

### Università degli Studi di Padova

# N-Body Simulation: Design, Optimizations and Benchmarking

M. Sc. Physics of Data, University of Padova Alessio

July 23, 2025

#### Outline



- 1. Introduction and Motivation
- 2. Project Evolution
- 3. Memory Management and Optimizations
- 4. Force and Integration Methods
- 5. Limitations and Lessons Learned
- 6. Conclusions

#### Introduction and Motivation



- Started as a simple single-file C++ code for Voyager II trajectory
- Evolved into a modular, configurable N-body simulator
- Main goals:
  - Flexible and extensible framework
  - Fair CPU vs GPU performance comparison
  - Advanced optimizations and interactive visualization

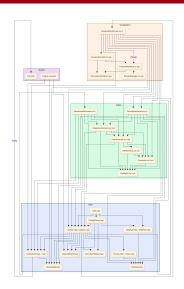
### Project Evolution



- Initial single-file C++ implementation
- Output separation and modularization
- Integration of OpenMP, CUDA, (AVX2)
- Refactoring, CMake build system, JSON configuration
- Interactive visualization with OpenGL

#### General Architecture



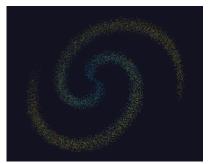


- Core: simulation logic, data structures, and CPU part
- CUDA: GPU kernels, utilities and host functions
- Visualization: OpenGL and OpenGL/CUDA interop
- Output: CSV output, dynamical buffer to use for other integrations
- Configuration: single JSON file

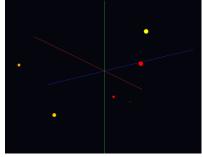
### Configuration and Initialization



- Single JSON configuration file
- Flexible initialization: random, file-based, or custom implementations (galaxy, stellar systems, etc.)



Example of a galaxy configuration



Example of random initialization

# Memory Management and Optimizations



- Structure of Arrays (SoA) for efficient access and fully coalesced memory with double4
- Pinned memory and CUDA streams
- AVX2 vectorization and OpenMP threading
- Low-ish memory footprint: 1.1 GiB for 1B particles
- $\bullet$  In CSV output mode, System is copied only at the start  $\to$  multiple cuda streams for memory and compute management.
- In visualization mode, System is copied every frame to a dynamic buffer, which is then used for rendering → worst case scenario ⇒ OpenGL-cuda interop and separate rendering stream → less bottleneck.

### Force and Integration Methods



- Pairwise, Adaptive Mutual Softening, Barnes-Hut (no GPU implementation)
- Configurable physical constants and softening
- Multiple integration schemes: Euler (not symplectic), Velocity Verlet (symplectic)

#### Pairwise Newtonian:

# $\vec{a}_i = G \sum_{j \neq i} m_j \frac{\vec{r}_j - \vec{r}_i}{|\vec{r}_j - \vec{r}_i|^3}$

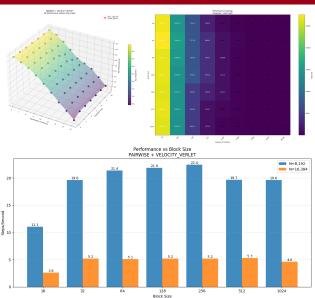
#### **Adaptive Mutual Softening:**

$$\begin{split} \vec{a}_i &= G \sum_{j \neq i} m_j \frac{\vec{r}_j - \vec{r}_i}{\left( |\vec{r}_j - \vec{r}_i|^2 + \epsilon_{ij}^2 \right)^{3/2}} \\ \epsilon_{ij} &= \max \left( \eta \, |\vec{r}_j - \vec{r}_i| \left( \frac{m_i + m_j}{3 \langle m \rangle} \right)^{1/3}, \, \epsilon_{\min} \right) \end{split}$$

where G is the gravitational constant,  $\epsilon_{ij}$  is the adaptive softening,  $\eta$  and  $\epsilon_{\min}$  are tunable parameters, and  $\langle m \rangle$  is the average mass.

### Block-Size Comparison Benchmark

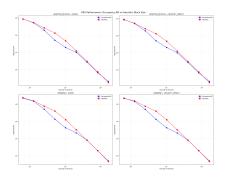


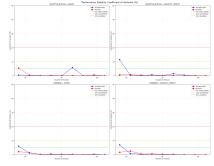


### Block Size Auto-Tuning



- **block size selection** is crucial for GPU efficiency in these problems.
- Two strategies compared:
  - Occupancy API (NVIDIA)
  - Heuristic (custom, optimized for this workload)





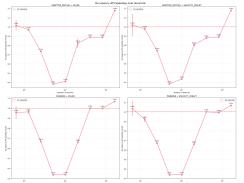
### Block Size Auto-Tuning



Grid size:

$$gridSize = \left\lceil \frac{n}{blockSize} \right\rceil$$

The heuristic is robust, saturates SMs, and respects shared memory limits. The grid size formula is exact for 1D N-body kernels.



#### Heuristic Block Size Selection (pseudocode)

blockSize = 1024 if  $n \le 32$ : return 32 while sharedMem(blockSize)

- > maxShared && blockSize
- > warp:

blockSize //= 2

blockSize = min(blockSize,
maxThreads)

while numBlocks(blockSize)

- < 2\*numSMs && blockSize
- > warp:

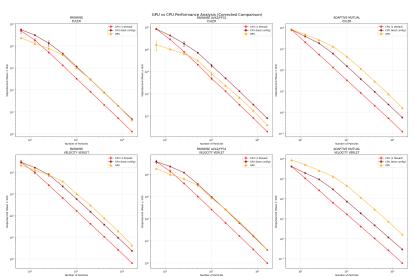
blockSize //= 2

blockSize = max(blockSize,
warp)

return blockSize

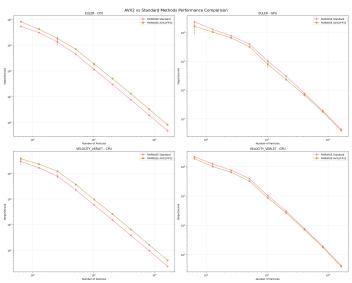
# Basic CPU-GPU Comparison Benchmark





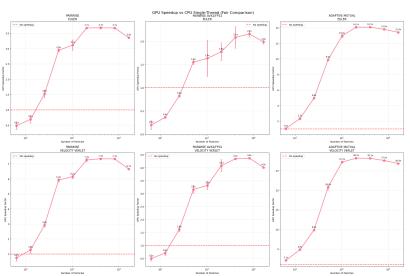
# Basic CPU-GPU Comparison Benchmark





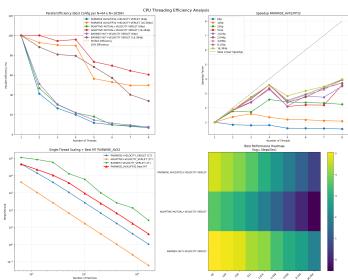
# Basic CPU-GPU Comparison Benchmark





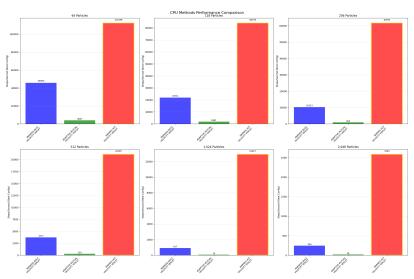
## CPU-Methods Comparison Benchmark





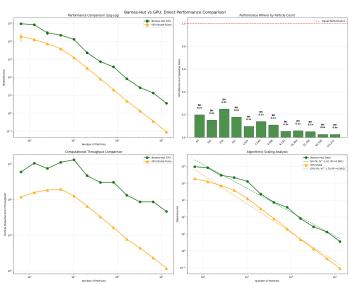
### CPU-Methods Comparison Benchmark





# BH vs GPU Comparison Benchmark





#### Limitations and Lessons Learned



- Bottlenecks in compute for very large N (different methods, ResNets?, ...)
- Tradeoffs between accuracy and speed (double in GPU, same memory pattern in CPU).
- Importance of a clear vision at the start of the project
- Need for a robust testing framework
- Make only one main entry point per external library (or as few as possible)
- Future improvements: a running program you can compile without looking at any library requirements

#### Conclusions



- We skipped many small details (tiling, async memory...) in this presentation, so if you want to know more, please check the GitHub repository
- High-performance, modular N-body simulation achieved
- GPU acceleration enables large-scale experiments of:
  - $\sim 10^{3--4}$  particles in real time (> 30 steps/s or FPS)
  - $\bullet~\sim 10^{5--6}$  particles not in real time
- Framework ready for extension and research

# Thank you for your attention! Questions?

Demo?