N-Body Simulation with CPU and CUDA

Alan Devkota, Student ID: 2215320
Department of Electrical and Computer Engineering
University of Houston
Houston, TX, USA
adevkota2@uh.edu

***Abstract*—The N-body problem is used in this project to mimic how particles move across space. The project comprises a Python-based serial implementation and a C++/Cuda-based parallel implementation. A program is generated that simulates gravitational force between n bodies in space, exploiting the massively parallel architecture provided by GPGPUs. This program generates N particle locations over a specified number of timesteps. Moreover, it produces gif plots to visualize the bodies in the simulation similar to stars in the galaxy. The results generated compare the performance of CPU-based implementation and GPU-based implementation.**

***Index Terms*—GPU, CUDA, N body simulation.**

I. Introduction

An N-body simulation numerically approximates the evolution of a system of bodies in which each body interacts with every other body continuously. An astrophysical simulation in which each entity represents a galaxy or a single star, and the bodies attract each other via gravitational pull is an example of n-body simulation. [1], [2], [3]

The all-pairs method for N-body simulation is a brute-force method that examines all pair-wise interactions between the N bodies. It is a straightforward approach, but because of its $O(N^2)$ computational cost, it is rarely utilized for large systems modeling. Instead, the all-pairs method is commonly employed as a kernel to calculate forces in close-range interactions. The all-pairs technique is paired with a faster method based on a far-field approximation of longer-range forces, which is only applicable between well-separated sections of the system. The all-pairs component of the algorithms takes a long time to compute and is thus an appealing target for GPU acceleration.[1]

Thus, the N-body problem is utilized to simulate how particles flow across space in this project for Python-based serial and C++/Cuda-based parallel implementation. First, the position of N particles is generated over a given number of timesteps and stored in an output text file for both CPU and GPU implementations. Next, the movement of the particle positions generated from the python and Cuda programs is plotted using a simulator program written in python. Finally, the runtime of both CPU and GPU implementations are plotted by increasing the number of bodies in powers of 2. The results generated compare the performance of CPU-based and GPU-based implementations.

II. All-Pairs N-Bbody Simulation

We use gravitational potential to illustrate the basic form of computation in an all-pairs N-body simulation.

The total force $\mathbf{F}_i$ on body i, as a result of its interactions with the other $N-1$ bodies, is obtained by summing all interactions:

$$\mathbf{F}_i = \sum_{\substack{1\le j\le N\\ j\neq i}} \mathbf{f}_{ij} = Gm_i \cdot \sum_{\substack{1\le j\le N\\ j\neq i}} \frac{m_j \mathbf{r}_{ij}}{\|\mathbf{r}_{ij}\|^3}$$

The force between bodies increases without bounds as they get closer to one another, which is an undesirable situation for numerical integration. In astrophysical simulations, collisions between bodies are generally precluded; this is reasonable if the bodies represent galaxies that may pass right through each other. Therefore, a *softening factor* $\epsilon^2 > 0$ is added, and the denominator is rewritten as follows:

$$\mathbf{F}_i \approx Gm_i \cdot \sum_{1\le j\le N} \frac{m_j \mathbf{r}_{ij}}{\left(\|\mathbf{r}_{ij}\|^2 + \epsilon^2\right)^{3/2}}$$

To integrate over time, we need the acceleration $\mathbf{a}_i = \mathbf{F}_i/m_i$ to update the position and velocity of the body i.

The preceding equations show that the cumulative influence of n-1 particles on a single particle may be estimated individually for each time step.

III. Methodology

The force on each body in the all-pairs method is the total acceleration induced by every other particle multiplied by the mass of that body. A single thread calculates forces on a single body individually. For every particle, the total of all accelerations is evaluated to calculate the velocity and new position. Each particle's position and velocity are updated at every time step. To parallelize the code 1024 threads were selected and for each thread aceleration, position, inverse division and softening were computed via kernels. (Run nbody.py OR nbody.cu to generate data from the N-body simulation. Run simulation.py to visualize the simulation.)

Steps to run the nbody simulation: First, the position of N particles is generated over a given number of timesteps and stored in an output text file for both CPU and GPU implementations. The nbody.py is the python CPU-based program that creates an output_py.txt text file. N and timesteps are command line arguments illustrated in the example below:

- python nbody.py 1000 150

The nbody.cu is the GPU-based program compiled to create an executable file called nbody. Executing nbody creates an output_cu.txt text file. N and timesteps are command line arguments illustrated in the example below:

- ./nbody 1000 150

Next, the movement of the particle positions generated from the python and Cuda programs is plotted using a simulator program written in python. This program's input file is supplied as a command line option for both CPU and GPU implementations illustrated in the examples below:

- python simulation.py output_py.txt anim_py.gif
- python simulation.py output_cu.txt anim_cu.gif

Note: To create gif files for the simulation we need to install imagemagick library using ! **apt install imagemagick.**

Finally, the runtime of both CPU and GPU implementations are plotted by increasing the number of bodies in powers of 2 via an evaluation program written in python. The number of iterations (times the number of bodies increases) is supplied as a command line input illustrated in the example below:

- python evaluate.py 7

The above example is running the evaluation with iterations=7. That is $(N = [2, 4, 8, 16, 32, 64, 128])$.

The results generated compare the performance of CPU-based and GPU-based implementations.

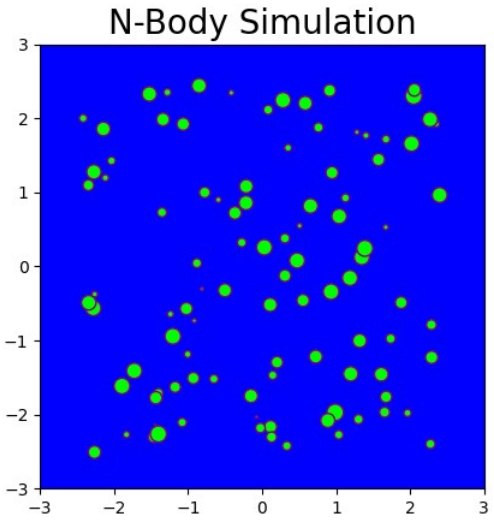

Fig. 1. View of N-body in CPU for N = 100 and timesteps = 150. **[please look gif images in the submitted folder]**

IV. Experimental Results

The experiment was conducted in Google Colab with Tesla T4 GPU (GPU name: Persistence-M) The gif images anim_cu.gif and anim_py.gif included in submitted folder shows an example of nbody simulations in CPU and GPU, respectively. Figure 1 is the screenshot of gif result after running nbody.py program for N=100 and timesteps=150.

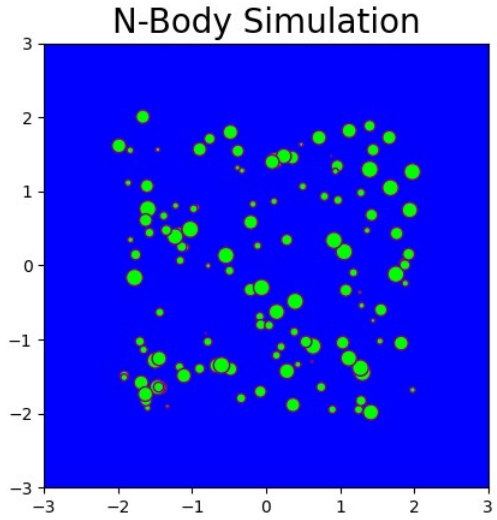

Fig. 2. View of N-body in cuda for N = 100 and timesteps = 150. **[please look gif images in the submitted folder]**

Figure 2 is the screenshot of gif result after running nbody.cu program for N=100 and timesteps=150.

Note: Only 100 bodies were used for the results because CPU took very much time for bodies higher than 128.

The runtime were plotted for both nbody.py and nbody.cu files by varying the number of bodies in power of 2 and selecting the timestep as 150.

Figure 3 shows the runtime for N=10 (1024 bodies). We can observe that by paralleled GPU implementation required approximately 30 seconds and serialized CPU implementation required approximately 3000 seconds (50 minutes approx).

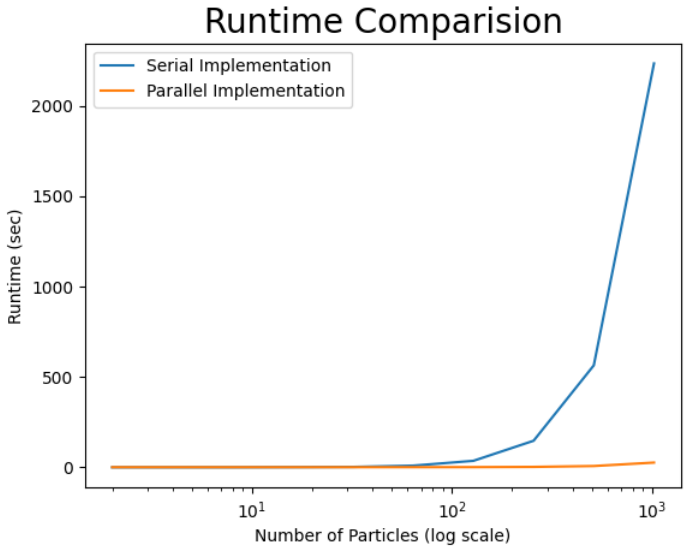

Fig. 3. Miss Rate for L2 cache by changing the cache size with constant block size of 64B and 4-way associativity.

V. Conclusion

CUDA provides an outstanding hardware layer for running massively parallel programs, and CUDA-enabled GPUs shine when pushed to their limits (upwards of 10000 threads per GPU). It is also dependent on utilizing the computing specifications. The location of the n bodies can be simulated over a specified number of timesteps using python CPU-based and CUDA GPU-based implementation. The performance of GPU-based implementation in GPU is approximately 99.6 times better than CPU-based implementation for my program.

Acknowledgment

The author would like to thank Dr. David Mayerich.

References

[1] L. Nylons, M. Harris, and J. Prins, "Fast n-body simulation with cuda," *GPU gems*, vol. 3, pp. 62–66, 2007.

[2] R. Zioma, "Gpu gems 3," 2007.

[3] Wikipedia contributors, "Cuda — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=CUDA&oldid=1127269916, 2022, [Online; accessed 16-December-2022].

VI. Appendix

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 460.32.03    Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   42C    P0    25W /  70W |      0MiB / 15109MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

Fig. 4. GPU info

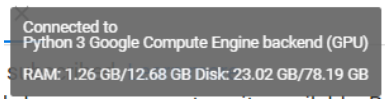

Fig. 5. more info

Use this link below to run everything from Google Colab: Nbody Simulation in Google Colab by Alan.