High-Performance Computing
Prof. Dr. Ivan Kisel
Robin Lakos, Akhil Mithran, Oddharak Tyagi

High-Performance Computer Architectures
Practical Course
WiSe 2024/25

Issued: 09.07.2025 / Due: 23.07.2025

Institute for Computer Science

Final Project

Introduction

In this project, you will implement a simulation of a simplified version of the N-body problem. The concept is straightforward: imagine N bodies (e.g., planets or stars) in the universe. Each body exerts a gravitational influence on all the other bodies. In our simplified case, we consider only gravity as the interaction between bodies. Every body has a mass, and the gravitational force it exerts on others depends on its mass and the distance to those bodies.

To make the application more presentable, we will make use of `SFML`, an external library that allows us to visualize the simulation nicely and fluently. We will provide all necessary information (e.g., required functions from `SFML` and constant values for simulation) on this exercise sheet.

Requirements

Please read the following exercise sheet carefully.

We expect that you submit a complete CMake project including a `CMakeLists.txt` with all necessary compiler flags, your source code, and other files that are necessary to compile or run your application.

Please do not submit any temporary files nor your build directory.

Exercise F.0: Installation of `SFML`

The simulation programmed during this exercise sheet is quite boring without proper visualization. Therefore, we will install the `SFML` library that allows us to render our simulation as a graphical application. The installation script is provided along with the exercise sheet and source code. To install the `SFML` library, run

```
./install_sfml.sh
```

In case of missing execution permissions, please run

```
chmod +x install_sfml.sh
```

or

```
chown <username>:<username> install_sfml.sh
```

to solve problems with file ownership. Then, try again to install the library.

After installing the `SFML` library, we can add `#include <SFML/Graphics.hpp>` in our source code to make use of it. Make use of the following compiler flags:

- `-lsfml-graphics`
- `-lsfml-window`
- `-lsfml-system`

Exercise F.1: Scalar Simplified N-Body Problem

i) Project Structure

As a first step to setup our project, we create the project structure for the required source code files. The minimal requirements of your project are shown below.

`NBody` (Project directory)

- `NBody.cpp`
- `src`
  - Body.cpp
- `include`
  - Body.h
- `<your_font>.ttf`

ii) The Body Class

Create a class or struct for `Body` instances to represent each body in the universe. At least the following properties for a body are necessary for the simulation:

- position in x
- position in y
- velocity in x
- velocity in y
- acceleration in x
- acceleration in y
- mass

iii) Application of Forces

Define the following constants in your source code.

```
constexpr float G = 1.f;
constexpr float dt = .1f;
constexpr float eps = 1e-1f;
constexpr size_t n_bodies = 5;
constexpr float center_mass = 1000.f;
```

Implement a function `compute_forces(std::vector<Body>& bodies, float G, float eps)` that will be used to compute the gravitational forces that each body exerts on all other bodies in the universe.

For the calculation, iterate over each body (i) in the universe and set its acceleration to 0. Then, for each body (i), iterate over all other bodies ($j \neq i$) in the universe and proceed as follows:

1) Calculate distances in x and y to the other body:

$$dx_{i,j} = x_j - x_i$$
$$dy_{i,j} = y_j - y_i$$

2) Calculate the cubed inverse distance, including the softening factor ϵ:

$$d^3_{\texttt{inv},i,j} = \left(\frac{1}{\sqrt{dx^2_{i,j} + dy^2_{i,j} + \epsilon^2}} \right)^3$$

3) Calculate the force applied from body B_j to body B_i:

$$f_{i,j} = G \cdot m_j \cdot d^3_{\texttt{inv},i,j}$$

4) Sum and update the acceleration in both directions for body B_i:

$$a_{x,i} \mathrel{+}= dx_{i,j} \cdot f_{i,j}$$
$$a_{y,i} \mathrel{+}= dy_{i,j} \cdot f_{i,j}$$

Note: Each body has no impact on itself, i.e., skip $j = i$.

iv) Updates of Body Positions

Implement a function `integrate_bodies(std::vector<Body>& bodies, float dt)` that is used to update the position and velocity of all bodies after each body has its new acceleration values.

$$v_{x,i} = v_{x,i} + a_{x,i} \cdot \Delta t$$
$$v_{y,i} = v_{y,i} + a_{y,i} \cdot \Delta t$$
$$x_i = x_i + v_{x,i} \cdot \Delta t$$
$$y_i = y_i + v_{y,i} \cdot \Delta t$$

v) Visualization with `SFML`

As a next step, prepare the rendering of the simulation by using the SFML library.

Define the following constant values in your source code.

```
constexpr float TARGET_FPS = 165.f;
const sf::Time FRAME_DURATION = sf::seconds(1.f / TARGET_FPS);
```

Define the window properties as follows. This can be done at the very beginning of your application. Then, adjust the path with a font of your choice (there is one in OLAT if you don't want to use your own).

```
const int WIDTH = 2560, HEIGHT = 1440;
sf::RenderWindow window(sf::VideoMode(WIDTH, HEIGHT), "N-Body Simulation");
sf::Font font;
if (!font.loadFromFile("<your_font>.ttf")) {
   std::cerr << "Failed to load font\n";
}
sf::Text fpsText("", font, 18);
fpsText.setFillColor(sf::Color::White);
fpsText.setPosition(10, 5);
```

If this is done, you can create your main loop that displays the current universe in the window.

```
sf::Clock frameClock;
sf::Clock fpsClock;
float lastFPS = 0.0f;

while (window.isOpen()) {
  sf::Event e;
  while (window.pollEvent(e)) if (e.type == sf::Event::Closed) window.close();

  compute_forces(bodies, G, eps);
  integrate_bodies(bodies, dt);

  window.clear(sf::Color::Black);

  // Drawing of bodies
  for (const Body& b : bodies) {
    sf::CircleShape circle(b.m > 50.0f ? 6 : 2);
    circle.setFillColor(mass_to_color(b.m));
    circle.setPosition(WIDTH / 2 + b.x, HEIGHT / 2 + b.y);
    circle.setOrigin(circle.getRadius(), circle.getRadius());
    window.draw(circle);
  }

  // FPS calculation and display
  float elapsed = fpsClock.restart().asSeconds();
  lastFPS = 1.0f / elapsed;
  fpsText.setString("FPS: " + std::to_string((int)lastFPS));
  window.draw(fpsText);

  window.display();

  sf::Time elapsed = frameClock.getElapsedTime();
  if (elapsed < FRAME_DURATION) sf::sleep(FRAME_DURATION - elapsed);
  frameClock.restart();
```

vi) Finalize Implementation

Add the following functions to your source code. You can adjust them as you like.

```
sf::Color mass_to_color(float m) {
  float norm = std::min(1.0f, m / 10.0f);
  return sf::Color(255 * norm, 50, 255 * (1 - norm));
}

float orbital_velocity_scalar(float M, float r) {
  return std::sqrt(1.0f * M / r); // G = 1.0 assumed
}
```

Additionally, think about the body initialization. Initialize the bodies in a separate function randomly in the universe, e.g. by using a random engine.

```
std::mt19937 rng(42);
std::uniform_real_distribution<float> angle_dist(0.0f, 2.0f * M_PI);
std::uniform_real_distribution<float> radius_dist(50.0f, std::min(width, height) / 2.f -
    20.f);
std::uniform_real_distribution<float> mass_dist(0.5f, 10.f);
```

and initialize the bodies with random positions and velocities. Make sure they are in a visible range of the window. Also, create another body with a heavier mass (e.g., $m = 1000$) in the center of the window for stability.

Exercise F.2: Parallel Simplified N-Body Problem

At this stage, you should have a fully functional sequential version of the simplified N-body problem. Increase the number of bodies to a larger number and you will quickly recognize that the simulation becomes slower and slower.

To tackle this problem, you are asked to write an OpenCL version of that simulation.

i) Update the Project Structure

`NBody` (Project directory)

- `NBody.cpp`
- `src`
  - Body.cpp
- `include`
  - Body.h
- `opencl`
  - NBody.cl
- `<your_font>.ttf`

ii) SOA Data Structure for Bodies

As a preparation for your OpenCL implementation, change the data structure that stores your bodies into a structure that follows the SOA (Struct of Arrays) principle. Here, instead of raw C-style arrays, use `std::vector<T>` for each body property.

This way, we can easily use the pointers to the underlying data for OpenCL.

iii) OpenCL Kernel

Write the kernel functions `compute_forces` and `integrate_bodies` both into an OpenCL file `NBody.cl`.

Initialize the OpenCL kernels and buffers once. Inside your main display loop, set the arguments for the functions and enqueue the kernel functions. Check if the queue is finished and readout the buffers for the rendering of bodies.

iv) Toroidal Universe

In many cases, depending on the bodies initial properties, bodies will fly out the window and disappear – at least visually.

To avoid that, modify your kernel function such that you create a toroidal world, such that bodies flying out on one side will re-enter the window on the opposite side.

Exercise F.3: Clean-Up

Before submitting your project, you want to make sure that you followed best practices.

Therefore, make sure that you, for example,

- used reasonable data types and data structures
- used only explicit type-casting
- used constness wherever reasonably possible
- tried to avoid unnecessary initializations
- prefer brace-initialization where applicable