Design Document: Large-Scale N-Body Accelerator

Lucy He (lh3365), Xiyuan Peng (xp2236), Jingzeng Xie (jx2668),
Pengpeng Wang (pw2660), Charlotte Chen (hc3558)

Spring 2026

Contents

1 Introduction

The goal of this project is to design and implement an FPGA-based hardware accelerator for gravitational N-body simulation, a classic $O(N^2)$ problem (1) in which each body interacts with every other body through pairwise forces. Our system targets the force-computation and time-update loop of a 2D gravity simulation. The hardware consists of four parallel two-body compute cores with pipelined datapath. Each core computes pairwise acceleration contributions using a fast inverse square root unit, followed by accumulation. A leapfrog integration step then updates particle velocities and positions over time. The host processor handles initialization, user interaction, and display coordination via an Avalon memory-mapped interface. The resulting simulation will be visualized on a VGA display as an interactive 2D gravity simulator.

2 System Block Diagram

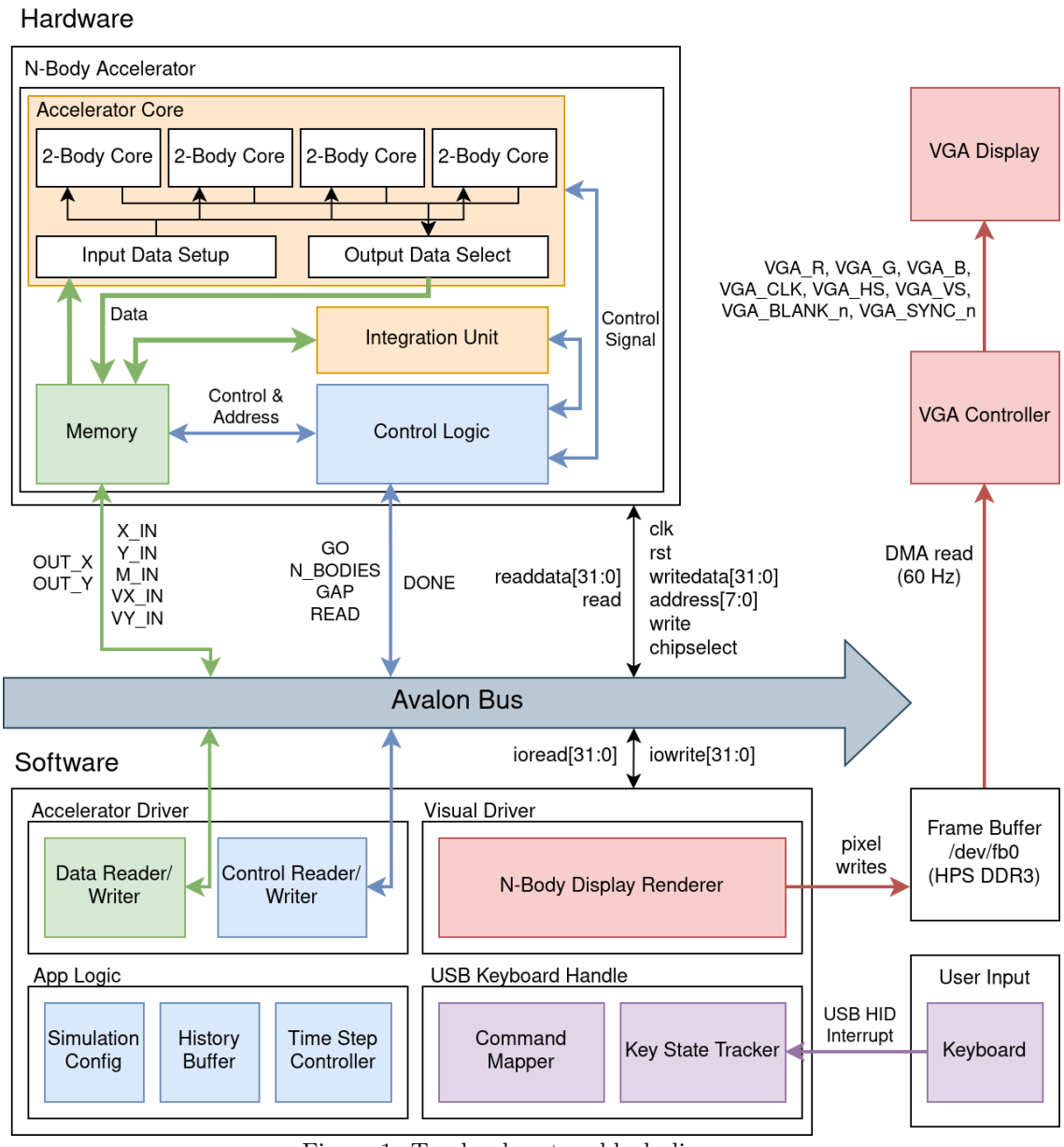

Figure 1: Top-level system block diagram.

3 Algorithms

3.1 The N-Body Problem

The N-body problem concerns the simulation of a system of particles in which each body is affected by the forces generated by all other bodies. In gravitational N-body simulation, the net force on each particle is computed by summing all pairwise interactions, and the resulting force is then used to update the particle's acceleration, velocity, and position over time. In the direct all-pairs formulation, each of the N bodies interacts with the other $N-1$ bodies, leading to a computational complexity of $O(N^2)$ (1). In the N-body simulation, the evolution of the system begins with the standard Newtonian gravity equation. For two bodies with masses m_i and m_j, the magnitude of the gravitational force is given by:

$$|F_{ij}| = G\frac{m_i m_j}{r_{ij}^2}, \tag{1}$$

where G is the gravitational constant and r_{ij} denotes the distance between the two bodies.

By applying Newton's second law, the acceleration induced by body j on body i can be written as

$$a_{ij} = \frac{F_{ij}}{m_i} = G\frac{m_j}{r_{ij}^2}. \tag{2}$$

After summing the contributions from all other bodies, the simulation obtains the total acceleration vector of body i. This acceleration is then used to advance the dynamical state of the system over a small discrete time step dt. More specifically, the updated velocity is computed by integrating the current acceleration over dt, and the updated position is subsequently obtained from the new velocity. To express both magnitude and direction in vector form, the relative displacement vector is introduced as $r_{ij} = r_i - r_j$. Accordingly, the acceleration contribution from body j to body i is written as

$$a_{ij} = G\frac{m_j\, r_{ij}}{\|r_{ij}\|^3}. \tag{3}$$

In practical simulation, a softening factor ϵ^2 is further added to avoid numerical instability when two bodies become too close. Therefore, the softened pairwise interaction is expressed as

$$a_{ij} \approx G\frac{m_j\, r_{ij}}{(\|r_{ij}\|^2 + \epsilon^2)^{3/2}}. \tag{4}$$

The total acceleration acting on body i is then obtained by summing the contributions from all other bodies:

$$a_i \approx G\sum_{j\neq i}\frac{m_j\, r_{ij}}{(\|r_{ij}\|^2 + \epsilon^2)^{3/2}}. \tag{5}$$

3.2 Hardware

3.2.1 Acceleration Decomposition

For the two-dimensional implementation adopted in this work, the acceleration vector is further decomposed into its Cartesian components. Let $\Delta x_{ij} = x_i - x_j$ and $\Delta y_{ij} = y_i - y_j$, then the squared distance between bodies i and j is

$$r_{ij}^2 = (\Delta x_{ij})^2 + (\Delta y_{ij})^2. \tag{6}$$

Accordingly, the softened pairwise acceleration can be expressed component-wise as

$$a_{ij,x} \approx Gm_j \frac{\Delta x_{ij}}{(r_{ij}^2 + \epsilon^2)^{3/2}}, \qquad a_{ij,y} \approx Gm_j \frac{\Delta y_{ij}}{(r_{ij}^2 + \epsilon^2)^{3/2}}. \tag{7}$$

The total acceleration of body i is then obtained by summing these contributions over all interacting bodies in the x and y directions separately. This decomposition is used directly in hardware, where the accelerator computes the two directional components before passing them to the integration stage.

3.2.2 Fast Inverse Square Root

The core computational bottleneck in the N-body force calculation is evaluating the term

$$g(r^2) = \frac{1}{(r^2 + \varepsilon^2)^{3/2}},$$

which requires both a square root and a division. On general-purpose hardware, dedicated square root and division units are expensive in terms of area and latency, making them poorly suited to a deeply pipelined FPGA design. Instead, we adopt the well-known Fast Inverse Square Root (FISR) algorithm (2; 3), originally popularized by the *Quake III Arena* source code, which approximates $1/\sqrt{x}$ using only integer bit manipulation, addition, and multiplication, which map efficiently onto the DSP blocks and LUTs of the Cyclone V.

Step 1: Integer Interpretation of a Float

The key insight is that the bit pattern of an IEEE 754 single-precision float encodes an approximation of its own base-2 logarithm. A 32-bit floating-point number is stored as three fields $[S \mid E \mid M]$, where S is the sign bit, E is the 8-bit biased exponent, and M is the 23-bit mantissa:

$$x = (-1)^S \cdot 2^{E-127} \cdot \left(1 + \frac{M}{2^{23}}\right)$$

Its integer reinterpretation is:

$$I = S \cdot 2^{31} + E \cdot 2^{23} + M$$

This gives the approximation:

$$\log_2(x) \approx \frac{I}{2^{23}} - 127$$

This approximation holds because the exponent field E directly encodes the integer part of $\log_2(x)$, while the mantissa M contributes a piecewise-linear approximation of the fractional part. As shown in Figure 2, the integer reinterpretation tracks $\log_2(x)$ closely across several orders of magnitude, with only small periodic errors arising from the linear approximation within each floating-point binade.

Step 2: Magic Number

To compute $y \approx 1/\sqrt{x}$, we require $\log_2(y) \approx -\frac{1}{2}\log_2(x)$. Substituting the integer approximation from Step 1 and solving for I_y gives:

$$I_y \approx c_0 - \frac{I_x}{2},$$

where the magic constant $c_0 = 0x5f3759df$ accounts for the floating-point bias and a small correction term σ that minimizes the worst-case approximation error (3):

$$c_0 = \frac{3}{2} \cdot 127 \cdot 2^{23} - \frac{\sigma}{2} \cdot 2^{23}.$$

The original *Quake III* implementation of this step in C is:

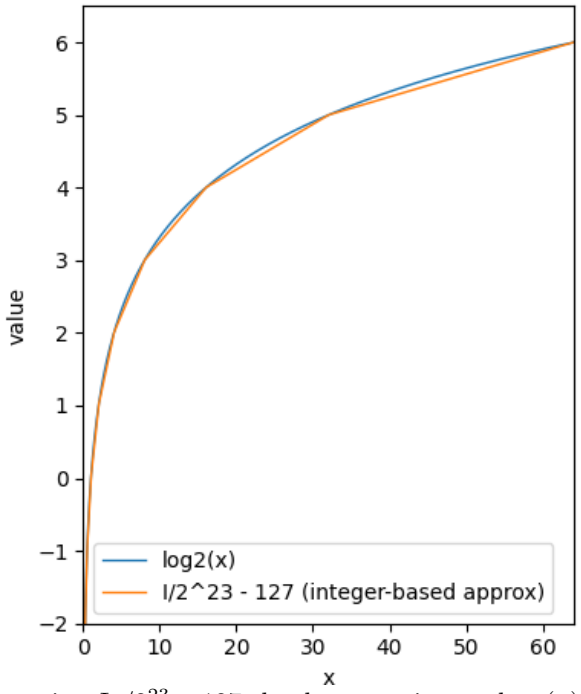

Figure 2: The integer reinterpretation $I_x/2^{23} - 127$ closely approximates $\log_2(x)$ across a wide range of inputs. The small periodic deviation within each binade arises from the piecewise linear nature of the mantissa encoding.

```
float Q_rsqrt(float x) {
    float x2 = 0.5f * x;
    int i    = *(int*)&x;
    i        = 0x5f3759df - (i >> 1);
    float y  = *(float*)&i;
    y        = y * (1.5f - x2 * y * y);
    return y;
}
```

Step 3: Newton's Method

The initial estimate from Step 2 is accurate to within a few percent but requires refinement for use in physical simulation. We apply one iteration of Newton's method (4) to improve the estimate. Newton's method finds the root of a function $f(y) = 0$ by iteratively updating an initial guess y_0 along the tangent line at that point, as illustrated in Figure 3:

$$y_{n+1} = y_n - \frac{f(y_n)}{f'(y_n)}.$$

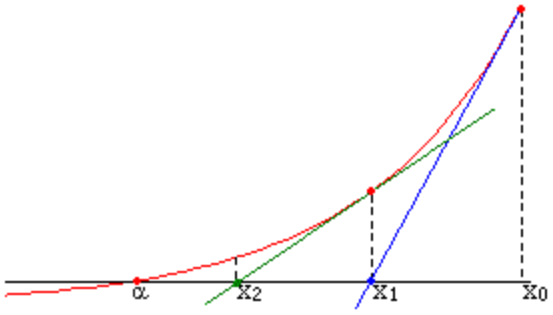

Figure 3: Newton's method iteratively refines an initial guess x_0 toward the root α of f by following the tangent line at each iterate. One iteration is sufficient to achieve the precision required by our floating-point format.

Setting $f(y) = 1/y^2 - x = 0$, whose root is $y = 1/\sqrt{x}$, with $f'(y) = -2/y^3$, one Newton–Raphson iteration gives:

$$y_{n+1} = y_n \left(1.5 - 0.5 \cdot x \cdot y_n^2\right).$$

This correction requires two multiplications and one subtraction — all cheap floating-point operations with no division or square root hardware involved.

3.2.3 Pipelined Two-Body Core

Each two-body core implements the full softened pairwise acceleration computation as a single fully pipelined datapath, accepting a new body pair every clock cycle (5). The pipeline proceeds in five stages, illustrated in Figure 4:

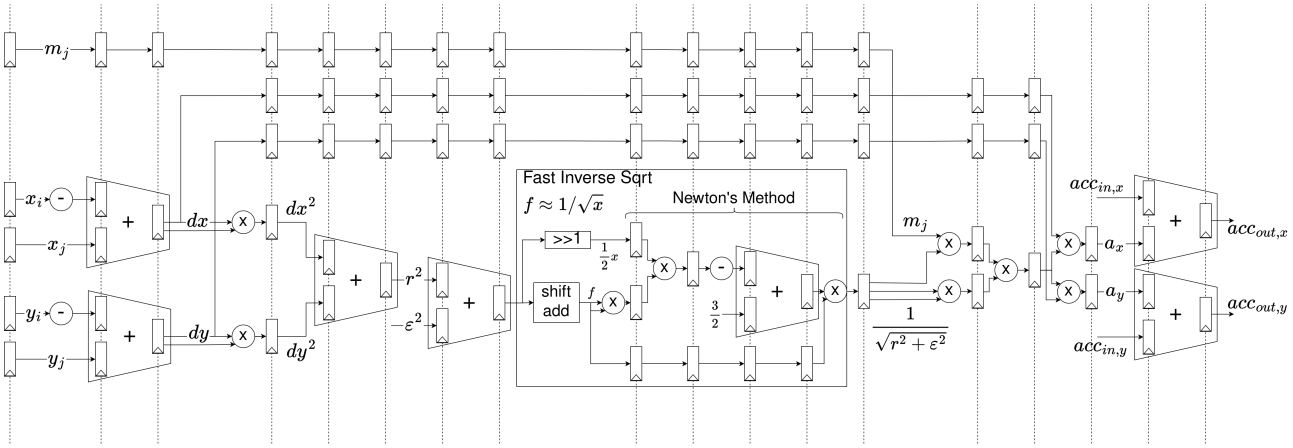

Figure 4: Pipelined datapath of a single two-body compute core. Inputs x_i, x_j, y_i, y_j, m_j enter from the left; the Fast Inverse Square Root unit occupies the central stages; and the final stages scale and accumulate the pairwise acceleration contributions into `acc_out_x` and `acc_out_y`. Vertical dashed lines denote pipeline stage boundaries; rectangular blocks denote pipeline registers.

1. **Displacement:** Compute $\Delta x = x_i - x_j$ and $\Delta y = y_i - y_j$ in parallel using floating-point subtractors.
2. **Squared distance:** Compute Δx^2, Δy^2, sum them, and add ε^2 to obtain the softened $r^2 + \varepsilon^2$.
3. **Fast inverse square root:** Apply the FISR pipeline (Section 2.2.2) to compute $f \approx 1/\sqrt{r^2 + \varepsilon^2}$, then multiply f^2 and apply one Newton–Raphson correction to obtain $f^3 \approx (r^2 + \varepsilon^2)^{-3/2}$.
4. **Force scaling:** Multiply f^3 by m_j to obtain the scalar force coefficient, then multiply separately by Δx and Δy (carried forward via pipeline registers) to obtain a_x and a_y.
5. **Accumulation:** Add a_x and a_y to the running partial sums `acc_in_x` and `acc_in_y`, producing `acc_out_x` and `acc_out_y`.

The inputs m_j, x_i, x_j, y_i, y_j are propagated through pipeline registers alongside the intermediate results so that each stage operates on temporally consistent data. Four such cores operate in parallel, each assigned a distinct body pair at each clock cycle by the Input Data Setup block.

3.2.4 Leapfrog Integration Algorithm

To update the position and velocity of each body over time, we adopt the leapfrog integration algorithm in hardware (6), replacing the simpler Euler method. Unlike the Euler update:

$$v_{new} = v_{old} + a \cdot dt$$

$$pos_{new} = pos_{old} + v_{new} \cdot dt$$

the leapfrog method splits each timestep into three stages:

Step 1: Update velocity by a half timestep using the current acceleration:

$$v_{i+1/2} = v_i + \frac{1}{2} a_i \cdot dt$$

Step 2: Update position using the half-step velocity:

$$x_{i+1} = x_i + v_{i+1/2} \cdot dt$$

Step 3: Compute new acceleration at x_{i+1}, then complete the velocity update:

$$v_{i+1} = v_{i+1/2} + \frac{1}{2} a_{i+1} \cdot dt$$

This approach uses the mid-interval velocity to update position, yielding better long-term accuracy than Euler integration and preserving energy conservation over extended simulations.

Critically, the leapfrog algorithm is well-suited to hardware implementation: Steps 1, 2, and 3 each require only simple multiply-add operations on position and velocity, while the acceleration computation embedded between Steps 2 and 3 is handled by the two-body cores. The velocity and position updates are therefore implemented directly in hardware alongside the acceleration computation pipeline, adding negligible area overhead while eliminating the need for host-side integration.

3.3 Software

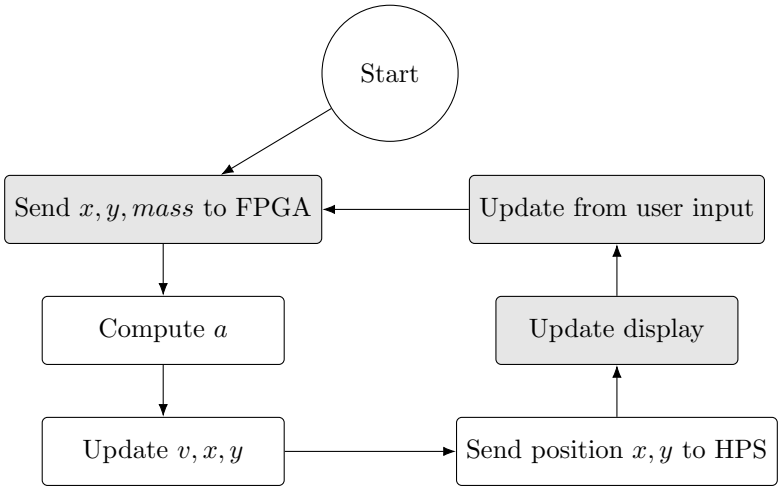

Figure 5: High-level workflow of the system with gray blocks representing the software part.

The high-level design of the project is that, at each simulation step, the HPS software sends the current body data, including position and mass, to the FPGA. The FPGA performs the main physics computation by calculating the gravitational acceleration and velocity, updating the body states. Once the computation is finished, the updated position data are returned to the HPS. The software uses the updated position data to refresh the display output and visualize the current state of the simulation.

It also monitors user input and applies any requested changes to the simulation, such as modifying parameters or updating object behavior. After these software-side updates are processed, the revised data are sent back to the FPGA for the next simulation cycle. In this way, the FPGA handles the computational kernel, while the software manages rendering, interaction, and overall simulation control.

4 Resource Budgets

4.1 Custom 27-bit Floating-Point Format

The choice of floating-point precision directly determines how efficiently the design maps onto the Cyclone V DSP blocks (7). The Cyclone V 5CSEA5 FPGA provides 174 independent 18 × 18 bit multipliers in its DSP blocks. The available multiplier configurations and device resource counts are shown in Figure 6.

Variant	Member Code	Variable-precision DSP Block	Independent Input and Output Multiplications Operator			18 x 18 Multiplier Adder Mode	18 x 18 Multiplier Adder Summed with 36 bit Input
			9 x 9 Multiplier	18 x 18 Multiplier	27 x 27 Multiplier		
Cyclone V E	A2	25	75	50	25	25	25
	A4	66	198	132	66	66	66
	A5	150	450	300	150	150	150
	A7	156	468	312	156	156	156
	A9	342	1,026	684	342	342	342
Cyclone V GX	C3	57	171	114	57	57	57
	C4	70	210	140	70	70	70
	C5	150	450	300	150	150	150
	C7	156	468	312	156	156	156
	C9	342	1,026	684	342	342	342
Cyclone V GT	D5	150	450	300	150	150	150
	D7	156	468	312	156	156	156
	D9	342	1,026	684	342	342	342
Cyclone V SE	A2	36	108	72	36	36	36
	A4	84	252	168	84	84	84
	A5	87	261	174	87	87	87
	A6	112	336	224	112	112	112

Figure 6: dsp_blocks_table

The IEEE 754 single-precision floating point uses a 23-bit stored mantissa (24 bits including the implicit leading 1). A 24×24 mantissa multiplication exceeds the native 18×18 DSP width and must be decomposed using the Karatsuba method:

$$24 = 18 + 6 \implies \text{requires 4 DSP blocks per multiplier}$$

since the four partial products (18×18, 18×6, 6×18, 6×6) each occupy one DSP block.

Our custom 27-bit format uses 1 sign bit, 8 exponent bits, and an 18-bit mantissa. An 18×18 mantissa multiplication maps exactly onto a single Cyclone V DSP block with no decomposition required. This is the key motivation for the 27-bit format: it is the largest mantissa width that fits within one native DSP block, maximizing precision per DSP block used.

The resulting multiplier density comparison is:

Format	**DSP blocks per multiplier**	**Max parallel multipliers**
IEEE 754 32-bit	4	$\lfloor 174/4 \rfloor = 43$
Custom 27-bit	1	174

This gives a 4× improvement in available multiplier count, directly translating to more parallel two-body compute cores on the FPGA fabric.

4.2 DSP Budget for the Accelerator

Each pipelined two-body core requires the following floating-point multipliers:

Operation	**Multipliers**
Squared distance (dx^2, dy^2)	2
FISR Newton step ($y \cdot y$, $x \cdot y^2$, $y \cdot c$)	3
f^3 computation ($f \cdot f$, $f^2 \cdot f$)	2
Force scaling ($f^3 \cdot m_j$, $\cdot dx$, $\cdot dy$)	3
Total per core	**10**

The Integration Unit requires an additional 4 multipliers for the leapfrog velocity and position updates ($v \cdot dt$ for x and y, twice per half-step).

With 4 parallel two-body cores plus the Integration Unit:

$$(4 \times 10) + 4 = 44 \text{ DSP blocks}$$

This leaves $174 - 44 = 130$ DSP blocks available, providing significant headroom for additional parallelism or future design changes. DSP utilization is therefore not a limiting constraint for this design.

4.3 Memory Usage

In regards to the memory required, the input size will be $n \times 5 \times 27$ bits, where n is the number of bodies, 5 is the number of values per body (x position and velocity, y position and velocity, and mass), and 27 bits is due to our custom floating-point format. In addition, n bits will be required for a bit vector map that dictates whether or not a given body is "active" in the system. During the actual calculations, previous states will be discarded as new ones are calculated, since they will no longer be needed. An additional $4 \times n \times 27$ bits will be needed during calculation to store temporary values: specifically, the half-step velocities ($v_{i+\frac{1}{2}}$) in both x and y directions produced in the first stage of leapfrog integration, and the accumulated accelerations (a_{i+1}) in both x and y directions computed during force accumulation. Both must reside in memory simultaneously before the final velocity update can be completed. Finally, output memory is $n \times 4 \times 27$ bits, as well as the same n-bit vector map, since mass is not required in the output.

Thus, we can estimate the total memory usage as:

$$(n \times 5 \times 27) + n + (4 \times n \times 27) + (n \times 4 \times 27) + n$$

$$= n \times (135 + 1 + 108 + 108 + 1)$$

$$= 353n \text{ bits}$$

For $n = 1024$ bodies:

$$353 \times 1024 = 361{,}472 \text{ bits} \approx 353 \text{ Kbits}$$

This corresponds to:

$$\left\lceil \frac{361{,}472}{10{,}240} \right\rceil = 36 \text{ M10K blocks}$$

Remaining memory:

$$397 - 36 = 361 \text{ M10K blocks}$$

$$361 \times 10{,}240 \approx 3.70 \text{ Mbits}$$

Even for simulations with 1024 bodies, the memory usage remains minimal compared to the available capacity. Only about 7% of the total on-chip memory is utilized.

This demonstrates that memory is not a limiting factor in the design, and the system can scale to significantly larger problem sizes. The primary constraint remains the number of available DSP multipliers, making the design compute-bound.

5 The Hardware/Software Interface

5.1 Accelerator Interface

Table 1: Register Map

Address	Register	R/W	Description
0x00	GO	W	Pulse high to start computation. Implicitly resets both input and output body pointers to 0.
0x01	N_BODIES	W	Number of active bodies in the simulation.
0x02	GAP	W	Number of timesteps executed internally between successive DONE assertions.
0x03	X_IN	W	Input X position for the current body.
0x04	Y_IN	W	Input Y position for the current body.
0x05	M_IN	W	Input mass for the current body.
0x06	VX_IN	W	Input X velocity for the current body.
0x07	VY_IN	W	Input Y velocity for the current body.
0x08	DONE	R	Set high by hardware upon completion of GAP timesteps. Cleared when software lowers READ.
0x09	READ	W	Asserted high by software after DONE is observed. Lowered by software once all outputs have been read, signalling readiness for the next GO.
0x0A	OUT_X	R	Output X position for the current body.
0x0B	OUT_Y	R	Output Y position for the current body.

The accelerator is controlled through a memory-mapped Avalon interface. Each transaction transfers one 32-bit word; for 27-bit floating-point values, bits [26:0] carry the data and bits [31:27] are ignored on write and return zero on read.

Body data are written sequentially through dedicated input registers. For each body, software must write the five input fields in the fixed order `X_IN`, `Y_IN`, `M_IN`, `VX_IN`, `VY_IN`. Writing `VY_IN` is treated as the completion of the current body entry; the hardware commits the assembled body data into internal memory and automatically increments the internal input pointer to the next body.

The `N_BODIES` register specifies the number of valid bodies in the simulation and is used by the accelerator as the loop bound for force computation and position update.

Output readback is symmetric with input streaming. After DONE is asserted, software reads OUT_X followed by OUT_Y for each body in sequence. Reading OUT_Y increments the internal output pointer to the next body. Both the input and output pointers are implicitly reset to zero when GO is asserted. Since the highest register address in the map is 0x0B, the address space requires 4 bits, and the interface is configured with an 8-bit address port on the Avalon bus.

5.2 User Interface

Listing 1: C Header for N-Body Accelerator Interface

```
/* Input/Output Data Types */
typedef struct {
    uint32_t x;
    uint32_t y;
    uint32_t vx;
    uint32_t vy;
    uint32_t m;
} body_t;

typedef struct {
    uint32_t x;
    uint32_t y;
} body_pos_t;

/* Accelerator Driver API */
// Initialization
void accel_set_n_bodies(uint32_t n);
void accel_set_gap(uint32_t gap);

// Data input (sequential, pointer auto-increments on each call)
void accel_write_body(body_t *b);
void accel_load_bodies(body_t *bodies, uint32_t n);

// Control
void accel_start(void);       // asserts GO, resets pointers
bool accel_is_done(void);     // polls DONE
void accel_wait_done(void);   // blocking poll until DONE

// Output (sequential, pointer auto-increments on each call)
void accel_read_body(body_pos_t *out);
void accel_read_all(body_pos_t *out, uint32_t n);
void accel_ack_read(void);    // asserts then lowers READ
```

The application is controlled entirely through a USB keyboard. Upon initialization, the user configures the simulation parameters: the number of bodies and the GAP value controlling how many timesteps the hardware executes between successive readbacks. Software randomly initializes the position, velocity, and mass of each body and streams them to the accelerator before asserting GO to begin the simulation.

The following key mappings are supported during simulation:

Key	Action
P	Pause / Resume the simulation
R	Reset to initial state with new random bodies
↑ / ↓	Increase / decrease GAP (simulation speed)
← / →	Step backward / forward through history by one readback frame
Q	Quit the simulation

Table 2: Keyboard controls for the simulation

Navigation through time is made possible by saving all positions returned by the hardware accelerator in a position history buffer in software. This allows the user to scrub forwards and backwards through previously computed frames without requiring the hardware to recompute them. Note that backward navigation is limited to the frames retained in the history buffer.

The VGA display visualizes the current state of the simulation, rendering all active bodies at their current positions as filled circles scaled by mass. A text overlay on the display indicates the current timestep index and simulation state (running or paused).

5.3 Display Interface

Rather than implementing a custom VGA peripheral on the FPGA fabric, the display subsystem uses the Linux framebuffer device `/dev/fb0` provided by the Lab 2 base image. The VGA Controller reads pixel data directly from HPS DDR3 memory via a dedicated DMA path at 60 Hz, completely independent of the Avalon bus. This frees the Avalon bus exclusively for accelerator communication and eliminates the need for any display-related hardware on the FPGA.

5.3.1 Framebuffer Access

The framebuffer represents a 640×480 pixel display at 4 bytes per pixel (R, G, B, unused), totalling approximately 1.2 MB. Software maps it into the process address space once at startup:

```
int fb_fd = open("/dev/fb0", O_RDWR);
uint8_t *fb = mmap(0, 640 * 480 * 4,
                   PROT_READ | PROT_WRITE,
                   MAP_SHARED, fb_fd, 0);
```

After this, all pixel writes are ordinary memory writes with no system calls or Avalon transactions required. Pixel (x, y) is located at byte offset $(y \times 640 + x) \times 4$.

5.3.2 Body Rendering

Bodies are rendered as filled white circles on a black background. Five circle sizes (radii 2, 4, 6, 8, and 10 pixels) are precomputed as bitmasks at startup. All masks are stored in a 21×21 grid sized for the largest radius; unused entries for smaller radii are explicitly zeroed by `memset` to avoid uninitialized values:

```
#define NUM_SIZES  5
#define MASK_DIM   21   /* 2 * max_radius + 1 */

static const int RADII[NUM_SIZES] = {2, 4, 6, 8, 10};
```

```
static uint8_t circle_mask[NUM_SIZES][MASK_DIM][MASK_DIM];

void precompute_circles(void) {
    memset(circle_mask, 0, sizeof(circle_mask));
    for (int s = 0; s < NUM_SIZES; s++) {
        int r = RADII[s];
        for (int dy = -r; dy <= r; dy++)
            for (int dx = -r; dx <= r; dx++)
                circle_mask[s][dy+10][dx+10] =
                    (dx*dx + dy*dy <= r*r) ? 1 : 0;
    }
}
```

The `draw_circle` function stamps the precomputed mask onto the framebuffer at a given center coordinate, iterating over the 21×21 bounding box and writing only pixels where the mask is set. A boundary check prevents out-of-bounds writes for bodies near the screen edge:

```
void draw_circle(uint8_t *fb, int cx, int cy,
                 int size, uint8_t color) {
    int r = RADII[size];
    for (int dy = -r; dy <= r; dy++) {
        for (int dx = -r; dx <= r; dx++) {
            if (!circle_mask[size][dy+10][dx+10])
                continue;
            int px = cx + dx;
            int py = cy + dy;
            if (px < 0 || px >= 640 ||
                py < 0 || py >= 480)
                continue;
            int offset = (py * 640 + px) * 4;
            fb[offset]   = color; /* R */
            fb[offset+1] = color; /* G */
            fb[offset+2] = color; /* B */
        }
    }
}
```

Each frame, the framebuffer is cleared with `memset` and all bodies are redrawn at their current positions. This correctly handles overlapping bodies without any additional bookkeeping:

```
void render_frame(uint8_t *fb, body_pos_t *pos,
                  int *sizes, int n_bodies) {
    memset(fb, 0, 640 * 480 * 4);
    for (int i = 0; i < n_bodies; i++)
        draw_circle(fb, pos[i].x, pos[i].y,
                    sizes[i], 0xFF);
}
```

A full `memset` of 1.2 MB takes approximately 0.3 ms on the ARM processor, which is negligible relative to the 16.7 ms frame budget at 60 Hz.

5.3.3 Coordinate Mapping

Body positions are stored in floating-point simulation coordinates and mapped linearly to integer screen pixel coordinates before rendering:

```
int sx = (int)((x - x_min) / (x_max - x_min) * 639.0f);
int sy = (int)((y - y_min) / (y_max - y_min) * 479.0f);
```

The world bounds `x_min`, `x_max`, `y_min`, `y_max` are fixed at initialization based on the initial body distribution, ensuring all bodies are visible within the 640×480 display area.

References

[1] Unknown, "N-body simulation on fpga (ieee paper)," *IEEE*, 2018. Available: https://ieeexplore.ieee.org/document/8445106.

[2] id Software, "Quake III arena source code." `https://github.com/id-Software/Quake-III-Arena`, 1999. Released under GPL v2, 2005.

[3] F. Stokes, "Fast inverse square root." `https://github.com/francisrstokes/githublog/blob/main/2024/5/29/fast-inverse-sqrt.md`, May 2024. Accessed: 2025.

[4] Wikipedia contributors, "Newton's method — Wikipedia, the free encyclopedia." `https://en.wikipedia.org/wiki/Newton%27s_method`, 2025. Accessed: 2025.

[5] Cornell ECE 5760, "Fpga-based n-body simulation final project." `https://people.ece.cornell.edu/land/courses/ece5760/FinalProjects/s2023/raf269_nkg37_tjw234/raf269_nkg37_tjw234/index.html`. Accessed: 2026-04-17.

[6] Wikipedia, "Leapfrog integration." `https://en.wikipedia.org/wiki/Leapfrog_integration`. Accessed: 2026-04-17.

[7] Intel (Altera), "Cyclone v device handbook volume 1." `https://docs.altera.com/r/docs/683375/current/cyclone-v-device-handbook-volume-1-device-interfaces-and-integration/resources`. Accessed: 2026-04-17.