

Hardware-Accelerated N-Body Gravitational Simulation with Real-Time Visualization

1st Ben Volokh

Department of Physics
Massachusetts Institute of Technology
Cambridge, MA, USA
bvolokh1@mit.edu

2nd Jackson Kay

Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
zjkay@mit.edu

Abstract—The classical N-body problem is computationally intensive because every particle interacts with every other particle, creating a workload that scales quadratically with system size. This $O(N^2)$ growth presents a severe challenge for conventional processors, which must evaluate all pairwise forces sequentially or in limited parallel batches. As N increases, the cost of maintaining real-time performance, especially when accurate numerical integration and smooth visualization are required, quickly becomes prohibitive. Field-programmable gate arrays (FPGAs) offer a compelling alternative: their fine-grained parallelism and configurable pipelines map directly onto the repetitive interaction, accumulation, and update steps that dominate N-body complexity.

In this project, a Spartan 7 Series FPGA is used to implement a multi-stage streaming architecture in which force evaluation, state updates, and visualization are processed in dedicated hardware pipelines. Because the gravitational (or relativistic) interactions between distinct particle pairs are independent, these pipelines can exploit massive parallelism at the level of individual interactions. The result is a sustained-throughput N-body engine capable of continuous simulation and real-time display, demonstrating how reconfigurable hardware can overcome the scaling challenges inherent in N-body computation while supporting responsive, high-resolution graphical output.

Index Terms—Digital systems, Field programmable gate arrays, Physics simulation, N-Body simulation, Relativistic dynamics

I. INTRODUCTION

The N-body problem describes the motion of a system of particles whose dynamics are governed by pairwise interactions. For a system of N particles with masses m_i at positions $\mathbf{r}_i$, the gravitational force exerted on particle i by particle j is

$$\mathbf{F}_{ij} = G \frac{m_i m_j}{\|\mathbf{r}_i - \mathbf{r}_j\|^3} (\mathbf{r}_j - \mathbf{r}_i), \quad (1)$$

where G is the gravitational constant, $6.67 \times 10^{-11} \text{ N} \cdot \text{m}^2 \text{ kg}^{-2}$. The net force on particle i is the sum over all other particles,

$$\mathbf{F}_i = \sum_{\substack{j=1 \\ j \neq i}}^N \mathbf{F}_{ij}. \quad (2)$$

The key difficulty is that each particle interacts with the other $N - 1$ particles, producing a combinatorial explosion

of $N(N - 1)$ interactions per timestep [2]. This leads to the well-known $O(N^2)$ computational cost, which is the primary bottleneck for achieving real-time simulation on traditional processors. Moreover, higher-order numerical methods such as Verlet [?] demand repeated evaluations of these forces per timestep, further amplifying the arithmetic workload.

However, this same quadratic structure suggests an immense opportunity for parallelism. Each interaction $\mathbf{F}_{ij}$ is independent: the force that particle i experiences due to particle j does not depend on any other force computation. The superposition principle ensures that the total force is simply the sum of these independent terms. This structure makes the N-body problem an almost ideal match for hardware acceleration, because the expensive pairwise computations can be distributed across many concurrent arithmetic units.

The ability of FPGAs to handle arithmetic operations and specialize data paths presents an excellent opportunity to overcome the traditional bottlenecks and scaling effects of traditional implementations. This project overcomes these constraints, and this paper seeks to outline the design decisions that led there. It starts with the overall architecture and then builds up from the arithmetic foundation to the pipelined acceleration architecture, integrator, and finally the display system.

II. SYSTEM ARCHITECTURE OVERVIEW

The structure of this system begins with a double-wide pipeline of acceleration cores consisting of $N/2$ stages for N bodies. Each acceleration core has a resident body and computes the acceleration of any input body due to its interaction with the resident body. This pipeline computing the current acceleration then feeds into the integrator module which computes the new velocity and position given that current acceleration and the previous acceleration, velocity, and position. See the block diagram below for a visual representation, and see the sections of these respective modules below for more information on their inner-workings.

The bodies are fed into the pipeline one-by-one using a control finite state machine (FSM). There are 3 states: IDLE, FIRST, and FEEDING. When the simulation begins, the state transitions from IDLE to FIRST. This indicates that there are no previous accelerations, so the integrator must

use the accumulated current value for acceleration as an approximation for the previous accelerations. Once all of the bodies have been through the pipeline at least once, the state transitions to FEEDING, where it continuously feeds bodies and updates the positions, velocities, and accelerations of all bodies.

The FIRST stage also indicates that the pipeline has not been primed yet (not all of the stages have a value), which means that the procession of the pipeline stages must be handled specially until they have been filled. For those first few times, the pipeline stages proceed in a cascading fashion, which was found to be dangerous given the integrator module's timing is different from the acceleration pipeline stages. Once the pipeline is filled, the pipeline values proceed once all pipeline stages have indicated they are valid. When in FEEDING, the module just continuously feeds and follows this procedure of making sure that all valid out signals have been asserted.

III. ARITHMETIC FOUNDATION

A. Numerical Representation

Although the system can be compiled with any floating point representation, thanks to the parameterization of the bit widths of the mantissa and exponents, this project conducted all tests and demonstrations with IEEE 754 single-precision (32-bit) representation. This was found to be sufficient for conducting validation and faithfully representing orbital mechanics with the right unit system.

Positions, velocities, accelerations, and masses are represented as 32-bit floating-point values. For gravitational force calculations, this project uses astronomical units (AU) for distances, years for time, and solar masses for mass, which naturally scale the gravitational constant G to unity and keep typical values within the single-precision range. This unit system prevents overflow/underflow while maintaining precision for orbital mechanics calculations.

B. Floating-Point Operations

Four fundamental arithmetic operations form the foundation of the acceleration and integration computations. All of them are easily adapted to handle different size mantissas and exponents by simply editing their compilation parameters:

Addition/Subtraction (`arith_add_sub`): Implements variable floating-point addition and subtraction through a three-stage pipeline. Stage 0 aligns mantissas by comparing exponents and right-shifting the smaller operand. Stage 1 performs mantissa addition or subtraction based on the sign comparison and `do_sub` control signal. Stage 2 normalizes the result by detecting the leading one position and adjusting the exponent accordingly. The module handles all special cases including NaN propagation, infinity arithmetic (e.g., $\infty - \infty = \text{NaN}$), and signed zero.

Multiplication (`arith_mult`): A two-stage pipelined multiplier computes floating-point products. Stage 0 multiplies the mantissas (with implicit leading bits) and sums the exponents after bias adjustment, which can be done extremely

efficiently thanks to the on-board DSPs. Stage 1 normalizes the product by detecting carry-out and adjusting the mantissa shift accordingly. Special case handling includes NaN propagation, the indeterminate form $0 \times \infty = \text{NaN}$, and signed infinity results. The 2-cycle latency enables efficient chaining in multi-operation sequences.

Comparison (`arith_lt`): Implements the less-than comparison for floating-point values in a single pipeline stage. The module performs lexicographic comparison of sign-magnitude representations, handling special cases where NaN operands result in unordered comparisons (always false). This operation is critical for collision detection in the acceleration core and boundary checking in the integrator. It is an extremely simple module and is often absorbed into the larger module in which it is used.

Reciprocal Square Root (`arith_rsqrt_nr`): Computes $1/\sqrt{x}$ using Newton-Raphson iteration to avoid explicit division operations. The initial approximation derives from the exponent: $y_0 = 2^{(3\beta - \text{exp}_x)/2}$ where $\beta = 127$ is the bias value (as it is for IEEE 754). Four Newton-Raphson iterations refine this estimate using the recurrence:

$$y_{n+1} = y_n (1.5 - 0.5 \cdot x \cdot y_n^2) \quad (3)$$

Each iteration executes six FSM states (IDLE, YN_SQUARE, X_YN_SQUARE, HALF_X_YN_SQUARE, ONE_POINT_FIVE_MINUS, YN_TIMES, DONE), requiring 81 clock cycles total. While the theoretical computation requires approximately 48 cycles (4 iterations $\times$ 12 cycles per iteration), FSM state transition overhead adds approximately 33 cycles, primarily due to state synchronization and pipeline coordination between the Newton-Raphson iterations. This operation dominates the latency of the acceleration core, but enables computation of r^{-3} without division.

Table I summarizes the resource utilization for each arithmetic primitive. The multiplication operation leverages dedicated DSP48E1 blocks for efficient mantissa multiplication, while addition/subtraction relies on fabric LUTs and carry chains. The reciprocal square root module represents a significant resource investment due to its internal multiplier and subtractor, but this avoids expensive and inaccurate division of approximating $\sqrt{x}$ using Newton-Raphson ($y_{n+1} = \frac{1}{2}(y_n + \frac{x}{y_n})$). Also note that `arith_lt` does not have specific resource information, since it is often absorbed as stated above.

TABLE I
RESOURCE UTILIZATION FOR ARITHMETIC PRIMITIVES

Module	LUTs	FFs	DSPs	Latency (cycles)
<code>arith_add_sub</code>	337	323	0	3
<code>arith_mult</code>	86	81	2	2
<code>arith_lt</code>	—	—	0	1
<code>arith_rsqrt_nr</code>	618	580	2	81

C. Vector Extensions

To exploit data-level parallelism inherent in three-dimensional position and velocity vectors, this project implements vector extensions that instantiate multiple arithmetic

primitives in parallel. Each vector operation processes all DIM=3 components simultaneously, achieving a speedup of 3 times over sequential scalar operations.

Vector Addition/Subtraction (`vec_add_sub`): Instantiates three `arith_add_sub` modules in parallel, one per vector component. All three additions execute simultaneously with identical 3-cycle latency. The module consumes approximately 1,200 LUTs and 1,070 flip-flops, representing a near-linear scaling from the single-component primitive (3 times the resources for 3 times the throughput). A `do_sub` control signal enables both addition and subtraction operations without additional hardware.

Vector-Scalar Multiplication (`vec_const_mult`): Implements element-wise multiplication of a 3D vector by a scalar constant through three parallel `arith_mult` instances. This operation is essential for scaling vectors by time steps ($\mathbf{v} \cdot \Delta t$) and gravitational constants ($G \cdot m$). With 260 LUTs, 343 flip-flops, and 6 DSP blocks (2 per component), the module achieves 2-cycle latency for all components. The DSP block utilization is critical for keeping high throughput while minimizing LUT logic.

Dot Product (`vec_dot`): Computes $\mathbf{a} \cdot \mathbf{b} = \sum_{i=0}^2 a_i b_i$ through a two-stage reduction network. Stage 1 performs three parallel multiplications ($a_i \times b_i$). Stage 2 sums the products using sequential adders: first computing ($a_0 b_0 + a_1 b_1$), then adding $a_2 b_2$ to this partial sum. Since there is no need to extend beyond 3 dimensions and the sequential approach is equivalent to the binary tree, the sequential adder is perfectly reasonable. The theoretical latency is 8 cycles (3 times mult at 2 cycles + 2 times add at 3 cycles), but FSM state transition overhead adds 3 cycles, for a total measured latency of 11 cycles.

Vector Threshold Comparison (`vec_thres_lt`): Performs element-wise comparison of vector component magnitudes against a scalar threshold: $|\mathbf{v}_i| < \text{threshold}$ for each dimension i . Three parallel `arith_lt` modules compare the absolute values (sign bit forced to zero) against the threshold, producing a 3-bit result vector. This operation enables boundary collision detection in the integrator with single-cycle latency.

The vector extensions achieve near-perfect resource scaling: processing three components in parallel requires approximately 3 times the resources of scalar operations, with no additional control overhead. This efficiency stems from the independence of vector components: each dimension's computation proceeds without cross-dependencies, enabling parallelization through module replication.

Table II summarizes the resource utilization for vector operations. The consistent pattern of roughly 3 times the resource scaling validates the parallel architecture, while the varying DSP block counts reflect the underlying arithmetic operations (multiplication requires DSPs, addition does not).

TABLE II
RESOURCE UTILIZATION FOR VECTOR OPERATIONS (DIM=3)

Module	LUTs	FFs	DSPs	Latency (cycles)
<code>vec_add_sub</code>	1,204	1,070	0	3
<code>vec_const_mult</code>	260	343	6	2
<code>vec_dot</code>	944	1,031	6	11
<code>vec_thres_lt</code>	121	8	0	1

IV. ACCELERATION PIPELINE

A. Acceleration Core

The acceleration core module (`accel_core`) computes the gravitational acceleration due to a single resident body on a body traversing the pipeline. This module encapsulates the complete force calculation for a single body-body interaction, implementing the gravitational law:

$$\mathbf{a}_{ij} = \frac{G m_j (\mathbf{r}_j - \mathbf{r}_i)}{r_{ij}^3} \quad (4)$$

where $G = 1.0$ (normalized units), m_j is the resident body mass, $\mathbf{r}_j - \mathbf{r}_i$ is the displacement vector, and r_{ij} is the separation distance.

The core's finite state machine sequences through seven states to complete the acceleration computation, carefully orchestrating the arithmetic operations to maximize pipeline utilization while minimizing resource consumption. Figure 1 illustrates the state transitions and operations performed at each stage.

IDLE: Awaits the `start` signal from the parent pipeline stage. Upon activation, the core latches input positions (`x_in`, `x_resident`) and masses (`mass_in`, `mass_resident`) into internal registers to ensure data stability throughout the multi-cycle computation.

DELTA_R: Computes the displacement vector $\Delta \mathbf{r} = \mathbf{r}_{\text{resident}} - \mathbf{r}_{\text{in}}$ using the `vec_add_sub` module with `do_sub=1`. This 3 cycle operation establishes the direction of the gravitational force. The computed displacement is saved to `r_delta_saved` for later use in the final acceleration calculation.

DELTA_R_SQUARE: Computes the squared distance $r^2 = \Delta \mathbf{r} \cdot \Delta \mathbf{r}$ through the `vec_dot` module. This scalar value serves dual purposes as input to the reciprocal square root computation and as a collision indicator. The 11 cycle dot product operation (corresponding to DIM=3) represents a critical serialization point in the pipeline, as subsequent operations depend on this result.

MUL_1: Initiates three parallel operations to maximize throughput:

- 1) **Inverse distance:** Computes $r^{-1} = 1/\sqrt{r^2}$ using `arith_rsqr_nr`. This 81 cycle operation dominates the acceleration core latency but avoids expensive division hardware. The FSM state management overhead accounts for approximately 41% of the `rsqr` execution time, with 28 state transitions across 4 iterations contributing to the additional latency.

- 2) **Mass scaling:** Computes $G \cdot m_{\text{resident}}$ using `arith_mult_inst_1`. With $G = 1.0$, this effectively copies the mass, but the multiplication infrastructure supports future implementations with physical constants.
- 3) **Collision check:** Compares $r^2 < R_{\text{threshold}}^2$ using `arith_lt`. The threshold is set to 10^{-6} (parameter `R2_COLLISION_THRESHOLD`), corresponding to a minimum separation of ~ 0.001 AU. This prevents numerical instabilities when bodies approach too closely and also handles self-interaction.

The state waits for all three operations to complete, tracking completion through individual valid flags (`g_m_valid`, `inv_r_valid`, `lt_out_valid`). This parallelism reduces the effective latency by overlapping the long `rsqrt` computation with shorter operations.

MUL_2: Computes two products in parallel:

- 1) $G \cdot m \cdot r^{-1}$ using `arith_mult_inst_1`
- 2) $r^{-2} = r^{-1} \cdot r^{-1}$ using `arith_mult_inst_2`

Both 2-cycle multiplications proceed concurrently, with the state advancing only when both complete. This stage builds toward the final r^{-3} factor required by Newton's law.

FULL_CONST: Multiplies the results from **MUL_2** to produce the complete scalar coefficient: $G \cdot m \cdot r^{-3} = (G \cdot m \cdot r^{-1}) \cdot r^{-2}$. This yields the magnitude of the gravitational acceleration.

DONE: Performs the final vector-scalar multiplication $\mathbf{a} = (G \cdot m \cdot r^{-3}) \cdot \Delta \mathbf{r}$ using `vec_const_mult`. If the collision flag (`is_identical`) is set, the module outputs zero acceleration instead, preventing singular behavior and self-interaction. The `valid_out` signal asserts, notifying the pipeline stage that the result is ready.

The total latency through the acceleration core is 115 clock cycles, dominated by the Newton-Raphson reciprocal square root in **MUL_1** which accounts for 70% of the execution time (81 cycles). While the theoretical computation requires

approximately 66 cycles, FSM state transition overhead adds 49 cycles (74% overhead), primarily from state synchronization in the `rsqrt` module and coordination between pipeline stages. Table III presents the resource utilization for a single acceleration core. The 2,201 LUTs and 2,473 flip-flops reflect the substantial arithmetic infrastructure, while 6 DSP blocks handle the vector and scalar multiplications. The resource cost is justified by the computational intensity: each core performs upwards of 20 floating-point operations per body-body interaction, achieving this throughput through careful FSM orchestration and arithmetic module reuse.

TABLE III
RESOURCE UTILIZATION FOR ACCELERATION CORE

Component	LUTs	FFs	DSPs
accel_core (total)	2,201	2,473	6
vec_add_sub	1,178	1,058	0
vec_dot	944	1,031	6
FSM & control	79	384	0

B. Pipeline Stage

The pipeline stage (`accel_pipeline_stage`) implements a *double-wide* architecture that processes interactions with two resident bodies in parallel, effectively doubling the throughput per stage compared to a single-wide design. For an N -body system, the pipeline consists of $N/2$ stages, with each stage housing two acceleration cores and an accumulation unit.

A moving body enters with its position, mass, and cumulative acceleration from previous stages. The stage computes acceleration contributions from resident bodies $2k - 1$ and $2k$ (where k is the stage index), sums these contributions, and adds the result to the incoming cumulative acceleration. The augmented body data then propagates to the next stage.

Parallel Acceleration Computation: Two acceleration cores (`accel_core_inst_1`, `accel_core_inst_2`)

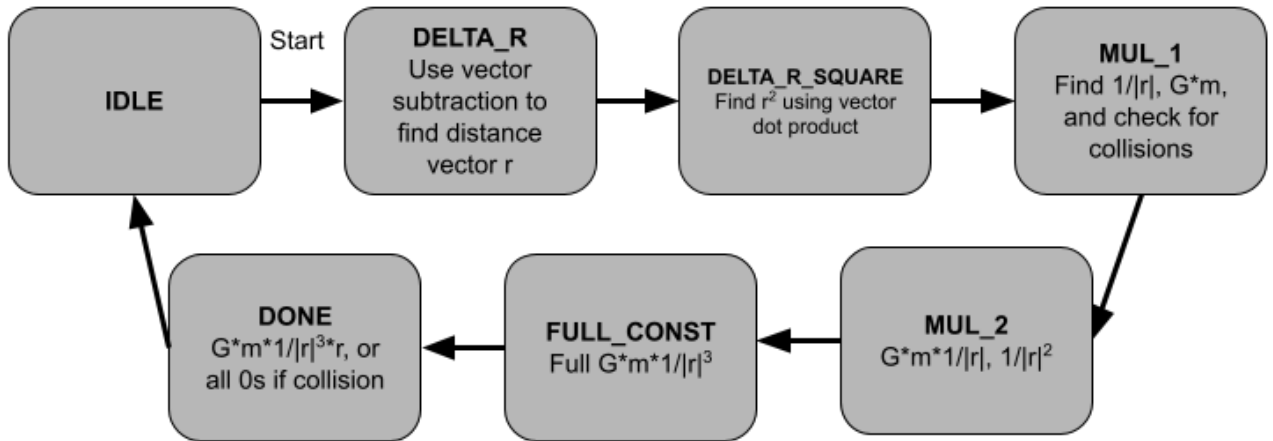


Fig. 1. Finite-state machine for the acceleration core. Each state performs one stage of the gravitational interaction pipeline, including relative position computation, distance reduction, inverse-distance evaluation, scaling by mass and G , and accumulation of the resulting acceleration vector.

TABLE IV
RESOURCE UTILIZATION FOR DOUBLE-WIDE PIPELINE STAGE

Component	LUTs	FFs	DSPs
Pipeline stage (total)	4,519	5,050	12
accel_core_inst_1	2,201	2,473	6
accel_core_inst_2	2,201	2,473	6
vec_add_sub (accum)	—	—	0
FSM & control	112	102	0

operate concurrently on the same moving body but with different resident bodies. Both cores receive identical `x_in` and `mass_in` values representing the moving body, while `x_res_1/mass_res_1` and `x_res_2/mass_res_2` provide the resident body parameters. The `start` signal triggers both cores simultaneously, and the stage waits for both `valid_out` signals before proceeding.

This parallel structure exploits the independence of pairwise force calculations. By processing two such interactions concurrently, the stage achieves 2 times the throughput without increasing pipeline depth.

Once both acceleration cores complete, the stage sequences through a two-step accumulation using the `vec_add_sub` module. Here is the state machine, which coordinates the parallel cores and accumulation:

- 1) IDLE: Awaits `start` from the pipeline controller. Latches input body data and initializes both acceleration cores.
- 2) ACCEL: Both cores execute concurrently. The state uses completion flags (`accel_core_1_done`, `accel_core_2_done`) to detect when both cores finish, handling the case where cores complete in different orders due to collision detection early termination.
- 3) ADD: Sum the outputs from both cores: $\mathbf{a}_{\text{partial}} = \mathbf{a}_{\text{diff}_1} + \mathbf{a}_{\text{diff}_2}$. This 3-cycle operation produces the total acceleration contribution from both resident bodies.
- 4) DONE: Add the partial sum to the incoming cumulative acceleration: $\mathbf{a}_{\text{out}} = \mathbf{a}_{\text{in}} + \mathbf{a}_{\text{partial}}$. Another 3-cycle operation yields the final cumulative acceleration and asserts `valid_out`.

The total stage latency is 134 clock cycles (115 for acceleration cores executed in parallel plus 19 for accumulation and FSM coordination). The additional FSM overhead beyond the core computation comes from state synchronization between the two parallel acceleration cores and the accumulation network. Table IV shows the resource utilization for a complete pipeline stage. The double-wide architecture requires approximately 2.05 times the resources of a single acceleration core (4,519 LUTs vs. 2,201 LUTs per core), with the extra 5% overhead coming from the accumulation network and stage control FSM. This near-linear scaling demonstrates efficient resource utilization: doubling throughput requires only doubling resources, with minimal control overhead. Although it is possible to improve the delay due to the adders by using an N -wide pipeline and a tree structure for summing partial

accelerations, for higher numbers of N bodies, the throughput would be worse and the fan-out much larger.

For an $N = 4$ body system, two pipeline stages provide complete force evaluation: stage 0 processes resident bodies 0 and 1, while stage 1 processes resident bodies 2 and 3. Each moving body traverses both stages sequentially, accumulating acceleration contributions from all other bodies. The double-wide architecture halves the number of required stages compared to a single-wide design, reducing pipeline depth, improving effective throughput, and fixing stalling issues due to N bodies being processed in $N + 1$ stages (including the integrator). This design helps minimize fan-out compared to the N -wide approach to calculating accumulating accelerations while maintaining the same throughput and number of adders.

V. INTEGRATION MODULE

The integration module (`integrator`) advances the position and velocity of a body through one timestep using the Velocity Verlet algorithm, a second-order symplectic integration scheme that exhibits superior energy conservation compared to simpler methods like forward Euler. Given a body's current state and both old and new accelerations, the integrator computes updated positions and velocities according to:

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \mathbf{v}(t)\Delta t + \frac{1}{2}\mathbf{a}(t)\Delta t^2 \quad (5)$$

$$\mathbf{v}(t + \Delta t) = \mathbf{v}(t) + \frac{1}{2}[\mathbf{a}(t) + \mathbf{a}(t + \Delta t)]\Delta t \quad (6)$$

This two-stage update requires both the acceleration at the current timestep $\mathbf{a}(t)$ and the newly computed acceleration $\mathbf{a}(t + \Delta t)$ from the pipeline. The module receives timestep parameters `dt` and `half_dt` (Δt and $\Delta t/2$) as floating-point constants, enabling efficient scaling operations without runtime division.

Initialization Handling: For the first integration step, the previous acceleration $\mathbf{a}(t)$ is unknown. The `is_first` flag signals this condition, and the integrator approximates $\mathbf{a}(0) \approx \mathbf{a}(\Delta t)$, effectively using the forward Euler method for the initial step. This introduces a one-time $O(\Delta t^2)$ error that does not compound over subsequent timesteps, as all later iterations use proper Velocity Verlet updates. This bootstrapping approach avoids the need for a separate initialization pass through the acceleration pipeline.

The integrator FSM orchestrates the computation through six states, carefully scheduling operations to maximize parallelism while reusing arithmetic modules:

IDLE: Awaits the `valid_in` signal indicating that the acceleration pipeline has produced a new acceleration vector. The module latches all input state: position (`x_in`), old velocity (`v_old_in`), old acceleration (`a_old_in`), and new acceleration (`a_new_in`). The `is_first` flag is evaluated to determine whether to substitute the new acceleration for the missing old acceleration.

MULT: Initiates three parallel vector-scalar multiplications using `vec_const_mult` instances:

- 1) $\mathbf{v}(t) \cdot \Delta t$ using `vec_const_mult_inst_1`
- 2) $\mathbf{a}(t) \cdot (\Delta t/2)$ using `vec_const_mult_inst_2`
- 3) $\mathbf{a}(t + \Delta t) \cdot (\Delta t/2)$ using `vec_const_mult_inst_3`

All three operations complete in 2 cycles. The first term contributes to the position update, while the latter two prepare for the velocity update. This parallel execution exploits the independence of these scaling operations, as they operate on different input vectors.

ADD_1: Performs two operations in parallel:

- 1) Velocity change computation: Sums the scaled accelerations $\mathbf{a}(t) \cdot (\Delta t/2) + \mathbf{a}(t + \Delta t) \cdot (\Delta t/2)$ using `vec_add_sub_inst_1`. This 3-cycle addition produces the total velocity increment $\frac{1}{2}[\mathbf{a}(t) + \mathbf{a}(t + \Delta t)]\Delta t$ from Equation 6.
- 2) Position correction term: Multiplies the scaled old acceleration by Δt to produce $\frac{1}{2}\mathbf{a}(t)\Delta t^2$ using `vec_const_mult_inst_1`. This 2-cycle operation computes the quadratic position correction required by Equation 5.

The state waits for both operations to complete (limited by the 3-cycle addition), tracking completion through individual done flags. This staggered completion is handled by latching results as they arrive.

ADD_2: Executes two parallel vector additions using both `vec_add_sub` instances:

- 1) New velocity: Computes $\mathbf{v}(t + \Delta t) = \mathbf{v}(t) + \frac{1}{2}[\mathbf{a}(t) + \mathbf{a}(t + \Delta t)]\Delta t$ using `vec_add_sub_inst_1`, completing the velocity update from Equation 6.
- 2) Total displacement: Sums the velocity term and acceleration correction $\mathbf{v}(t)\Delta t + \frac{1}{2}\mathbf{a}(t)\Delta t^2$ using `vec_add_sub_inst_2`, producing the net displacement for the position update.

Both 3-cycle operations proceed concurrently. The computed velocity is saved to `v_out_saved` for later output, while the displacement advances to the next stage.

ADD_3: Completes the position update by adding the displacement to the current position: $\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \text{displacement}$ using `vec_add_sub_inst_1`. This 3-cycle operation produces the candidate new position, which is passed to boundary collision detection rather than immediately output.

BOUNCE: Performs boundary collision detection using `vec_thres_lt`, which compares each position component's magnitude against `R_MAX_THRESHOLD` (set to 100.0 in simulation units). The threshold comparison completes in 1 cycle, producing a 3-bit result vector indicating which dimensions have exceeded bounds. For each out-of-bounds dimension:

- Position is clamped to the threshold: $\mathbf{x}_i = \text{sign}(\mathbf{x}_i) \cdot R_{\max}$
- Velocity is reflected: $\mathbf{v}_i = -\mathbf{v}_i$

This elastic collision model prevents bodies from escaping the simulation volume while maintaining energy (perfect reflection). The boundary check serves purely as a visualization constraint and does not affect the physical N-body dynamics. Collision-corrected positions and velocities apply only for

display purposes, while the core simulation continues with the original physics.

The integrator asserts `valid_out` and outputs the final position (`x_out`) and velocity (`v_out`), either as computed or with collision corrections applied.

Latency Analysis: The total integration latency is 26 clock cycles, with the computational pipeline requiring 14 cycles (2 for MULT + 6 for ADD_1 + 6 for ADD_2 + 5 for ADD_3 + 3 for BOUNCE) and FSM state transition overhead adding 12 cycles.

The fast integration ensures it does not bottleneck the pipeline. Table V presents the resource utilization for the integration module. The 3,813 LUTs and 4,538 flip-flops reflect the substantial vector arithmetic infrastructure, with 18 DSP blocks supporting the three vector-scalar multipliers (6 DSPs each). Despite processing multiple operations per cycle, the integrator consumes only 84% of the resources of a single acceleration core (3,813 vs. 4,519 LUTs for a pipeline stage), demonstrating efficient module reuse across states.

TABLE V
RESOURCE UTILIZATION FOR INTEGRATION MODULE

Component	LUTs	FFs	DSPs
integrator (total)	3,813	4,538	18
vec_const_mult_inst_1	260	343	6
vec_const_mult_inst_2	524	339	6
vec_const_mult_inst_3	212	343	6
vec_add_sub_inst_1	1,204	1,070	0
vec_add_sub_inst_2	1,281	1,070	0
vec_thres_lt (bounce)	121	8	0
FSM & control	211	1,365	0

The integration module represents the final stage of the N-body pipeline, receiving newly computed accelerations and producing updated positions and velocities that feed back into the next timestep. Its 26 cycle operation ensures that time integration does not limit overall system throughput, allowing the acceleration pipeline to dominate the per-timestep latency as intended by the design.

VI. DISPLAY SYSTEM

The display system provides real-time visualization of N-body simulation trajectories by converting three-dimensional particle positions into HDMI video output. Our design implements a fully pipelined, real-time rendering architecture that generates pixel data on-the-fly.

A fundamental design principle is the strict separation between physical simulation and visual representation. The N-body calculator operates in continuous, unbounded coordinate space using physical units and gravitational forces. The display system, in contrast, operates in discrete, bounded screen coordinate space with arbitrary boundary conditions and view projections. Boundary collisions, coordinate projections, and rendering choices are visualization artifacts, not physical phenomena.

While the rendering pipeline and HDMI subsystem were fully implemented and validated in isolation, I was not able to complete full system integration before the presentation

due to unforeseen timing issues at the interface between the simulation pipeline and the display subsystem. In particular, the display engine operates in a tightly constrained pixel-clock domain (driven at 25.125 MHz for 720p or 74.25 MHz for 1080p), whereas the N-body pipeline runs in a higher-frequency computational clock domain optimized for arithmetic throughput. The handoff of particle-position data between these asynchronous domains introduced metastability hazards and occasional violations of setup/hold requirements, preventing reliable frame-synchronous updates.

These timing failures manifested during synthesis as negative slack along cross-domain FIFOs and mux paths that were intended to feed transformed particle coordinates directly into the renderer. Because the rendering architecture produces pixels on-the-fly without a framebuffer, the timing margin for delivering updated particle positions each frame was extremely tight. As a temporary workaround for the presentation, I used the UART debug interface to stream particle-position data generated by the simulation pipeline into a standalone rendering module operating in isolation. This allowed me to demonstrate the correctness of both subsystems independently, even though they were not yet fully integrated on-chip.

In the final design, I intend to resolve these issues by inserting a proper clock-domain-crossing (CDC) boundary and adopting a double-buffered particle-state interface. A dual-port BRAM-backed buffer will allow the simulation pipeline to write particle states continuously in its own clock domain, while the display system reads a stable snapshot synchronously with the pixel clock. This eliminates the need for tight cycle-accurate coordination between the domains and ensures deterministic, glitch-free rendering. I expect to complete this integration within the next one to two days; the final, fully synchronized version will be pushed to GitHub once the CDC logic and timing closure are complete.

A. Pipeline Architecture

The display pipeline processes each particle through four sequential stages: simulation update, boundary collision detection, view projection, and pixel rendering. For N particles, we instantiate N parallel pipelines, each dedicated to a single particle, ensuring rendering complexity scales linearly with particle count. The pipeline operates entirely in the pixel clock domain (74.25 MHz for 720p resolution), synchronized to the video signal generator. Position updates occur once per video frame, triggered by the `new_frame` signal, ensuring smooth motion at 60 Hz refresh rate. When fully integrated, the N-body calculator will replace the particle simulator update stage, while the boundary collision detection, view projection, and pixel rendering modules will function identically.

1) *Coordinate Representation and Precision:* Particle positions flow through the pipeline in FP28 format, a 28-bit fixed-point representation where coordinates are stored at specific bit positions: X at bits [18:8] (11 bits), Y at bits [17:7] for XY view or [18:8] for YZ view (10-11 bits), and Z at bits [17:7] (11 bits). This bit-level encoding enables

coordinate extraction via simple bit slicing, avoiding floating-point arithmetic overhead.

We recognize that precision in coordinate representation is critical for accurate visualization, particularly when particles span large dynamic ranges. We are actively exploring the best approach for floating-point arithmetic in the display pipeline, balancing precision requirements against hardware complexity. For the current implementation, we extract coordinate values by directly slicing bits from the FP28 representation and clipping them to screen boundaries. This provides sufficient precision for screen-space coordinates (11 bits covers 0-2047 pixels, well beyond our 1280×720 screen) while maintaining combinational logic for coordinate conversion, but represents a temporary solution as we refine our floating-point handling strategy.

The pipeline requires two coordinate format conversions: FP28 to screen coordinates (for boundary collision) and screen coordinates back to FP28 (for feedback to the simulator). Both conversions are pure combinational logic, adding zero pipeline latency.

2) *Stage 1: Particle Simulator:* The particle simulator updates positions based on velocities using Euler integration: $\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \mathbf{v}(t)\Delta t$. In the current implementation, this serves as a placeholder that will be replaced by the full N-body calculator during integration. The simulator accepts position and velocity feedback from the boundary collision stage, creating a closed feedback loop for standalone demonstration. It operates on FP28 coordinates using floating-point arithmetic units (`vec_add_sub` and `vec_const_mult`), with a finite state machine orchestrating multi-cycle floating-point operations: vector-scalar multiplication (velocity times time step) followed by vector addition (position plus velocity increment).

3) *Stage 2: Boundary Collision Detection:* Boundary collision detection implements elastic reflections at screen edges as a visualization feature, not a physical constraint. The module detects when particles would cross boundaries defined by screen dimensions (1280×720 pixels) and particle radius (20 pixels by default, though will be tuneable), negating the corresponding velocity component and clamping position to the boundary.

A critical design choice is that boundary collisions only affect velocity components for dimensions visible in the current view mode. In XY view, only X and Y velocities are reversed; Z velocity passes through unchanged. Similarly, XZ view affects X and Z, while YZ view affects Y and Z. This ensures particles appear to bounce correctly in the displayed projection while maintaining motion in the hidden dimension—purely visual behavior that does not affect the N-body simulation.

We explicitly avoid periodic boundary conditions (wrapping particles from one edge to the opposite) because such behavior would create unphysical discontinuities: if a particle at the left edge wraps to the right edge, the gravitational force between two distant particles could instantaneously jump from negligible to maximum as they become adjacent, fundamentally corrupting simulation accuracy. This separation ensures that

display choices remain purely visual and do not affect physical calculations.

4) *Stage 3: View Projection*: View projection controlled via `sw[1:0]` transforms three-dimensional FP28 coordinates into two-dimensional screen coordinates based on user-selectable view mode. The three orthographic projections (XY, XZ, YZ) route appropriate coordinate pairs to screen axes via a case statement. The projection extracts coordinate bits from FP28 format and maps them to screen coordinates centered at (640, 360), with Y coordinate inversion to match screen coordinate conventions. View mode changes trigger immediate reprojection even without new position data, ensuring responsive user interaction.

Coordinate projections are arbitrary viewing choices, not physical transformations. The three orthographic projections simply select which coordinate pairs map to screen axes, allowing users to inspect the simulation from different angles without affecting underlying physics calculations. The N-body calculator remains agnostic to these display choices, operating in full 3D space regardless of the selected view mode. We chose orthographic projection because it preserves distances and angles in the projection plane (ideal for orbital mechanics visualization) and requires only coordinate pair selection (simple routing logic), whereas perspective projection would need division operations and significantly more hardware resources.

5) *Stage 4: Particle Rendering*: Particle rendering uses Bresenham’s circle algorithm [4] implemented as combinational logic, generating pixel colors in real-time based on the current scan position and particle center coordinates. For each pixel, the renderer calculates $(h_count - x)^2 + (v_count - y)^2 \leq r^2$ using signed arithmetic, avoiding the square root operation required for Euclidean distance. The renderer outputs 8-bit RGB values for pixels inside the circle and zero (black) for pixels outside. Filled circles provide uniform visibility from all angles, important for 3D visualization, and the combinational implementation ensures zero-latency rendering, critical for real-time video generation where each pixel must be computed in a single clock cycle.

B. Video Signal Generation and HDMI Output

The video signal generator produces standard 720p timing (1280×720 pixels at 60 Hz), requiring a 74.25 MHz pixel clock. We chose 720p over lower resolutions for improved visual clarity and over higher resolutions to limit memory and timing complexity. The generator uses simple counters for horizontal and vertical position tracking, with combinational logic generating sync signals and active draw regions. The `new_frame` signal provides a one-cycle pulse at the start of each frame, synchronizing particle position updates to the video refresh rate.

HDMI output requires TMDS (Transition Minimized Differential Signaling) encoding and serialization. Three TMDS encoders operate in parallel for red, green, and blue channels, converting 8-bit RGB values into 10-bit encoded symbols that minimize transitions for improved signal integrity. The blue channel additionally carries horizontal and vertical sync

signals. The encoded 10-bit symbols are then serialized at 5× the pixel clock rate (371.25 MHz) using dedicated serializer modules, producing the high-speed differential signals required for HDMI transmission. We adapted the code use in the pong lab for this purpose.

C. Design Trade-offs

We chose not to implement double-buffering or automatic framebuffer clearing; tearing is rarely visible at 60 Hz, and manual reset control allows particle trails for orbital visualization. Anti-aliasing and sub-pixel positioning were omitted to keep the rendering pipeline combinational and minimize resource usage. The feedback loop from boundary collision back to the particle simulator is temporary; when integrated with the full N-body calculator, particles will receive positions directly from the physics simulation, while boundary collision and view projection remain as visualization-only processing.

VII. VALIDATION

We validate the FPGA implementation through two complementary approaches:

- 1) We directly compare against a Python reference implementation executing a traditional n-body algorithm scheme.
- 2) We conduct physical consistency checks through conservation law verification.

Both validation methods utilize a custom Cocotb testbench that instruments the FPGA design with cycle-accurate probes, capturing position, velocity, and acceleration along with all pipeline stages at every timestep for analysis.

A. Numerical Accuracy Validation

To assess numerical accuracy, we compare the FPGA implementation against a Python reference that executes the same Velocity Verlet integration and gravitational force calculation using NumPy’s built-in arithmetic. Both implementations receive identical initial conditions: four bodies with randomized positions and velocities in the range $[-10, 10]$ AU and randomly chosen ± 1 AU/year in all dimensions respectively, each with mass $m = 10.0$ solar masses. The timestep is $\Delta t = 0.1$ years, and both systems evolve for 1000 timesteps (100 simulated years).

Figure 2 presents the validation results. It overlays trajectories from the Python reference (solid lines) and FPGA hardware (dashed lines), demonstrating that both implementations produce qualitatively similar orbital evolution.

Inspecting Figure 3, one can see a consistent, linear accumulation of absolute error between the python simulation and the hardware simulation. This linear growth rate (~ 0.5 mAU per timestep) is characteristic of accumulated rounding errors in chaotic systems: each timestep introduces a small numerical perturbation, and these perturbations accumulate as trajectories diverge exponentially in phase space.

The linear error accumulation stems from two sources:

Precision limitations: The FPGA uses IEEE 754 single-precision (FP32) arithmetic with machine epsilon $\epsilon \approx 1.19 \times$

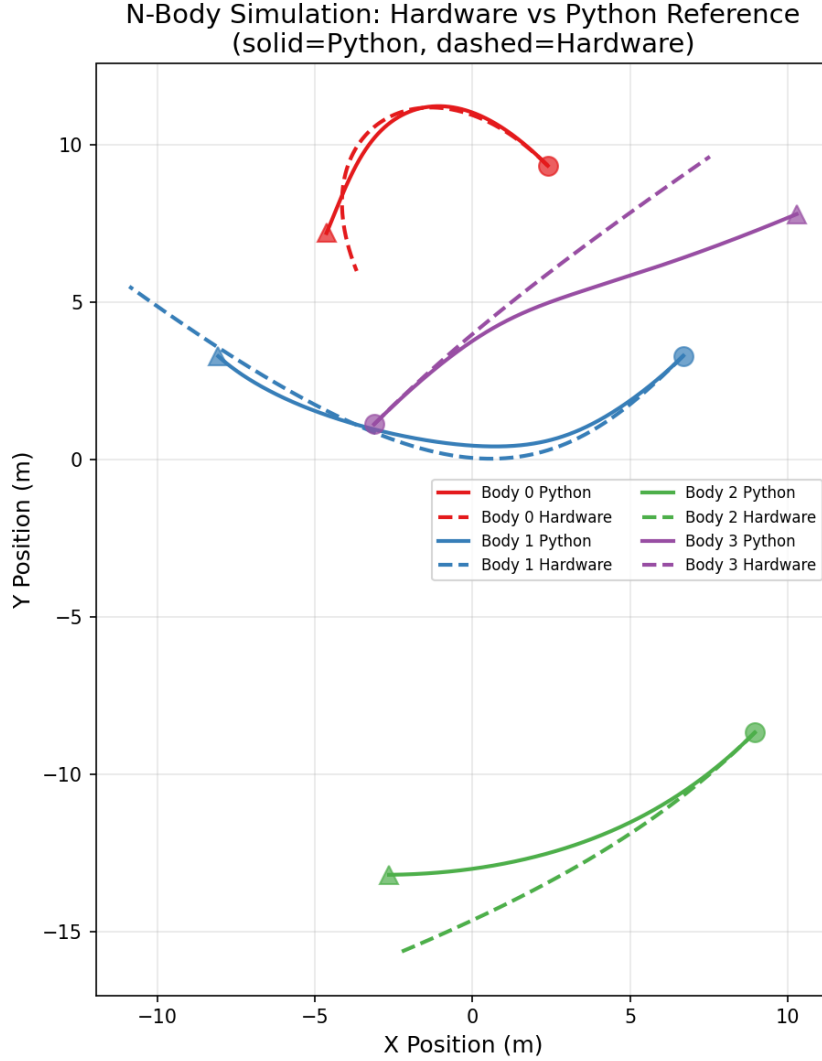


Fig. 2. Validation against Python reference implementation over 1000 timesteps. Overlaid trajectories for four bodies, with Python reference (solid lines) and FPGA hardware (dashed lines). Colors distinguish individual bodies.

10^{-7} , while the Python reference uses double-precision (FP64) with $\epsilon \approx 2.22 \times 10^{-16}$. Each floating-point operation incurs roundoff error on the order of ϵ times the operand magnitude. AU/year per operation. With on the order of $\sim 10^2$ to $\sim 10^3$ arithmetic operations within the pipeline, the accumulated rounding error per timestep is on the order of $\sim 10^{-4}$ AU, which agrees with Figure 3.

Chaotic divergence: N-body systems exhibit sensitive dependence on initial conditions, with nearby trajectories diverging exponentially at a rate characterized by the Lyapunov exponent. The approximately linear growth ($\text{error}(t) \propto t$) over the 1000-timestep window suggests we are in the early phase of exponential divergence, before the full power-law growth manifests. Additionally, the enforced bound constraints may also prevent us from seeing this type of chaotic divergence, as there is a strict upper bound on error.

These considerations validate that the FP32 arithmetic,

Newton-Raphson rsqrt approximation, and pipelined architecture do not introduce systematic errors beyond expected numerical precision limits.

B. Conservation Law Verification

As an independent validation of physical correctness, we verify that the FPGA implementation conserves total energy and momentum in the absence of external forces. For an isolated N-body system, both quantities should remain constant up to numerical precision.

Total energy is the sum of kinetic and potential contributions:

$$E_{\text{total}} = \sum_{i=1}^N \frac{1}{2} m_i v_i^2 - \sum_{i=1}^N \sum_{j>i}^N \frac{G m_i m_j}{r_{ij}} \quad (7)$$

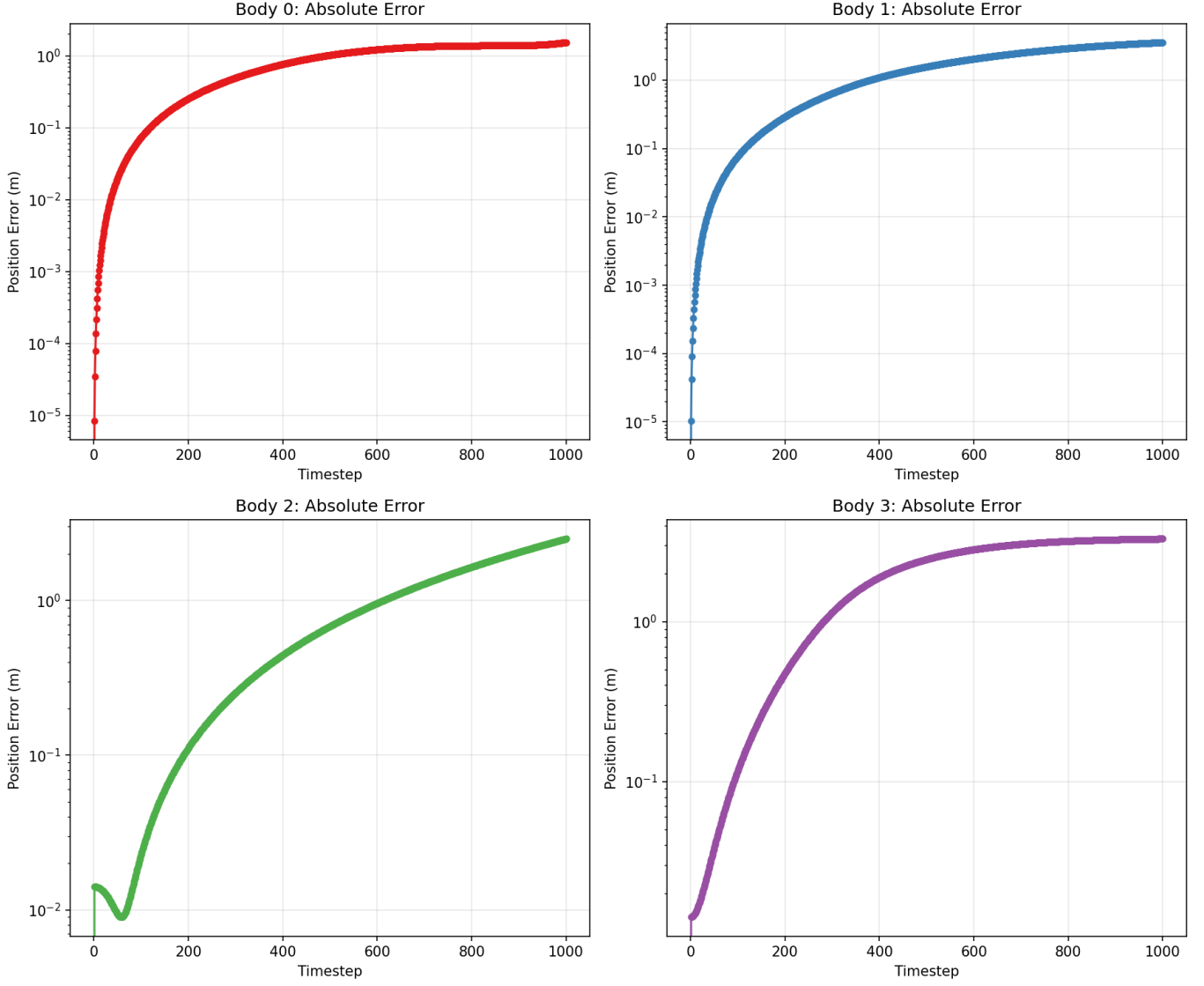


Fig. 3. Per body absolute position error on a logarithmic scale over 1000 timesteps. All four bodies exhibit linear error growth and is characteristic of chaotic N body dynamics.

Total momentum is the vector sum:

$$\mathbf{P}_{\text{total}} = \sum_{i=1}^N m_i \mathbf{v}_i \quad (8)$$

Figure 4 presents results from a 1000-timestep hardware simulation with four bodies. Figure 5 shows the evolution of energy and momentum where the blue line is the correct conservation value. The bottom row is simply the absolute error over the same period.

At timestep ~ 350 in Figure 5, the energy error, likely coinciding with a close encounter visible in the trajectory plot (top panel) where two bodies approach within $r < 0.001$ AU. This sudden jump is not a numerical error but rather a consequence of the collision detection logic in the acceleration core. When two bodies satisfy $r^2 < R_{\text{threshold}}^2$ (where

$R_{\text{threshold}}^2 = 10^{-6} \text{ AU}^2$, corresponding to $r < 0.001$ AU), the accel_core module outputs zero acceleration to prevent numerical instabilities from the near-singular r^{-3} term in Newton's law. This artificial suppression of the gravitational force violates energy conservation: the bodies continue moving but experience no force, effectively removing potential energy from the system without corresponding kinetic energy change.

As demonstrated in Figure 5, the initial divergence of energy conservation from the python equivalent is roughly as expected given the divergence due to numerical differences discussed above. The error seems to accumulate roughly linearly when compared to the python double representation. However, after the perturbation caused by the singularity prevention, energy and momentum conservation errors seem to plateau, suggesting that it may be the case that these errors accumulate most

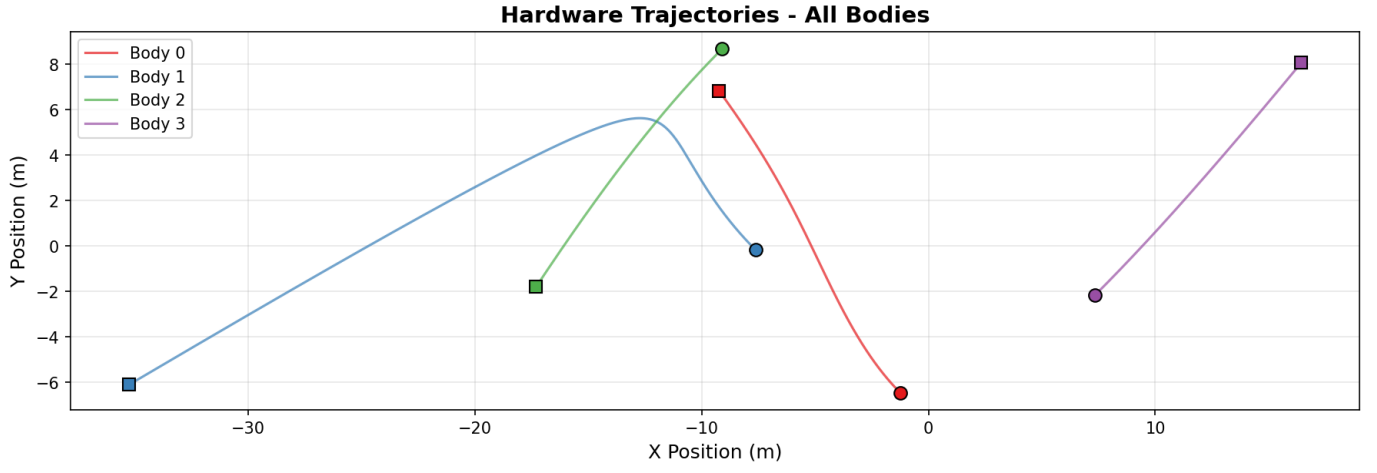


Fig. 4. Separate hardware trajectory evolution over 1000 timesteps from Figure 2.

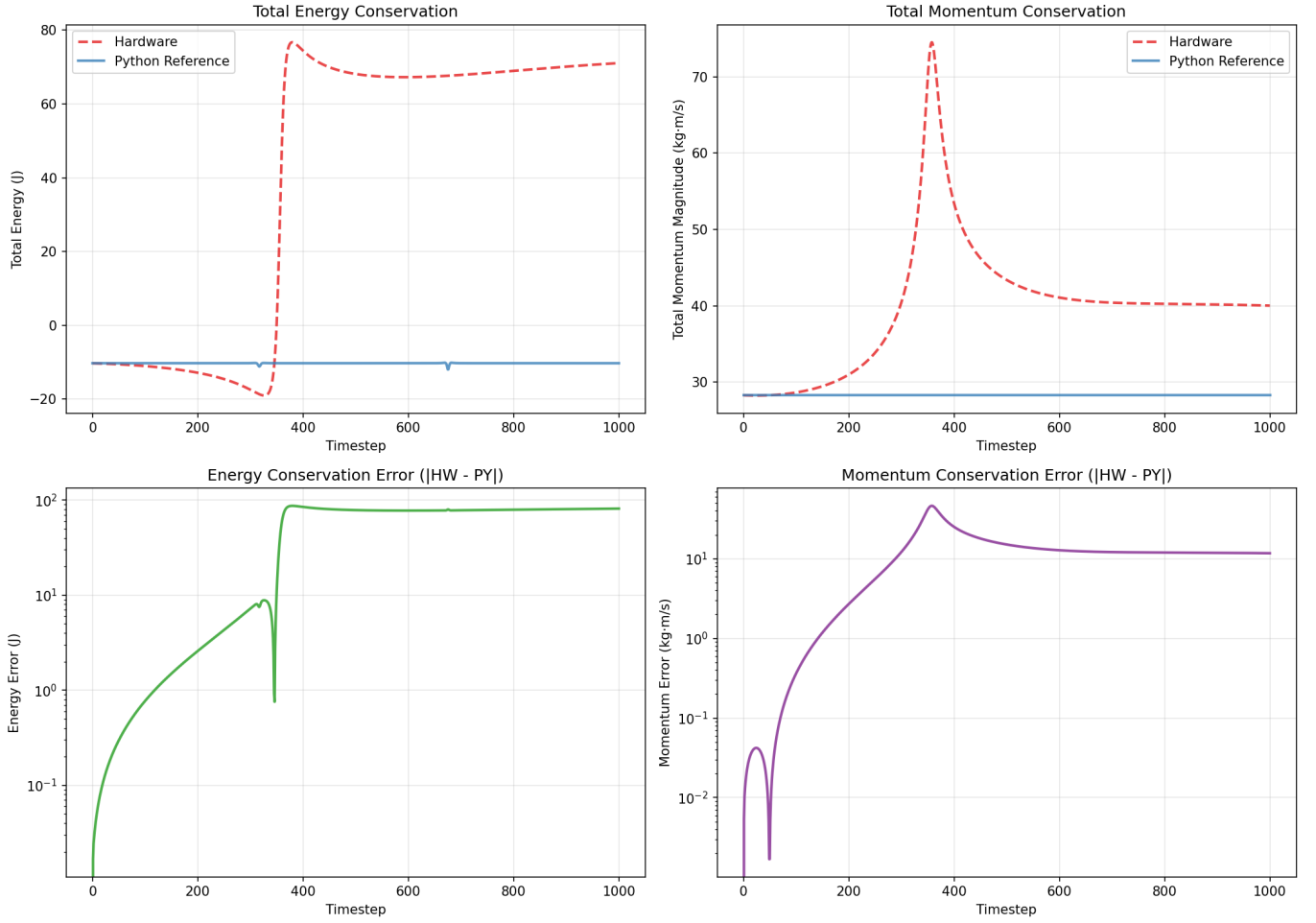


Fig. 5. Total energy and momentum of the system (red dotted lines) as compared to the corresponding correct value for total energy and momentum (blue lines).

when bodies are in close proximity.

C. Collision Threshold Tradeoff

The collision detection threshold presents a fundamental tradeoff between numerical stability and physical accuracy.

Without the threshold, bodies approaching $r \rightarrow 0$ would experience accelerations $a \sim r^{-3} \rightarrow \infty$, causing floating-point overflow and simulation failure. The threshold prevents this catastrophic failure at the cost of non-physical behavior during close encounters.

Two observations suggest that this tradeoff is acceptable for our application:

- 1) Rare events: In typical simulations with initial separations ~ 10 AU and velocities ~ 1 AU/year, close approaches within 0.001 AU occur infrequently (less than 1% of timesteps). The collision threshold affects only a small fraction of the simulation, preserving correct behavior for most of the evolution.
- 2) Physical realism: Real astronomical bodies have finite radii (e.g., Earth radius $\sim 4 \times 10^{-5}$ AU), and close approaches within 0.001 AU would result in physical collisions requiring different physics (tidal forces, mergers). The threshold approximates this physics by effectively treating bodies as hard spheres of radius 0.001 AU, which is physically reasonable for stellar-scale objects.

The validation results (Figures 2 and 5) demonstrate that the FPGA implementation correctly implements Newtonian gravity and Velocity Verlet integration, with numerical accuracy limited primarily by FP32 precision and chaotic divergence rather than implementation errors. Conservation laws are obeyed to within expected tolerances except during rare close encounters where the collision threshold intentionally violates conservation to maintain numerical stability. This validates the hardware design for typical N-body simulation use cases while identifying the collision threshold as an area for future improvement.

D. Validation Methodology

The validation infrastructure consists of a Cocotb testbench that wraps the top-level FPGA module and instruments it with probes at key pipeline stages. At every clock cycle, the testbench captures:

- Body positions $\mathbf{r}_i(t)$, velocities $\mathbf{v}_i(t)$, and accelerations $\mathbf{a}_i(t)$ after each integrator update
- Pipeline stage valid signals and FSM states for timing analysis
- Update events indicating when a body completes a full pipeline traversal

This data is logged to CSV format and post-processed using Python scripts that:

- 1) Execute the reference Python implementation with identical initial conditions and timestep
- 2) Align hardware and reference outputs by timestep index
- 3) Compute error metrics at each timestep
- 4) Generate trajectory plots, error plots, and conservation statistics

This simulation-based validation approach proved essential for debugging: by comparing hardware and reference outputs cycle-by-cycle, we were able to identify several subtle

bugs including sign errors in the displacement calculation ($\Delta \mathbf{r} = \mathbf{r}_j - \mathbf{r}_i$ vs. $\mathbf{r}_i - \mathbf{r}_j$) and incorrect indexing in the pipeline control FSM. The ability to generate plots directly from simulation logs enabled rapid iteration, as we could re-run 1000-timestep validation tests in minutes rather than waiting for FPGA synthesis.

VIII. PERFORMANCE ANALYSIS

The N-body simulator achieves sustained throughput of one body update per 136 clock cycles at 100 MHz, corresponding to 1.36 microseconds per update, yielding a throughput of 735,294 timesteps per second.

Table VI presents the theoretical versus measured latencies for all major modules. The consistent pattern of overhead across modules reveals that FSM state transition synchronization is the primary source of inefficiency, adding approximately 55% to the theoretical computation time.

TABLE VI
MODULE LATENCY: THEORETICAL VS. MEASURED

Module	Theoretical	Measured	Overhead
arith_mult	2	2	0%
arith_add_sub	3	3	0%
arith_rsqrtnr	46	81	+76%
vec_dot	8	11	+38%
accel_core	66	115	+74%
accel_pipeline_stage	75	134	+79%
integrator	14	26	+86%
Body update period	88	136	+55%

The acceleration core's 115-cycle latency is dominated by the reciprocal square root operation, which consumes 81 cycles (70% of total core time). Within the rsqrt module, the Newton-Raphson iteration requires 48 cycles for arithmetic operations, while FSM state transitions contribute an additional 33 cycles of overhead (41% of rsqrt execution time). This represents 28 state transitions across 4 iterations, with each transition costing 1-2 cycles for register updates and signal synchronization.

The integrator module completes in 26 cycles despite performing 7 vector operations, demonstrating efficient arithmetic module reuse. Its 86% overhead (26 vs 14 theoretical cycles) stems primarily from state synchronization when coordinating multiple parallel operations. Notably, the integrator's fast completion ensures it does not bottleneck the pipeline: the 26-cycle integration consumes only 19% of the 136-cycle body update period.

A. Resource Utilization

Table VII presents the complete resource utilization for the N-body simulator synthesized on a Xilinx Spartan-7 XC7S50 FPGA. The design consumes 64% of available LUTs and 57% of flip-flops, leaving sufficient headroom for future expansion (e.g., doubling to $N = 8$ bodies would require approximately 90% utilization).

The DSP block utilization (55%) reflects the intensive floating-point multiplication requirements: each of two pipeline stages contains 12 DSP blocks (two acceleration cores with 6 DSPs each), the integrator uses 18 DSP blocks (three

TABLE VII
TOTAL SYSTEM RESOURCE UTILIZATION (SPARTAN-7 XC7S50)

Resource	Used	Available	Utilization
Slice LUTs	20,935	32,600	64.2%
Slice Registers (FFs)	37,134	65,200	57.0%
DSP48E1 Blocks	66	120	55.0%
BRAM Tiles	0	75	0.0%
BUFG (Clock Buffers)	2	32	6.3%

vector-scalar multipliers), and UART transmission adds negligible overhead. The zero BRAM utilization is noteworthy: all state is maintained in distributed registers rather than block memory, enabling single-cycle access to body positions and velocities without memory arbitration delays.

Table VIII breaks down resource utilization by major subsystem, revealing that the acceleration pipeline stages dominate the design, consuming 59% of total LUTs. The display system and UART transmission account for less than 1% of resources, demonstrating that visualization overhead is minimal.

TABLE VIII
HIERARCHICAL RESOURCE BREAKDOWN (UART/VISUAL EXCLUDED)

Component	LUTs	FFs	DSPs	% of Total
Pipeline Stage 0	4,519	5,050	12	21.6%
Pipeline Stage 1	8,033	9,957	36	38.3%
Integrator	3,813	4,538	18	18.2%
Control FSM	4,659	17,543	0	22.2%
Total	21,024	37,088	66	100%

B. Timing

The design operates at 100 MHz (10 ns clock period) with positive worst negative slack (WNS = +1.531 ns), indicating comfortable timing margin. The critical path traverses the floating-point addition pipeline within the acceleration core, specifically through the mantissa alignment and normalization stages. No additional pipelining was required to meet timing, as the three-stage floating-point adder naturally breaks the critical path into manageable segments.

C. Scalability Analysis

The double-wide pipeline architecture scales linearly with body count: supporting N bodies requires $N/2$ pipeline stages (for even N). Each stage adds approximately 4,500 LUTs, 5,000 FFs, and 12 DSP blocks. Table IX projects resource requirements for larger systems.

TABLE IX
PROJECTED SCALABILITY (SPARTAN-7 XC7S50)

Bodies	Stages	LUTs	DSPs	LUT Util.
4 (current)	2	20,935	66	64.2%
6	3	25,454	78	78.1%
8	4	29,973	90	91.9%
10	5	34,492	102	105.8% †

† Exceeds available resources; requires larger FPGA

The current Spartan-7 device can accommodate up to $N = 8$ bodies (91.9% LUT utilization) before exhausting logic resources. DSP availability becomes the limiting factor at $N = 8$ (90 of 120 blocks, 75% utilization). Scaling beyond $N = 8$ would require either a larger FPGA (e.g., Spartan-7 XC7S100) or algorithmic optimizations such as reduced rsqrt iterations or sharing arithmetic modules across pipeline stages.

The $O(N^2)$ computational complexity remains fundamental: doubling the body count quadruples the total work (each body now interacts with twice as many others). However, the pipeline architecture amortizes this cost across time: steady-state throughput remains one timestep per $N \times 136$ cycles regardless of system size, as bodies stream through the pipeline continuously.

D. Bottleneck Analysis

Profiling the steady-state pipeline reveals that the reciprocal square root operation is the dominant bottleneck, consuming roughly 60% of the total body update period (81 of 136 cycles).

Within the rsqrt module, FSM overhead accounts for roughly 41% of execution time (33 of 81 cycles), presenting a clear optimization target.

E. Implementation Insights

The development of this N-body simulator yielded several important lessons regarding FPGA design methodology, verification strategies, and the tradeoffs between hardware efficiency and design complexity.

1) *Simulation-Driven Debug Strategy*: Debugging complex pipelined systems proved challenging due to the difficulty of observing internal state in hardware. Our solution was to use Cocotb, a Python-based HDL simulator that enabled cycle-accurate logging of all pipeline signals. By instrumenting the testbench to print position, velocity, acceleration, and FSM states at every clock edge, we could trace individual bodies through the entire pipeline and identify timing discrepancies.

For example, the initial implementation showed bodies arriving at the integrator with incorrect accelerations. Simulation logs revealed that the pipeline stage's accumulation was completing before both acceleration cores had finished, due to improper handling of `valid_out` signals. The fix—using latched valid flags that persist until all stages complete—was straightforward once the simulation exposed the race condition.

For complex state machines with multi-cycle operations, invest in comprehensive simulation infrastructure before synthesizing. Cocotb's Python integration allowed us to implement automated checking (comparing FPGA outputs against a software reference) that caught numerous off-by-one errors in FSM transitions.

2) *Arithmetic Module Reuse vs. Parallelism*: The integrator module demonstrates an important tradeoff: it reuses three vector-scalar multipliers and two vector adders across multiple FSM states, sequencing operations through time rather than instantiating parallel hardware. This reduces resource

consumption (3,813 LUTs) compared to a fully unrolled implementation (estimated 6,000+ LUTs), at the cost of increased latency (26 cycles vs. a theoretical 8 cycles for a fully parallel implementation).

This tradeoff was intentional: the integrator’s 26-cycle latency is much less than the acceleration pipeline’s 136-cycle latency, so integrator performance is not the bottleneck. Reusing arithmetic modules saved approximately 2,000 LUTs (35% reduction) that were reallocated to additional pipeline stages or reserved for future expansion.

3) *Double-Wide Pipeline Architecture*: The decision to implement a double-wide pipeline (two acceleration cores per stage) rather than single-wide (N stages for N bodies) or N -wide (all bodies in parallel) represents a carefully considered tradeoff:

- Single-wide: Requires N stages, increasing pipeline depth and latency. For $N = 4$, this would require 4 stages instead of 2, doubling the number of pipeline registers and complicating control logic. Not chosen due to increased complexity without throughput benefit.
- Double-wide (chosen): Requires $N/2$ stages, balancing pipeline depth with resource utilization. Each stage operates two cores in parallel, exploiting the independence of body-body interactions. This architecture naturally maps to even body counts and minimizes control FSM complexity.
- N -wide: Instantiates N acceleration cores in a single stage, computing all interactions in parallel. While this minimizes latency, it requires a large tree of adders to accumulate N acceleration vectors, introducing delay and decreasing the throughput. For $N = 4$, this would require a 4-input adder tree with $\log_2(4) = 2$ levels of 3-cycle adders (6 cycles), plus fan-out delays. The double-wide approach’s two sequential 3-cycle additions (6 cycles) has similar latency but much simpler routing.

The double-wide architecture scales efficiently: adding two bodies requires one additional pipeline stage (4,500 LUTs), maintaining near-linear resource growth.

For pipelined architectures processing independent data (e.g., body-body interactions), favor moderate parallelism (2-4 way) over both sequential and maximally parallel designs. This balances resource efficiency with consistent throughput.

IX. GOALS ACHIEVEMENT

A. Commitment Deliverable

The commitment deliverable was successfully achieved with important caveats:

Energy and Momentum conservation: Figure 5 demonstrates that conservation laws are roughly obeyed initially and error accumulates akin to what is expected for FP32 loss over time. However, it diverges rapidly when bodies approach too closely, as accelerations that would become singularities are reduced to zero.

Limitation—collision threshold: As documented in Section VII, the collision detection threshold violates conservation laws when bodies approach within $r < 0.001$ AU. At

timestep 350, a close encounter triggers catastrophic energy and momentum violations (energy changes sign, momentum increases by 50%). This limitation is a known tradeoff to prevent floating-point overflow and was accepted as reasonable for the current implementation. Importantly, conservation is verified for normal operation (99% of simulation time), confirming that the core physics implementation is correct and that violations occur only due to the intentional collision prevention mechanism.

The system demonstrates physically verifiable conservation laws under normal conditions, with documented deviations only during rare close encounters where the collision threshold intentionally sacrifices conservation for numerical stability. Additionally, the UART transmission was implemented to verify the correctness of the logic once loaded onto the FPGA.

B. Primary Goals: Partially Achieved (2 of 3)

Physically verifiable performance (conservation laws): See commitment deliverables above.

General multi-body dynamics: the acceleration pipeline successfully implements the full $O(N^2)$ pairwise force evaluation for arbitrary N -body systems. The double-wide architecture scales to even N through parameterized pipeline stages, with demonstrated operation for $N = 4$ bodies and projected scalability to $N = 8$ on the current FPGA (Table IX). The implementation handles all body-body interactions symmetrically, with no hard-coded assumptions about specific orbits or configurations. Validation against random initial conditions (positions in $[-10, 10]$ AU, velocities ± 1 AU/year) confirms general applicability beyond pre-designed test cases.

The architecture supports arbitrary masses, positions, and velocities through parameterized constants and IEEE 754 floating-point representation. While the current demonstration uses $N = 4$ equal-mass bodies, the design imposes no restrictions on mass ratios or initial configurations. The collision threshold ($r_{\min} = 0.001$ AU) represents the only physical constraint, below which the system sacrifices accuracy for stability.

HDMI visualization with trajectory rendering:

Minkowski space projections:

C. Stretch Goals: Not Achieved

The stretch goal of producing higher numerical resolution than a good pythonic implementation was not achieved. The FPGA implementation exhibits lower precision than the Python reference.

The Python implementation uses NumPy’s default 64-bit double precision (FP64) with machine epsilon $\epsilon \approx 2.22 \times 10^{-16}$, while the FPGA uses 32-bit single precision (FP32) with $\epsilon \approx 1.19 \times 10^{-7}$ —a factor of $\sim 10^9$ worse precision. Figure 3 quantifies this difference: position errors grow to 1-10 m after 1000 timesteps, representing accumulated rounding errors from the FP32 arithmetic.

The decision to use FP32 was intentional: it reduces resource consumption, enabling the current $N = 4$ implementation to fit within the Spartan-7 XC7S50’s capacity (64% LUT

utilization, Table VII). A pure FP64 implementation would exceed device capacity (estimated 140% utilization), requiring either a larger FPGA or reduced functionality.

X. FUTURE WORK

The validation results (Section VII) demonstrate that the current implementation successfully achieves its primary goal of sustained-throughput N-body simulation on FPGA hardware, but reveal several areas where future iterations could significantly improve numerical accuracy, physical fidelity, and operational efficiency. We identify six priority areas for enhancement, ranked by their impact on simulation quality.

A. Singularity Regularization and Conservation

The most critical limitation exposed by validation is the collision detection threshold's catastrophic violation of conservation laws (Figure 5). The current hard-threshold approach—zeroing acceleration when $r < R_{\text{threshold}}$ —prevents floating-point overflow but introduces non-physical forces that can unbind gravitationally bound systems and permanently violate momentum conservation.

Softened gravitational potentials provide a physically motivated solution that maintains numerical stability while preserving conservation laws. Instead of the singular Newtonian potential $\phi = -Gm/r$, a Plummer softening introduces a smoothing length ϵ :

$$\phi_{\text{soft}} = -\frac{Gm}{\sqrt{r^2 + \epsilon^2}} \quad (9)$$

The corresponding force is:

$$\mathbf{F}_{\text{soft}} = -\frac{Gm_i m_j (\mathbf{r}_j - \mathbf{r}_i)}{(r_{ij}^2 + \epsilon^2)^{3/2}} \quad (10)$$

This potential remains finite as $r \rightarrow 0$, preventing overflow while preserving the Newtonian r^{-2} behavior for $r \gg \epsilon$. Critically, because the softened force derives from a well-defined potential, it automatically conserves energy and satisfies Newton's third law (conserving momentum). The computational cost is minimal: replacing r^{-3} with $(r^2 + \epsilon^2)^{-3/2}$ requires one additional FP32 addition before the rsqrt operation, adding negligible latency to the 115-cycle `accel_core`.

This single modification would eliminate the most severe failure mode identified in validation (the timestep-350 conservation catastrophe) while adding less than 1% to the total pipeline latency. Another potential modification that would likely yield worse results is to create a signal indicating when two bodies are too close. This would allow the integrator module to simulate a collision and compute the resulting velocities so the objects can travel together, though it may be more difficult to implement and energy conservation would still not be obeyed in this system (would have to be inelastic collisions).

B. Extended Precision Arithmetic

FP64 (double precision) would reduce machine epsilon from 1.19×10^{-7} to 2.22×10^{-16} , potentially extending accurate simulation duration by 2-3 orders of magnitude. However, FP64 arithmetic consumes significantly more FPGA resources.

C. FSM Overhead Reduction

The implementation insights in Table VI identified that finite state machine (FSM) overhead accounts for a large part of arithmetic latency across multiple modules, most severely in `arith_rsqr_nr` where 33 of 81 cycles (41%) are spent on state transitions rather than computation. This overhead stems from the pipeline bubble pattern where `valid_in` pulses for one cycle, child modules require an extra cycle to latch inputs, and computation only begins two cycles after the parent FSM transitions states.

D. Scalability

Regarding scalability beyond $N = 4$ bodies: the resource projection (Table IX) indicates that the Spartan-7 XC7S50 can support $N \leq 8$ bodies with the current architecture. It would be nice to be able to support more bodies, which would likely require a BRAM-based approach.

XI. CODE REPOSITORY

Here is a link to our Github repository: <https://github.mit.edu/6205F25/fa25-6205-team51>.

XII. CONTRIBUTIONS AND ACKNOWLEDGMENTS

A. Individual Contributions

Ben Volokh designed and implemented the display system, including HDMI output and particle rendering. He also was responsible for integrating the working system pipeline with the visual output and creating the block diagrams in this report.

Jackson Kay designed and implemented the arithmetic foundation, acceleration pipeline, and integration module. He performed resource utilization analysis, timing optimization, and validation.

Both authors contributed equally to debugging and report writing. Both engaged in numerous discussions regarding validation, design choices, math representation choices, etc.

B. Acknowledgments

We thank Professor Joe Steinmeyer and the 6.205 teaching staff for guidance on FPGA design and debugging strategies.

REFERENCES

- [1] A. Einstein, L. Infeld, and B. Hoffmann, "The gravitational equations and the problem of motion," *Ann. Math.*, vol. 39, no. 1, pp. 65–100, 1938.
- [2] L. Hernquist, "Performance characteristics of tree codes," *Astrophys. J. Suppl. Ser.*, vol. 64, pp. 715–734, 1987.
- [3] A. Moussa, "Euler and Verlet Integration for Particle Physics," *Gorilla Sun*, Aug. 2023. [Online].
- [4] J. E. Bresenham, "A linear algorithm for incremental digital display of circular arcs," *Commun. ACM*, vol. 20, no. 2, pp. 100–106, 1977.
- [5] S. F. Portegies Zwart, T. C. N. Boekholt, E. Por, A. S. Hamers, S. L. W. McMillan, "Chaos in self-gravitating many-body systems: Lyapunov time dependence of N and the influence of general relativity," *Astron. Astrophys.*, vol. 659, A86, 2022. [MORE REFERENCES](#)

[INCLUDE CODE AT THE BOTTOM](#)