Using CUDA to Increase the Accuracy and Performance of Particle-Particle N-Body Simulations

Ryland Goldman

28 April 2023

Abstract

N-body simulations are computer models of systems of bodies typically used for physics and astronomy applications, such as predicting orbits of planets. These simulations typically require large amounts of processing power and time for physics computations. To solve this issue, developers use rounding and make compromises on accuracy in order to optimize the software. This project aims to use hardware acceleration rather than mathematical approximations to improve the performance of the simulation, written in Python.

The project compares a NumPy-based approach running on a 16-thread Intel 12600K CPU (compiled with Numba JIT) with CuPy interfacing with a NVIDIA 3090 GPU via the CUDA framework. The CPU group was the control, and CUDA was the experimental group. Two additional test groups used PyOpenCL to directly compare each device. One hundred trials were run on each of the four groups, and repeated using powers of two between 2^{13} and 2^{18} bodies.

Using 2^{16} bodies, the speed up multiple for CuPy was 3.66x, OpenCL (GPU) was 1.05x, and OpenCL (CPU) was 0.56x. This suggests that CUDA is significantly faster than only using the CPU for computations, and the GPU OpenCL implementation was about twice as fast as the CPU OpenCL implementation.

1 Introduction

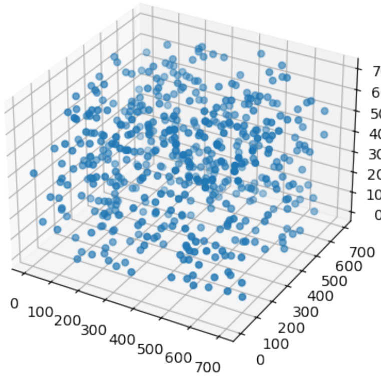

Figure 1: Example Output

N-body simulations are physics simulations used to study the motion of a number of particles or bodies, such as stars, galaxies, or planets (as shown in Figure 1). They calculate the trajectories of the individual bodies by taking into account the gravitational forces between them, and are commonly used for understanding the nature of the universe — for example, in Newtonian orbital predictions and studying the formation of galaxies (Antoñana et al., 2022). The time complexity of computing such forces increases on the order of $O(n^2)$, meaning that doubling the number of bodies takes four times as long to run (Trenti and Hut, 2008). Historically, mathematical approximations, such as the Barnes-Hut tree algorithm (Anderson, 1999), have been used to reduce the complexity to $O(n \log n)$. However, these approximations can introduce errors and can be less accurate than direct, ”particle-particle” simulations.

To prevent the errors caused by rounding, this project focuses on hardware acceleration on a Graphics Processing Unit (GPU) while keeping the most accurate, though relatively slow, ”particle-particle” algorithm. GPUs are designed for parallel programming, and are commonly used for general-purpose GPU computing (GPGPU) in addition to their primary target of rendering video. Rather than stepping through calculations one at a time, a GPU can run thousands of operations concurrently. This is done with a compute kernel, a single function which sends instructions to every core in the GPU at once (Rasch et al., 2020).

The kernels are written with two of the most popular frameworks, CUDA and OpenCL. CUDA is an interface first developed by Nvidia in 2007 specifically to run on its GPUs (Rasch et al., 2020). OpenCL is a similar framework created by Apple to run cross-platform (i.e., on both Central Processing Units [CPUs] and GPUs of any brand). The libraries CuPy and PyOpenCL are used to interface the GPU with the main Python program, and NumPy runs on the CPU.

The dependent variable is the runtime of the simulation. A faster runtime with the GPU indicates a more efficient program, and therefore a successful project.

2 Methods and Materials

The computer used in this experiment has an Intel® Core™ i5-12600K CPU and an Nvidia GeForce® RTX™ 3090 GPU. The computer has 32 gigabytes of random-access memory (RAM). The Python version is 3.10.7, with CuPy 11.3.0, NumPy 1.23.3, and PyOpenCL 2022.3.1 installed with dependencies.

Four primary test groups were used: NumPy, CuPy, PyOpenCL GPU, and PyOpenCL CPU. The NumPy framework, compiled with the Numba library, ran only on the CPU. CuPy, only compatible with Nvidia GPUs, was GPU-only. PyOpenCL represented the two remaining groups, one on the CPU and one on the GPU. Each group ran for one hundred trials with one iteration per trial. The number of bodies varied from 2^{13} (8,192) to 2^{18} (262,144) particles. After running the simulations, the runtime in seconds was recorded to a web server and later analyzed with R, a statistical programming language.

The main Python script, which can be found on GitHub[†], contains separate sub-programs for each test group. Each iteration runs a nested loop (hence the $O(n^2)$ time complexity) which computes the distance between each particle, the resulting gravitational force, and uses trigonometry to update velocity through integrating with small time steps (Trenti and Hut, 2008).

[†] http://github.com/ryland-goldman/n-body-simulation

The parameters of the simulation included the number of iterations, the framework to use, and the number of bodies. At the beginning of the run, initial conditions were randomly generated and the code was compiled by the respective library. After completion, the data is compiled into a video animation or interactive plot output. Only the sub-programs were timed.

3 Results

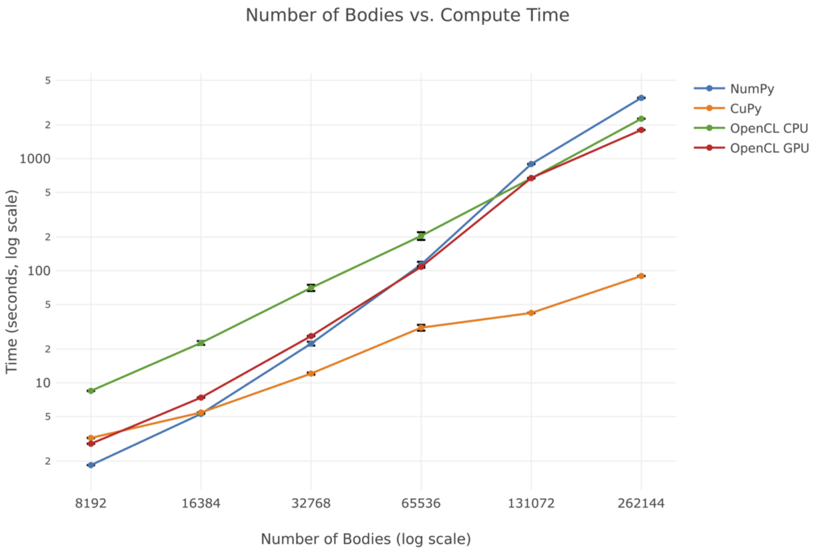

Figure 2: Bodies vs. Runtime (log-log) Scatter Plot

Figure 2 compares the number of bodies used in the simulation, ranging in powers of two from 2^{13} to 2^{18}, to the runtime of the simulation. The power regression, or trendline, of the NumPy (CPU-only) group was $cx^{2.26}$, while the CuPy (GPU-only) group was $cx^{0.978}$.

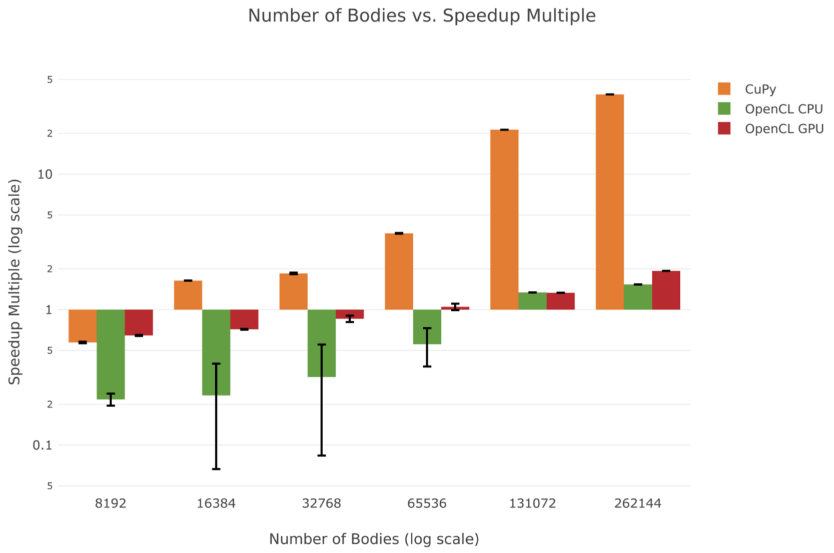

Figure 3: Bodies vs. Speedup (log-log) Bar Graph

Figure 3 examines the speedup of using different frameworks as a multiple of the performance increase compared to NumPy.

Both figures were generated with the Plotly library in R. The error bars represent the 95% confidence interval using 2 × $SE_{\bar{x}}$. There was no overlap between the NumPy and CuPy test groups, so the data is likely statistically significant.

4 Analysis

Overall, when the number of bodies is greater than or equal to 2^{14}, the CuPy group was the fastest. As the number of bodies grew, the speedup over NumPy increased to 38.8x faster with the maximum 2^{15} bodies.

Unexpectedly, the trendline of the CuPy group was $cx^{0.978}$ — since the time complexity is $O(n^2)$, the trendline should be around cx^2. This could mean that there are spare GPU threads available, decreasing the runtime for larger groups which have a higher GPU utilization. Assuming no random or systematic errors, the trendline should approach cx^2 as the number of bodies increases to infinity.

When the number of particles was small (2^{13} bodies), the CuPy speed was 0.572x slower than the NumPy speed. Each iteration of the simulation, data is sent from the CPU's random-access memory (RAM) to the GPU's video RAM in order for the necessary computations to take place on the GPU. There is limited bandwidth between the two components, so a latency is expected. When the number of calculations is small, the latency can overtake the benefits of parallelization. This is akin to using a calculator to find the answer to a simple arithmetic problem—the calculator can do the work instantly, but the delay of typing in all of the numbers makes the total process longer than mental math. The breakeven point, where NumPy and CuPy times are roughly equivalent, seems to be just below 16,000 bodies.

The OpenCL groups were a control to more directly compare the CPU and GPU using the same framework. In total, the GPU OpenCL group was consistently faster than the CPU OpenCL group, validating the NumPy/CuPy results. Both OpenCL groups, however, were consistently lower (in all but two test cases) than the native frameworks (NumPy and CuPy). For example, with 2^{18} bodies, the CuPy speedup was 38.8x compared to the OpenCL GPU speedup of 1.9x. This suggests that using OpenCL provides the benefits of cross-platform portability, enabling it to run on a wide variety of hardware, but comes at the cost of performance. CuPy and NumPy are specifically designed to run optimally on their respective devices. OpenCL does not have that freedom, causing it to run slower across the board.

5 Conclusion

For any researcher looking to improve the performance of a computational physics simulation, not limited to N-body simulations, this study finds that the CUDA framework is generally the fastest method for a large number of bodies. For smaller simulations, such as orbital predictions limited to less than a few thousand bodies, a CPU-based Python model is likely to give the best performance. While OpenCL benefits from portability, it is better to use the native programming on the respective device in order to achieve optimal performance.

Future studies should address the magnitude at which a Barnes-Hut algorithm decreases the accuracy compared to a particle-particle simulation. Additionally, various compiled languages such as C or C++ should be tested for further potential performance improvements, as well as the trade-offs of techniques like adaptive time stepping (where the integration time steps are varied through machine learning). This project's budget was limited, so only a single CPU and GPU pair was tested. A datacenter or supercomputer is likely to have many devices running concurrently and could be another space for exploration by testing whether the current trends continue as the number of bodies increases.

References

Anderson, R. J. (1999). Tree data structures for N-body simulation. *SIAM Journal on Computing*, *28*(6), 1923–1940. https://doi.org/10.1137/S0097539797326307

Antoñana, M., Chartier, P., & Murua, A. (2022). Majorant series for the N-body problem. *International Journal of Computer Mathematics*, *99*(1), 158–183. https://doi.org/10.1080/00207160.2021.1962848

Lai, J., Yu, H., Tian, Z., & Li, H. (2020). Hybrid MPI and CUDA parallelization for CFD applications on multi-GPU HPC clusters. *Scientific Programming*, *2020*, 1–15. https://doi.org/10.1155/2020/8862123

Lee, C., Ro, W. W., & Gaudiot, J.-L. (2014). Boosting CUDA applications with CPU–GPU hybrid computing. *International Journal of Parallel Programming*, *42*(2), 384–404. https://doi.org/10.1007/s10766-013-0252-y

Mathias, S., Coulier, A., & Hellander, A. (2022). CBMOS: A GPU-enabled Python framework for the numerical study of center-based models. *BMC Bioinformatics*, *23*(1), 55. https://doi.org/10.1186/s12859-022-04575-4

Rasch, A., Bigge, J., Wrodarczyk, M., Schulze, R., & Gorlatch, S. (2020). Docal: High-level distributed programming with OpenCL and CUDA. *The Journal of Supercomputing*, *76*(7), 5117–5138. https://doi.org/10.1007/s11227-019-02829-2

Trenti, M., & Hut, P. (2008). N-body simulations (Gravitational). *Scholarpedia*, *3*(5), 3930. https://doi.org/10.4249/scholarpedia.3930