
#include <cmath>
#include <chrono>
#include <filesystem>
#include <fstream>
#include <iostream>
#include <random>
#include <string>
#include <vector>
#include <omp.h>

namespace fs = std::filesystem;

struct Body {
    double x, y;
    double vx, vy;
    double m;
};

const double G = 1.0;
const double DT = 0.01;
const double SOFTENING = 1e-4;
const double BOX_SIZE = 1.0;

void init_bodies(std::vector<Body> &bodies, unsigned int seed = 42) {
    std::mt19937 gen(seed);
    std::uniform_real_distribution<double> pos_dist(0.0, BOX_SIZE);
    std::uniform_real_distribution<double> vel_dist(-0.01, 0.01);
    std::uniform_real_distribution<double> mass_dist(0.5, 1.5);

    for (auto &b : bodies) {
        b.x = pos_dist(gen);
        b.y = pos_dist(gen);
        b.vx = vel_dist(gen);
        b.vy = vel_dist(gen);
        b.m = mass_dist(gen);
    }
}

void step_omp(std::vector<Body> &bodies) {
    int n = static_cast<int>(bodies.size());
    std::vector<double> fx(n, 0.0), fy(n, 0.0);

    #pragma omp parallel for schedule(static)
    for (int i = 0; i < n; ++i) {
        double xi = bodies[i].x;
        double yi = bodies[i].y;
        double mi = bodies[i].m;
        double fxi = 0.0;
        double fyi = 0.0;

        for (int j = 0; j < n; ++j) {
            if (i == j) continue;
            double dx = bodies[j].x - xi;
            double dy = bodies[j].y - yi;
            double dist2 = dx * dx + dy * dy + SOFTENING;
            double invDist = 1.0 / std::sqrt(dist2);
            double invDist3 = invDist * invDist * invDist;
            double f = G * mi * bodies[j].m * invDist3;
            fxi += f * dx;
            fyi += f * dy;
        }
        fx[i] = fxi;
        fy[i] = fyi;
    }

    for (int i = 0; i < n; ++i) {
        double ax = fx[i] / bodies[i].m;
        double ay = fy[i] / bodies[i].m;

        bodies[i].vx += ax * DT;
        bodies[i].vy += ay * DT;
        bodies[i].x += bodies[i].vx * DT;
        bodies[i].y += bodies[i].vy * DT;

        if (bodies[i].x < 0.0) {
            bodies[i].x = 0.0;
            bodies[i].vx *= -1.0;
        } else if (bodies[i].x > BOX_SIZE) {
            bodies[i].x = BOX_SIZE;
            bodies[i].vx *= -1.0;
        }

        if (bodies[i].y < 0.0) {
            bodies[i].y = 0.0;
            bodies[i].vy *= -1.0;
        } else if (bodies[i].y > BOX_SIZE) {
            bodies[i].y = BOX_SIZE;
            bodies[i].vy *= -1.0;
        }
    }
}

int main(int argc, char **argv) {
    if (argc < 5) {
        std::cerr << "Usage: " << argv[0]
                  << " N num_steps output_dir num_threads\n";
        return 1;
    }

    int N = std::stoi(argv[1]);
    int num_steps = std::stoi(argv[2]);
    std::string output_dir = argv[3];
    int num_threads = std::stoi(argv[4]);

    omp_set_num_threads(num_threads);

    std::vector<Body> bodies(N);
    init_bodies(bodies);

    fs::path out_dir(output_dir);
    try {
        fs::create_directories(out_dir);
    } catch (const std::exception &e) {
        std::cerr << "Failed to create output directory: " << e.what() <<
            "\n";
        return 1;
    }

    fs::path out_path =
        out_dir / ("omp_N" + std::to_string(N) +
                   "_steps" + std::to_string(num_steps) +
                   "_t" + std::to_string(num_threads) + ".txt");

    std::ofstream ofs(out_path);
    if (!ofs) {
        std::cerr << "Error opening output file: " << out_path << "\n";
        return 1;
    }

    auto start = std::chrono::high_resolution_clock::now();

    for (int step = 0; step < num_steps; ++step) {
        step_omp(bodies);
    }

    for (int i = 0; i < N; ++i) {
        ofs << i << " "
            << bodies[i].x << " " << bodies[i].y << " "
            << bodies[i].vx << " " << bodies[i].vy << "\n";
    }

    ofs.close();

    auto end = std::chrono::high_resolution_clock::now();
    double elapsed = std::chrono::duration<double>(end - start).count();
    std::cout << "OpenMP time (" << num_threads << " threads): "
              << elapsed << " s\n";
    std::cout << "OpenMP output written to: " << out_path << "\n";

    return 0;
}
```

C CUDA implementation

Listing 3: CUDA N body implementation (`nbody_cuda.cu`)

```
#include <cuda_runtime.h>
#include <cmath>
#include <chrono>
#include <filesystem>
#include <fstream>
#include <iostream>
#include <random>
#include <string>
#include <cstdlib>

namespace fs = std::filesystem;

const float Gf = 1.0f;
const float DTf = 0.01f;
const float SOFTENINGf = 1e-4f;
const float BOX_SIZEf = 1.0f;

#define CUDA_CHECK(call)                                                        \
    do {                                                                        \
        cudaError_t err = call;                                                 \
        if (err != cudaSuccess) {                                               \
            std::cerr << "CUDA error at " << __FILE__ << ":" << __LINE__        \
                      << " - " << cudaGetErrorString(err) << "\n";              \
            std::exit(1);                                                       \
        }                                                                       \
    } while (0)

__global__
void nbody_step_kernel(float *x, float *y,
                       float *vx, float *vy,
                       const float *m,
                       int n, float dt,
                       float G, float softening,
                       float box_size) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= n) return;

    float xi = x[i];
    float yi = y[i];
    float mi = m[i];

    float fxi = 0.0f;
    float fyi = 0.0f;

    for (int j = 0; j < n; ++j) {
        if (j == i) continue;
        float dx = x[j] - xi;
        float dy = y[j] - yi;
        float dist2 = dx * dx + dy * dy + softening;
        float invDist = rsqrtf(dist2);
        float invDist3 = invDist * invDist * invDist;
        float f = G * mi * m[j] * invDist3;
        fxi += f * dx;
        fyi += f * dy;
    }

    float ax = fxi / mi;
    float ay = fyi / mi;

    float vxi = vx[i] + ax * dt;
    float vyi = vy[i] + ay * dt;
    float xi_new = xi + vxi * dt;
    float yi_new = yi + vyi * dt;

    if (xi_new < 0.0f) {
        xi_new = 0.0f;
        vxi *= -1.0f;
    } else if (xi_new > box_size) {
        xi_new = box_size;
        vxi *= -1.0f;
    }

    if (yi_new < 0.0f) {
        yi_new = 0.0f;
        vyi *= -1.0f;
    } else if (yi_new > box_size) {
        yi_new = box_size;
        vyi *= -1.0f;
    }

    x[i] = xi_new;
    y[i] = yi_new;
    vx[i] = vxi;
    vy[i] = vyi;
}

void init_bodies_host(float *x, float *y,
                      float *vx, float *vy,
                      float *m, int n,
                      unsigned int seed = 42) {
    std::mt19937 gen(seed);
    std::uniform_real_distribution<float> pos_dist(0.0f, BOX_SIZEf);
    std::uniform_real_distribution<float> vel_dist(-0.01f, 0.01f);
    std::uniform_real_distribution<float> mass_dist(0.5f, 1.5f);

    for (int i = 0; i < n; ++i) {
        x[i] = pos_dist(gen);
        y[i] = pos_dist(gen);
        vx[i] = vel_dist(gen);
        vy[i] = vel_dist(gen);
        m[i] = mass_dist(gen);
    }
}

int main(int argc, char **argv) {
    if (argc < 4) {
        std::cerr << "Usage: " << argv[0]
                  << " N num_steps output_dir\n";
        return 1;
    }

    int N = std::stoi(argv[1]);
    int num_steps = std::stoi(argv[2]);
    std::string output_dir = argv[3];

    size_t bytes = static_cast<size_t>(N) * sizeof(float);

    float *h_x = nullptr;
    float *h_y = nullptr;
    float *h_vx = nullptr;
    float *h_vy = nullptr;
    float *h_m = nullptr;

    h_x = static_cast<float*>(std::malloc(bytes));
    h_y = static_cast<float*>(std::malloc(bytes));
    h_vx = static_cast<float*>(std::malloc(bytes));
    h_vy = static_cast<float*>(std::malloc(bytes));
    h_m  = static_cast<float*>(std::malloc(bytes));

    if (!h_x || !h_y || !h_vx || !h_vy || !h_m) {
        std::cerr << "Host allocation failed\n";
        return 1;
    }

    init_bodies_host(h_x, h_y, h_vx, h_vy, h_m, N);

    float *d_x = nullptr;
    float *d_y = nullptr;
    float *d_vx = nullptr;
    float *d_vy = nullptr;
    float *d_m = nullptr;

    CUDA_CHECK(cudaMalloc(&d_x, bytes));
    CUDA_CHECK(cudaMalloc(&d_y, bytes));
    CUDA_CHECK(cudaMalloc(&d_vx, bytes));
    CUDA_CHECK(cudaMalloc(&d_vy, bytes));
    CUDA_CHECK(cudaMalloc(&d_m, bytes));

    CUDA_CHECK(cudaMemcpy(d_x, h_x, bytes, cudaMemcpyHostToDevice));
    CUDA_CHECK(cudaMemcpy(d_y, h_y, bytes, cudaMemcpyHostToDevice));
    CUDA_CHECK(cudaMemcpy(d_vx, h_vx, bytes, cudaMemcpyHostToDevice));
    CUDA_CHECK(cudaMemcpy(d_vy, h_vy, bytes, cudaMemcpyHostToDevice));
    CUDA_CHECK(cudaMemcpy(d_m, h_m, bytes, cudaMemcpyHostToDevice));

    int blockSize = 256;
    int gridSize = (N + blockSize - 1) / blockSize;

    auto start = std::chrono::high_resolution_clock::now();

    for (int step = 0; step < num_steps; ++step) {
        nbody_step_kernel<<<gridSize, blockSize>>>(
            d_x, d_y, d_vx, d_vy, d_m,
            N, DTf, Gf, SOFTENINGf, BOX_SIZEf
        );
        CUDA_CHECK(cudaGetLastError());
        CUDA_CHECK(cudaDeviceSynchronize());
    }

    auto end = std::chrono::high_resolution_clock::now();
    double elapsed = std::chrono::duration<double>(end - start).count();
    std::cout << "CUDA time: " << elapsed << " s\n";

    CUDA_CHECK(cudaMemcpy(h_x, d_x, bytes, cudaMemcpyDeviceToHost));
    CUDA_CHECK(cudaMemcpy(h_y, d_y, bytes, cudaMemcpyDeviceToHost));
    CUDA_CHECK(cudaMemcpy(h_vx, d_vx, bytes, cudaMemcpyDeviceToHost));
    CUDA_CHECK(cudaMemcpy(h_vy, d_vy, bytes, cudaMemcpyDeviceToHost));

    fs::path out_dir(output_dir);
    try {
        fs::create_directories(out_dir);
    } catch (const std::exception &e) {
        std::cerr << "Failed to create output directory: " << e.what() <<
            "\n";
        return 1;
    }

    fs::path out_path =
        out_dir / ("cuda_N" + std::to_string(N) +
                   "_steps" + std::to_string(num_steps) + ".txt");

    std::ofstream ofs(out_path);
    if (!ofs) {
        std::cerr << "Error opening output file: " << out_path << "\n";
        return 1;
    }

    for (int i = 0; i < N; ++i) {
        ofs << i << " "
            << h_x[i] << " " << h_y[i] << " "
            << h_vx[i] << " " << h_vy[i] << "\n";
    }
    ofs.close();

    std::cout << "CUDA output written to: " << out_path << "\n";

    CUDA_CHECK(cudaFree(d_x));
    CUDA_CHECK(cudaFree(d_y));
    CUDA_CHECK(cudaFree(d_vx));
    CUDA_CHECK(cudaFree(d_vy));
    CUDA_CHECK(cudaFree(d_m));

    std::free(h_x);
    std::free(h_y);
    std::free(h_vx);
    std::free(h_vy);
    std::free(h_m);

    return 0;
}
```