CUDA-Based Parallel N-Body Simulation in C++

Erik Saule

Goal: Implement a parallel N-body simulation in C++ to compute the interactions between particles based on gravitational forces and simulate their motion using CUDA on a GPU. The simulation should be designed to run on the Centaurus using SLURM.

1 Overview of the N-Body Problem

The N-body problem involves predicting the individual motions of a group of particles that interact with each other through gravitational forces. This problem is foundational in astrophysics, computational physics, and numerical simulations.

Key aspects of the N-body problem:

- **Gravitational Force**: Each particle in the system exerts a gravitational force on every other particle. The force between two particles is computed using Newton's law of gravitation:

$$F = G\frac{m_1 m_2}{r^2}, \tag{1}$$

  where $G = 6.674 * 10^{-11}$ is the gravitational constant, m_1 and m_2 are the masses of the two particles, and r is the distance between them.

  The direction of the force is from the particle that applies the force and towards the particle that it is applied on. That is to say, for particle 1 and 2 at location l_1, l_2, the force applied by particle 2 on particle 1 is $\vec{F} = \frac{l_1 - l_2}{||l_1 - l_2||} G \frac{m_1 m_2}{||l_1 - l_2||^2}$

- **Softening Factor**: A small value is added to r^2 (or $||l_1 - l_2||^2$) to prevent numerical instability when two particles are very close. (In order to avoid a division by 0.)

- **Equations of Motion**: The motion of each particle is updated based on the net gravitational force acting on it. The equations of motion are:

$$\vec{a} = \frac{\vec{F}}{m}, \quad \vec{v}_{new} = \vec{v}_{old} + \vec{a}\Delta t, \quad \vec{x}_{new} = \vec{x}_{old} + \vec{v}_{new}\Delta t, \tag{2}$$

  where $\vec{a}$ is the acceleration, $\vec{v}$ is the velocity, $\vec{x}$ is the position, and Δt is the time step.

- **Applications**: The N-body problem is used to model planetary systems, galaxy formation, and interactions between celestial bodies.

2 Programming Requirements

You can start from your own nbody sequential code. Or start from the sequential code provided.

TODO. Implement the parallel N-body simulation in C++ using CUDA by completing the following tasks:

1. **Define the Simulation Structure**: Create a structure or class to represent the state of the simulation. This should include:
   - Particle properties: masses, positions, velocities, and forces (using vectors to store these values for all particles).
   - Initialization functions:
     - Random initialization of particle properties (masses, positions, velocities). Use appropriate random distributions.
     - Predefined configurations such as the solar system model with planets.
     - Load from file. Check recommended format.
2. **Memory Management and Transfers**:
   - For each array used by sequential:
     - Allocate memory on the device using `cudaMalloc`.
     - Copy initial data to the GPU using `cudaMemcpy`.
   - Copy results back to the host periodically for output.
3. **CUDA Kernel Implementation**: Write CUDA kernels to:
   - Calculate net gravitational force on each particle.
   - Update particle velocities and positions using the equations of motion.
   - Use appropriate grid and block dimensions.
   - Avoid unnecessary synchronizations.
4. **Output State**: Output the state of the simulation (e.g., positions, velocities, forces) to standard output in a structured format.
5. **Simulation Loop**: Write a main simulation loop that:
   - Iterates over a fixed number of time steps.
   - Calls CUDA kernels to update particle states.
   - At regular intervals, copies data back and prints the state.
6. **SLURM Execution**: Provide a SLURM script to run your simulation on Centaurus. Example:

```
#!/bin/bash
#SBATCH --job-name=yourprogramname
#SBATCH --partition=GPU
#SBATCH --time=00:10:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --gres=gpu:1

./yourprogramname 100000 0.01 50 101 128
```

Output Requirements. Your program should output the state of the simulation (positions, velocities, and forces of all particles) at regular intervals to standard output in the recommended format.

Suggestions.

- Use double precision for all calculations to ensure numerical stability.
- Start with a small number of particles (e.g., the solar system model) to debug and validate your implementation.
- **Command-Line Interface**: Ensure the program accepts:
  - Input specification: a number, `planet`, or a filename
  - Time step size (Δt)
  - Number of time steps
  - Print interval
  - CUDA block size
- **Output format:** Same format as `solar.tsv`:
  - One line per state
  - Each line starts with number of particles
  - Then each particle is represented with mass, position (x, y, z), velocity (vx, vy, vz), force (fx, fy, fz)
  - All values are tab-separated

3 Workflow support

If you have an nvidia GPU available, you probably can program directly from that machine using CUDA. But we suggest you work directly from Centaurus. When booking a node on Centaurus, you need to make sure you get a GPU node. You can request one by passing the following SLURM parameters `--partition=GPU --gres=gpu:1`.

You can get an interactive shell on a GPU node by first booking a node with `salloc` and then by getting a shell on that node using `srun`. Remember that with `salloc`, your reservation will keep running until it times out or until you terminate it explicitely. For instance, this works:

```
$ salloc --partition=GPU --time=01:00:00 --gres=gpu:1
$ srun --pty bash
```

If you have multiple reservations pending, you can attach to a particular one with `srun` by passing a jobid, for instance: `srun --jobid=12523 --pty bash`.

You will need to load the cuda module. We suggest adding to your `.bashrc` the following line: `module load cuda/12.4`

When compiling, you need to specify to nvcc which generation of GPUs to support. Compile with `-arch=sm_61` for compatibility with all GPUs available on Centaurus.

You have a GPU debugger available called `cuda-gdb` that works similarly to GDB. You have a tool akin to `valgrind` called `compute-sanitizer`.

You also have access to profilers. Old style profiler used `nvprof` and it is deprecated on more recent GPUs. More recent GPUs use `ncu`. Depending on the GPU you get in the SLURM allocation, you can use one or the other.

You can request a particular generation of GPU when you reserve the job with SLURM by passing different `--gres`. For instance: `--gres=gpu:TitanV:1`, `--gres=gpu:TitanRTX:1`, `--gres=gpu:V100:1`.

4 Benchmark

You should benchmark the time taken to compute nbody on both a CPU (even sequential) and on a GPU. Try different number of particles. Try 1000 particles, 10,000 particles and 100,000 particles.

You probably want to only run a few time step (5? 10?) to run the benchmark. The CPU should take significantly longer than the GPU.

5 Submission Instructions

TODO. Submit a single archive containing the following:

- Source code files implementing the simulation
- A Makefile to compile the project
- A README file with instructions
- Your SLURM script
- Log files for at least three test cases