EECS147 Project
Gravity Gurus

Merrick Slane
Neo Marin
Ted O'Young

University of California, Riverside
Department of Electrical and Computer Engineering

Overview

The idea of this project is to create an n-body planet simulator, and to create its efficiency between a CPU and GPU implementation. When the code compiles, it will prompt the user to simulate either the solar system or a variety of planets with a randomized mass, radius, position, and velocity. An Ascii display of the bodies will be displayed in the terminal, which demonstrates the gravitational interactions and motions between said bodies. Given that each interaction is calculated independently, it scales with a factor of $\frac{n(n-1)}{2}$.

GPU Advantage

The project aims to use a GPU rather than a CPU to demonstrate that it is capable of reducing the time needed to simulate characteristics of bodies in space.

Parallel Algorithm: The parallel simulation algorithm speeds up the simulation by parallelizing the physics calculations for each body. Every body in the simulation needs to obtain the gravitational force exerted on it by every other body. This normally requires two nested for loops. In CUDA, one of these for loops can be eliminated since each body has its own CUDA thread. This means that execution time will scale with O(n) instead of O (n^2) (discounting memory transfer overhead).

An atomicAdd operation to a variable called collisions is used to detect if any bodies have collided in the latest simulation tick. The collisions variable is written back from the GPU to the CPU after the GPU is done executing the current tick. If the collisions variable is greater than zero, a CPU-based function is called to delete colliding bodies and replace them with a single body. This is done on the CPU because it is a serial process that would encounter race conditions if attempted on the GPU.

Memory Allocation:

Central Processing Unit (CPU) - Kernel
Graphical Processing Unit (GPU) - Shared Memory

Pipelined Stage: In this algorithm, all individual body physics calculations and collision detections are performed in parallel on the GPU.

Thread Partitioning: Each body is given its own thread during the simulation. Since each body needs to know the position and mass of every other body, body data is loaded into shared memory during each tick in order to reduce memory access time during computation. This caps the maximum number of bodies at 256, since all bodies must occupy the same thread block.

Implementation Details

Solar System

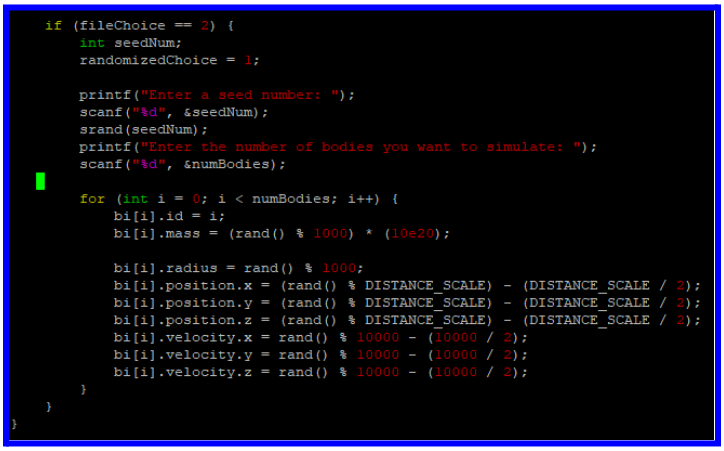
Simulating the solar system in CUDA requires that we have the data for each planet. **Figure 1** shows each variable needed to calculate the planet's position and trajectory in space. This data is read in via a Comma Separated Values (CSV) file. A user can also choose to upload any data for other galaxies as long as the file is in the same format.

Mass(kg)	Radius(m)	X_Position(m)	Y_Position(m)	Z_Position(m)	X_Velocity(m/s)	Y_Velocity(m/s)	Z_Velocity(m/s)
1.989e30	6.957e8	0e1	0e1	0e1	0e1	0e1	0e1
0.330e24	2439.5e3	4.094e10	4.094e10	0e1	33516.86e3	33516.86e3	0e1
4.87e24	6052e3	7.651e10	7.651e10	0e1	24748.73	24748.73	0e1
5.97e24	6378e3	1.057e11	1.057e11	0e1	21071.78	21071.78	0e1
0.642e24	3396e3	1.612e11	1.612e11	0e1	17041.27	17041.27	0e1
1898e24	714492e3	5.504e11	5.504e11	0e1	9263.09	9263.09	0e1
568e24	60268e3	1.012e12	1.012e12	0e1	6858.93	6858.93	0e1
86.8e24	25559e3	2.027e12	2.027e12	0e1	4808.32	4808.32	0e1
102e24	24764e3	3.192e12	3.192e12	0e1	3818.37	3818.37	0e1
0.130e24	1188e3	4.176e12	4.176e12	0e1	3323.40	3323.40	0e1

Figure 1 - Data of the Solar System

Random Bodies

The program can also be used to simulate random bodies in space. In some cases, a body can be seen orbiting another. **Figure 2** shows how the radius, position, and velocity of the planetary bodies are randomized.



```
    if (fileChoice == 2) {
        int seedNum;
        randomizedChoice = 1;

        printf("Enter a seed number: ");
        scanf("%d", &seedNum);
        srand(seedNum);
        printf("Enter the number of bodies you want to simulate: ");
        scanf("%d", &numBodies);

        for (int i = 0; i < numBodies; i++) {
            bi[i].id = i;
            bi[i].mass = (rand() % 1000) * (10e20);

            bi[i].radius = rand() % 1000;
            bi[i].position.x = (rand() % DISTANCE_SCALE) - (DISTANCE_SCALE / 2);
            bi[i].position.y = (rand() % DISTANCE_SCALE) - (DISTANCE_SCALE / 2);
            bi[i].position.z = (rand() % DISTANCE_SCALE) - (DISTANCE_SCALE / 2);
            bi[i].velocity.x = rand() % 10000 - (10000 / 2);
            bi[i].velocity.y = rand() % 10000 - (10000 / 2);
            bi[i].velocity.z = rand() % 10000 - (10000 / 2);
        }
    }
}
```

Figure 2 - Code Snippet of the Randomized Functionality

Inelastic Collisions

While simulating either the solar system or random bodies, the program is capable of simulating inelastic collisions between bodies in space. Bodies who collide with one another will be merged. **Figure 3** shows the deletion and creation of new bodies through inelastic collisions.

```
unsigned int CPU_collisions(struct body* bodies, int num_bodies) {
    struct body delete_bodies[MAX_BODIES];
    unsigned int delete_index = 0;

    struct body new_bodies[128];
    unsigned int new_bodies_index = 0;

    char deleted;

    for (int i = 0; i < num_bodies; i++) {
        deleted = 0;

        for (int k = 0; k < delete_index; k++) { //if body is marked for death
            if (delete_bodies[k].id == bodies[i].id) {
                deleted = 1;
                break;
            }
        }

        if (deleted == 0) {
            for (int j = 0; j < num_bodies; j++) {
                if ((distance(&bodies[i], &bodies[j]) < (bodies[i].radius + bodies[j].radius)) && (i != j)) { //if colliding
                    delete_bodies[delete_index] = bodies[i]; //mark for deletion
                    delete_index++;
                    delete_bodies[delete_index] = bodies[j];
                    delete_index++;

                    new_bodies[new_bodies_index] = create_new_body(&bodies[i], &bodies[j]);
                    new_bodies[new_bodies_index].id = bodies[i].id; //get new ID we know won't be used
                    new_bodies_index++;
                }
            }
        }
    }

    for (int i = 0; i < delete_index; i++) { //delete bodies
        num_bodies = delete_body_id(delete_bodies[i].id, bodies, num_bodies);
    }

    for (int i = 0; i < new_bodies_index; i++) { //add bodies
        bodies[num_bodies] = new_bodies[i];
```

Figure 3 - Code Snippet of the Inelastic Collision Functionality

Documentation

How to Run the Code

1. Run ./simulator to begin the simulation

```
EECS147-Project $ ./simulator
```

2. This is the first user prompt. Enter 1 to simulate using the CPU or 2 using the GPU

```
Press 1 for CPU calculations or 2 for GPU calculations:
```

3. This will lead to the second user prompt. Enter 1 to simulate using body data from a .csv file or 2 for bodies with randomized properties, i.e. mass, radius, position, and velocity.

```
Press 1 for solar system simulation or 2 for randomly generated simulation:
```

4. .csv file) If 1 is pressed, this is the third user prompt, asking which file to simulate (bodydata.csv includes all the values)

```
Type in the file name you would like to simulate:
```

   Randomly generated) If 2 is pressed, it will prompt the user to enter a seed number (recommend entering seed value of six digits or more), followed by another prompt regarding how many bodies to simulate.

```
Enter a seed number:
```

5. .csv file) After the filename is entered, the user is asked for how long they would like to simulate the bodies.

```
Enter Max Ticks for Runtime:
OR Enter 0 for Unlimited Ticks.
```

   Randomly generated) The next prompt asks the user how many bodies they would like to simulate. The limit is 256. Afterwards, it will ask the same prompt as the solar system.

```
Enter the number of bodies you want to simulate [MAX:256] :
```

6. Decide on the number of ticks per display frame.

```
Enter number of ticks per display frame [RECOMMENDED: 10] : 10
```

7. Enter the number of max ticks, i.e. how long you want the simulation to run.

```
Enter max ticks for runtime OR Enter 0 for unlimited ticks:
```

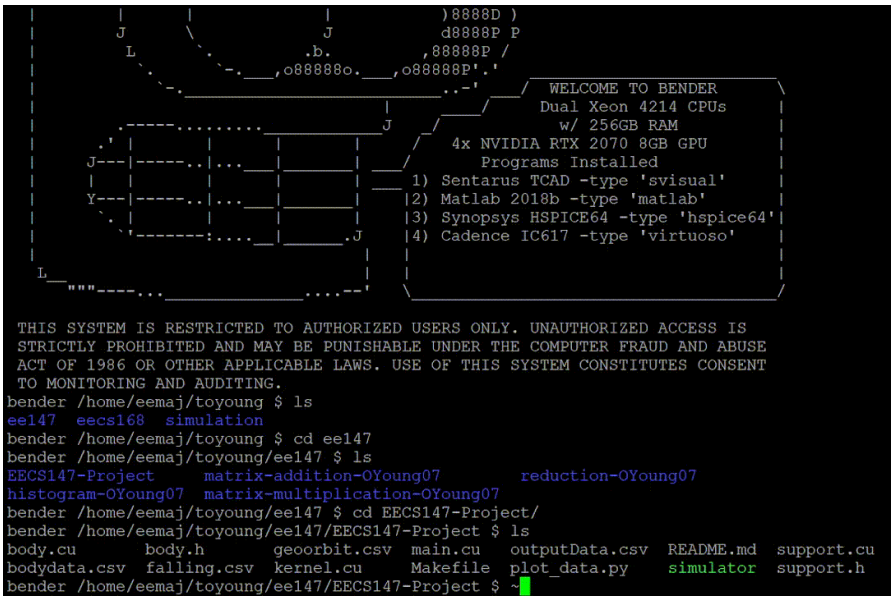
8. Decide whether you would like to simulate the collisions

```
Press 1 for inelastic collisions and 0 for no collisions:
```

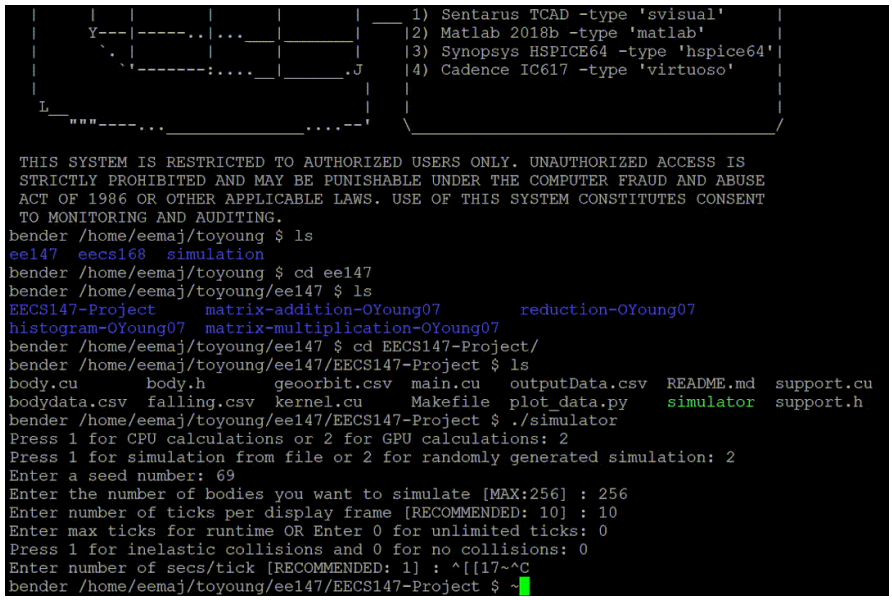
9. Enter the seconds per tick, i.e. how fast you want the simulation to run.

```
Enter number of secs/tick [RECOMMENDED: 1] :
```

10. Solar system) This is a gif of the simulation of the solar system:



Randomly generated) This is a gif of 256 bodies with randomized properties:



11. This is a screenshot of the result of the randomized simulation.

```
Bodies:256, Scale=1.188993e+06 meters, Tick=1900
----======+++ SIMULATION RESULTS +++======----
Elapsed Time: 2.243573 s | Seed: 12345 | Ticks/Second: 1.000000 | Max Ticks: 2000
```

Evaluation

Results

The simulation was run for 10000 ticks with collisions turned off so that the number of bodies would remain constant for the entire simulation. The simulation was run using bodies with float attributes and bodies with double attributes. Simulation time results for the CPU and GPU are tabulated below. Smaller numbers correlate to better performance for this case.

Number of Bodies		**2**	**50**	**100**	**150**	**200**	**256**
Execution Time (s)	**CPU**	0.043	0.69	2.57	5.70	10.14	16.38
	GPU	2.37	4.89	9.18	14.32	22.47	31.2

Table 1 - Simulation results for various body counts (with doubles)

Number of Bodies		**2**	**50**	**100**	**150**	**200**	**256**
Execution Time (s)	**CPU**	0.048	0.89	2.59	5.64	9.96	16.86
	GPU	2.40	3.65	5.33	6.97	8.79	11.72

Table 2 - Simulation results for various body counts (with floats)

It can be seen that the GPU is much faster for larger body counts when floats are used instead of doubles. This is because the number of bytes that must be transferred between the device and host are halved when using floats, resulting in much better memory bandwidth. Additionally, Nvidia GPUs have a 32-bit ALU, meaning that they can natively operate on floats, but extra overhead is needed to operate on doubles.

Plots of system energy over time are given below. In an ideal system, net energy would remain constant over time. Here we see that energy is constant for discrete regions of the simulation, but net energy occasionally changes abruptly due to floating point imprecision and simulation roughness. Typically these jumps are correlated with a large merge of several bodies, as can been seen in figures 4 and 5. Overall, this performance is satisfactory for the purpose of basic n-body simulations.

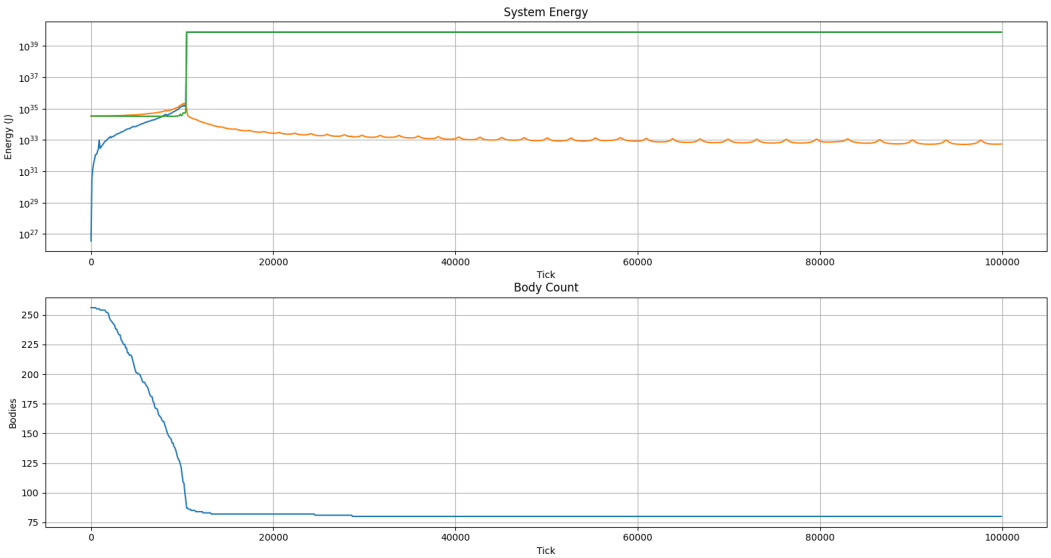


Figure 4 - Energy over time in a system with inelastic collisions

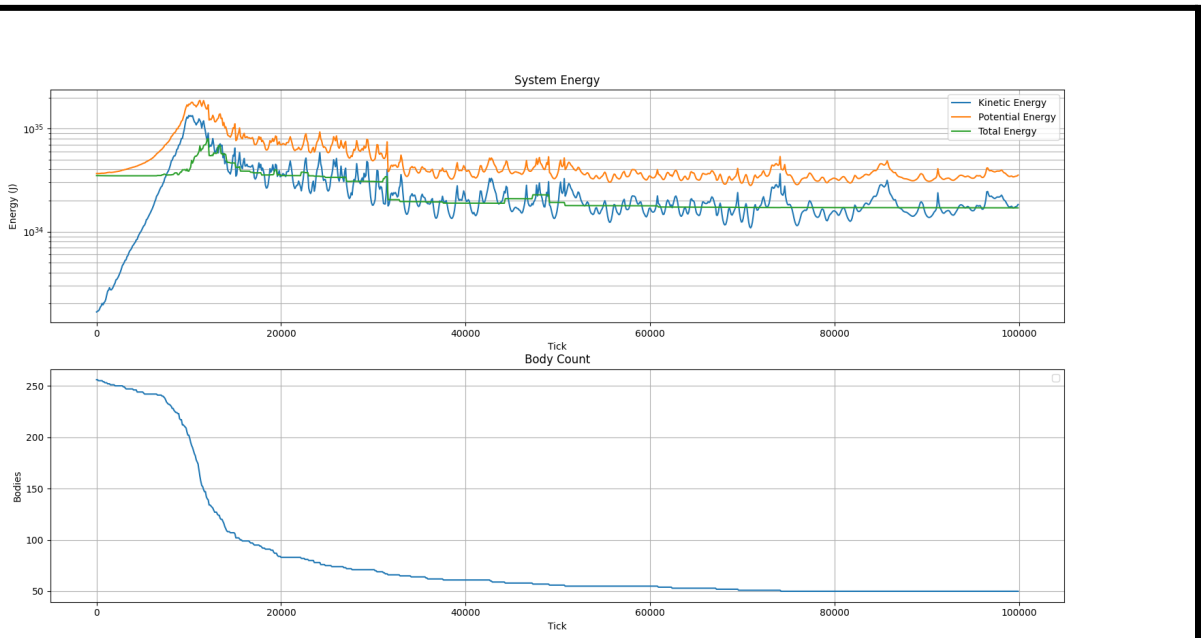


Figure 5 - Energy over time in a system with inelastic collisions and nonzero starting velocities

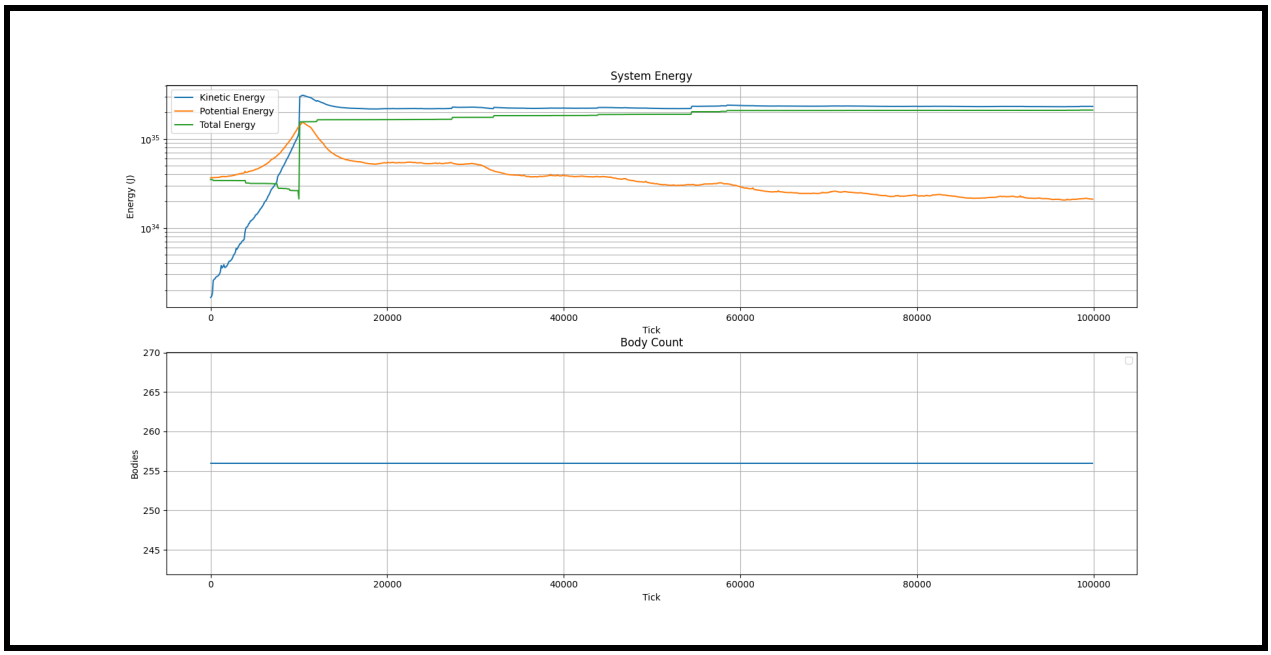


Figure 6 - Energy over time in a system with no collisions

Issues Encountered

There were several issues encountered throughout the process of the project. Detailed below are some of the most notable issues.

- One non-technical issue was dealing with git. Given that several team members would work on the project outside of assigned meeting times, remembering to git pull was essential to avoid any unnecessary merge conflicts.
- In terms of technical problems, there was an overflow issue for floats when performing calculations among the planets in our solar system. A float value cannot exceed 3.402823466 E + 38. When multiplying the masses between two large bodies, it would surpass this threshold and cause display errors.
- The data of the solar system is contained in a separate file named "bodydata.csv". Reading the data of the first line of this file was challenging due to unknown errors. Magically, calling fgets(line, MAX_LENGTH_LINE, file) twice fixed this problem.
- The last problem was generating the randomized values for the planetary bodies. Several random negative numbers would be applied to the mass, which would not be possible. The fix to this was to multiply the final result by 10e20.

Responsibilities

Task	Breakdown
Main Simulator Logic & Functions	**Merrick Slane** - 100% 🐐 **Ted O'Young** - 0% **Neo Marin** - 0%
User Prompts	**Merrick Slane** - 0% **Ted O'Young** - 100% **Neo Marin** - 0%
File Reader	**Merrick Slane** - 0% **Ted O'Young** - 100% **Neo Marin** - 0%
Timer Functionality	**Merrick Slane** - 0% **Ted O'Young** - 0% **Neo Marin** - 100%
Randomized Bodies	**Merrick Slane** - 20% **Ted O'Young** - 80% **Neo Marin** - 0%
Inelastic Collision Functionality	**Merrick Slane** - 95% **Ted O'Young** - 5% **Neo Marin** - 0%
Solar System Integration	**Merrick Slane** - 20% **Ted O'Young** - 80% **Neo Marin** - 0%
Ascii Display of Bodies in Terminal	**Merrick Slane** - 100% **Ted O'Young** - 0% **Neo Marin** - 0%
Simulation Results Data & CSV Output	**Merrick Slane** -10% **Ted O'Young** - 0% **Neo Marin** - 90%
Project Report	**Merrick Slane** - 30% **Ted O'Young** - 40% **Neo Marin** - 30%

Demonstration

You can find a video of our demonstration here: https://www.youtube.com/watch?v=xfyBU3uOHeQ