Hardware Accelerated N-Body Simulations

CSEE 4840 - Spring 2025

Adib Khondoker (aak2250), Moises Mata (mm6155), Kristian Nikolov (kdn2117),

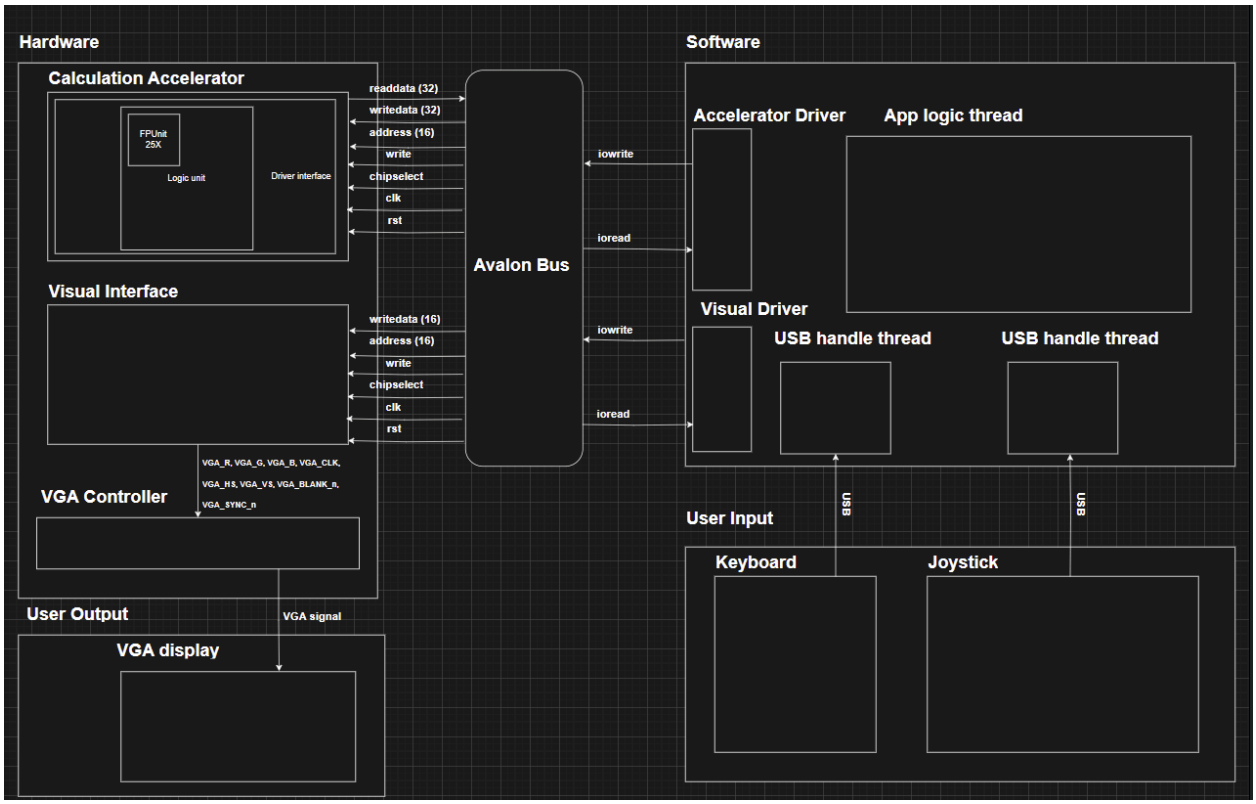
Isaac Trost (wit2102) , Robert Pendergrast (rlp2153)

Contents:

Abstract:

The purpose of this project is to develop a hardware-accelerated and interactive N-body simulation. Given a set number of bodies (preemptively 8), a hardware-based implementation of the leapfrog integration algorithm will iteratively compute a set of positions corresponding to each value. These positions will be sent to the VGA display for visualization. The user will be able to interface with the simulation through the use of a joystick, which will be used to cycle forwards and backwards through the simulation. An in-depth outline of this project is provided in the following sections.

Block Diagram:



Algorithms:

In order to produce accurate N-Body simulations, it is essential that the algorithms used to calculate and update the body positions are well defined. The selected process for this project makes use of a two step process, outlined below:

N-Body Force Calculation:

The motions of the bodies are governed by Newton's Law of Universal Gravitation (Equation 1), which relates the force vector produced on one object by another to their respective masses (m_1 and m_2) and the distance (r) between them:

$$F = G\frac{m_1 m_2}{r^2} \qquad \text{Equation 1}$$

Where G is the gravitational constant $6.67428 \times 10^{-11}\ m^3 kg^{-1} s^{-2}$.

For the purposes of the simulation, the net force action on *each* body must be calculated as a sum of all the individual gravitational forces acting upon it. Then, the acceleration each body is subject to can be determined by dividing each net force by the mass of its respective object, as per Newton's Second Law (Equation 2):

$$A = \frac{\Sigma F}{M} \qquad \text{Equation 2}$$

This process lends itself to an intuitive algorithm by which the accelerations of all the bodies can be obtained:

```python
def get_acceleration(R,G,M,N): #position, G, mass array, # bodies
    acceleration = np.zeros((N,2))
    F = np.zeros((N,2))
    for i in range(N):
        for j in range(N):
            if i != j:
                F[i] +=  1 * (G * M[i] * M[j] * (R[i] - R[j]) /
(np.linalg.norm((R[i]-R[j])))**3)

        acceleration[i] = -1 * (F[i] / M[i])

    return acceleration
```

Leapfrog Integration Algorithm:
In order to update the position of each body, the leapfrog integration algorithm will be used. The leapfrog integration algorithm is ideal for this project because it is both efficient and precise, making use of a three step integration process outlined below:

1. Update the velocity for half a time step using previous acceleration:
$$v_{i+1/2} = v_i + \frac{1}{2}a_i\Delta t \qquad \text{Equation 3}$$
2. Update the position with the new velocity:
$$x_{i+1} = x_i + v_{i+1/2}\Delta t \qquad \text{Equation 4}$$
3. Update the velocity for the second time step with updated acceleration:
$$v_{i+1} = v_{i+1/2} + \frac{1}{2}a_{i+1}\Delta t \qquad \text{Equation 5}$$

This algorithm is written below:

```python
while t0 < T:
        V += A * dt/2 # Calculate v i+1/2
        R += V * dt # Calculate x i+1
        A = get_acceleration(R,G,M,N) # Calculate a i+1
        V += A * dt/2 # Calculate v i+1
        t0 += dt
```

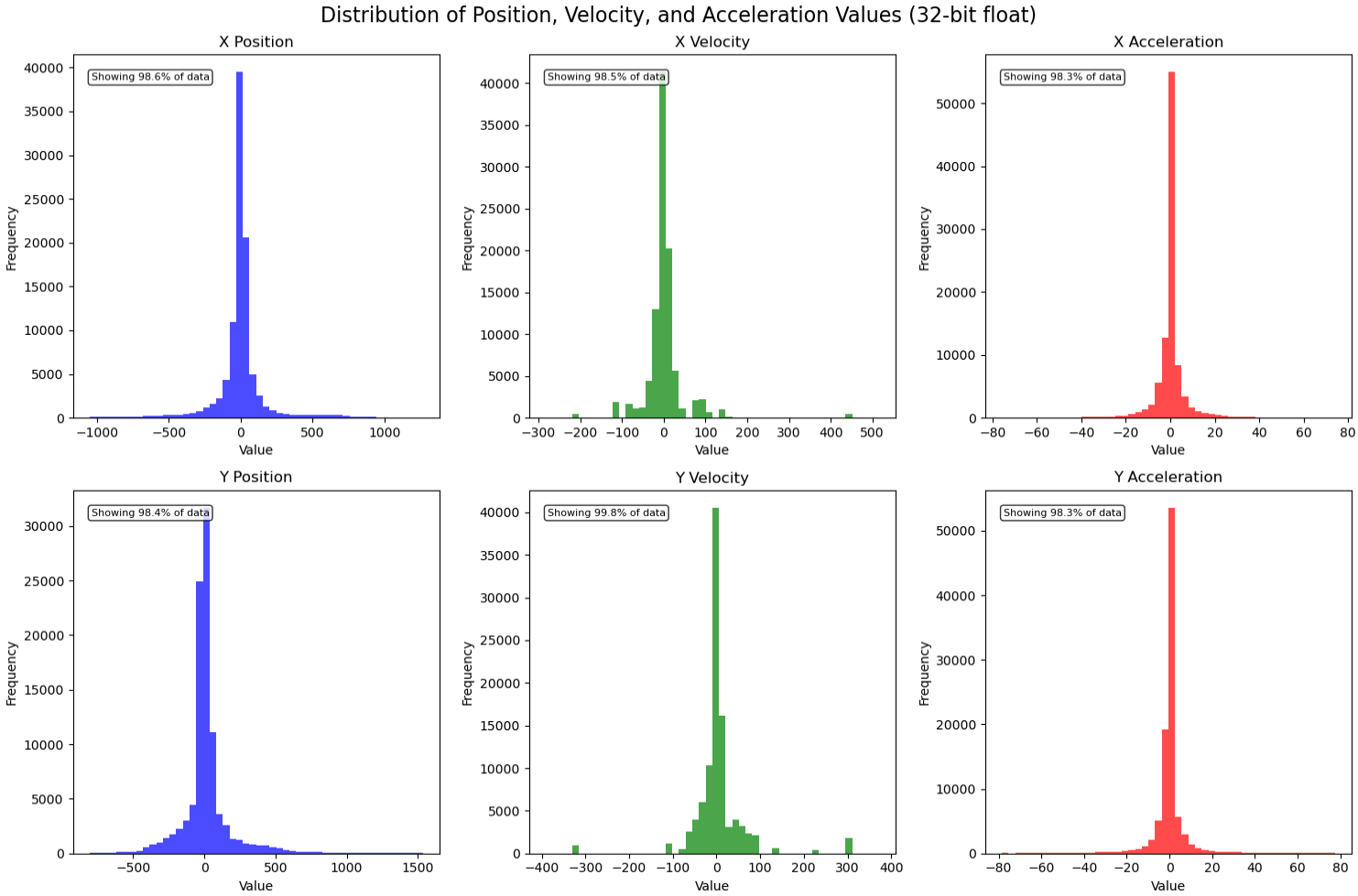
The leapfrog integration step makes use of the N-Body in order to determine the updated acceleration in step 2.

As illustrated by the block diagram, both the acceleration vector calculations and the actual leapfrog integration will be carried out by the hardware. The software will be solely responsible for handling user input and driving the VGA display.

Hardware will accelerate the standard N-body calculation by parallelizing the above algorithm for each body. Leapfrog numerical integration is uniquely suited to hardware parallelization because of its two half step velocity update procedure. Though a serial version of the algorithm would be hindered by a second velocity calculation, our hardware will be able to calculate this velocity at the same time as the acceleration calculation.

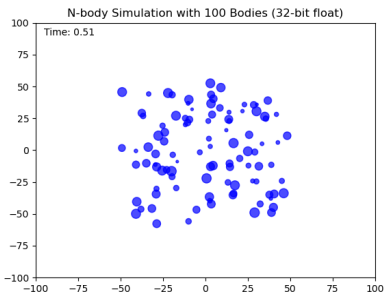
We will be using 32 bit floating point numbers as our main numeric type, this choice balances precision, with the constrained resources of our platform, and is nicely supported by the built in multipliers. Position, Velocity, and Acceleration, in both x and y directions will be represented by

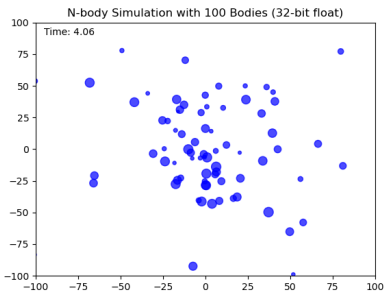
this data type. The domains for all of these values are shown below (generated from a 100 body simulation in python), 32 bit floating points are capable of representing all of these domains.

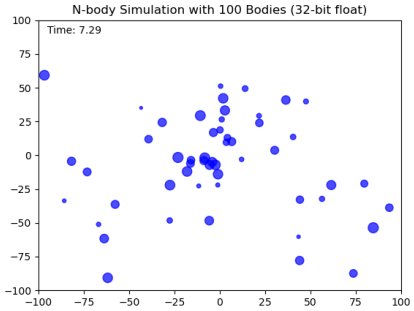


User Interface:

The application will be controlled by the user in software. Upon initialization, the user will be able to indicate: the number of bodies, the number of timesteps to simulate, and the length of each timestep. Software will handle random initialization of the initial position and velocity of each body and signal to begin the simulation. The user will then be able to use the *Saitek ST90* Joystick to evolve the simulation through time, both forwards and backwards. This will be done by saving all positions returned by the hardware accelerator in software.







Resource Budgets:

Our two main constraints will be memory and the multipliers needed for the floating point hardware components. The resources required by a single IEEE 754 float arithmetic unit is as follows [1]:

- Resource consumption in a typical system:
  - Approximately 2500 4-input LEs
  - Nine 9-bit multipliers
  - Three M9K memories or larger

On the 5CSEA5 Cyclone V SE SoC there are 85000 LEs, 261 9 bit multipliers, and 397 M10K memory blocks available. These numbers are seen below:

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

Product Line		Cyclone V SE SoCs[1]		
		5CSEA2	5CSEA4	5CSEA5
Resources	LEs (K)	25	40	85
	ALMs	9,430	15,880	32,070
	Registers	37,736	60,376	128,300
	M10K memory blocks	140	270	397
	M10K memory (Kb)	1,400	2,700	3,970
	MLAB memory (Kb)	138	231	480
	Variable-precision DSP blocks	36	84	87
	18 x 18 multipliers	72	168	174

Thus the limiting factor for the amount of float arithmetic units is the number of multipliers, which give us a maximum of 261/9 or 29 u
\nits. However, this would use up all of the DSP blocks, so we will only use 25. This will use 75 M10k memory blocks, leaving us with 322 kilobits left.

In regards to the memory required, the input size will be n * 5 * 32 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 32 bits is due to the standard float size. 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 as they will no longer be needed. An additional n * 32 bits will be needed during calculation to store temporary values. Finally, output memory is n * 4 * 32bits as well as the same n bit vector map. This is nearly identical to the input with the exception of the number of values passed in, which has decreased from 5 to 4, due to the fact that mass is not needed to be passed out.

Thus we can roughly estimate (n*5*32)+n+(n*32)+(n*4*32)+(n) or 322*n bits required. Given that we have 322 kilobits left of memory, and if we run simulations with 512 bodies we will only require around 165 kilobits, we do not appear to have any issues with the memory budget.

Hardware-Software Interface:

The project requires two hardware-software interfaces, one for the accelerator module and another for the VGA display. These two interfaces are described in detail below:

Accelerator Interface:
The accelerator interface must allow for the hardware to iteratively compute updated x and y positions of the *n* bodies. The x and y coordinates for each body, their velocities in x and y, and their masses, will be written into memory.

There will be a driver interface module that will take in the input from the avalon bus. It will map address 0 to GO, 1 to the n_timesteps and timestep size (each 16 bits). The next 16 addresses (each is 32 bits) will be mapped to a bitmap showing if that body is active (for a max of 512 bodies.) After that, the addresses will map to each of the 5 inputs for the potentially 512 bodies. Everything but the 5 inputs for each of the 512 bodies will be placed in registers to allow more data access parallelism in the accelerator itself. Those 5 inputs for each of the 512 bodies will be placed in memory.

There will be a second region of hardware flash memory that will be used to output values to software. The accelerator module will store a timestep it is outputting for, an updated bitmap showing which of the 512 bodies are active (accounting for the possibility that it had to deactivate some if there was a numerical error like an overflow,) and finally, the x and y positions of each of the 512 bodies. The software can pass in these addresses to read those values once the module indicates it is done via the done register which the software will poll.

The interface for this driver interface will look like this:

```
C/C++
module driver_interface(
            //Input Registers
            input logic clk,
            input logic  reset,
            input logic [31:0] writedata,
            input logic [15:0] address,
            input logic write,
            input logic chipselect,
            //Output
            output logic [31:0] writedata
);
```

Memory layout (software perspective, 0 is base pointer)

Byte address	0-3	4-5	6-7	8-23	24-28	…	2579-2583
Thing at location	GO	n_timesteps	timestep_len	Activity bitmap	Params for body 1	…	Params for body 512

Byte address	2584	2585-2600	2601-2602	…	3623-3624
Thing at location	Done	Output activity bitmap	X and Y for body 1	…	X and Y for body 512

The hardware driver will take in these addresses, and input and output values from registers and/or memory depending on the nature of the object stored at that virtual address (generally, body specific inputs and outputs are stored in flash memory, and more general values that could be used by more than one body are stored in registers for more effective parallelism.)

The interface for the actual accelerator, which is defined as a submodule of the driver interface:

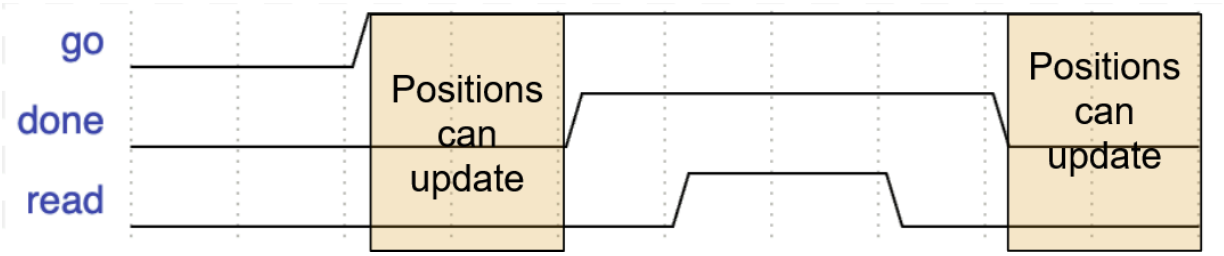
```
C/C++
module n_bodies(
            //Input Registers
            input logic clk,
            input logic go,
            input logic read,
            input logic [511:0] bitmap
            input logic [15:0] n_timesteps,
            input logic [15:0] timestep
            //Output
            output logic done);
```

The registers n_timesteps and timestep are both 16 bit unsigned registers, representing the number of timesteps for the simulation and the size of each timestep respectively. These registers were selected to be 16 bits in order to ensure that simulations of adequate lengths could be run.

The clock (clk), go, and read input signals are all single bit values, and therefore can be abstracted to being set to either high or low. For a given cycle, if go is set to true, then the module will begin iteratively computing updated *n* body positions and velocities for each timestep. After the updated positions and velocities are computed for a single timestep, done will be set to high. The software will be constantly polling for the done signal, upon which it will read the updated positions from memory, setting input logic to 1. Once this finishes, read will be

again set to 0, following which output signal done will also be lowered, signifying that that the next timestep update can be computed.

This means that a positional update can only occur when Go = 1, Done = 0, and Read = 0. A timing diagram* for this process is provided below for increased clarity:



*Note that the horizontal spacing is not to scale.

Finally, the bitmap will show which of the bodies are active at the start.

Display Interface

The hardware interface for the VGA display is largely based off of the VGA_ball module used in lab three. However because of the fact that all of the body positional information will be stored in memory, the logic for handling the positional information for each of the balls will be different.

```
C/C++
module vga_n_bodies(
            //Input Registers
            input logic        clk,
            input logic  reset,
            input logic [15:0] writedata,
            input logic [15:0] address,
            input logic write,
            input logic chipselect,

            //Output Registers
            output logic [7:0] VGA_R, VGA_G, VGA_B,
            output logic       VGA_HS, VGA_VS, VGA_CLK,
            output logic       VGA_BLANK_n, VGA_SYNC_n);
```

The input logic clk and reset are used to control the VGA display's output, with the display being updated on each clock edge, and reset being used to restart the display. The three 8-bit output registers VGA_R, VGA_G, and VGA_B reference the R,G, and B color channels for a given

pixel, respectively. The output signals VGA_HS and VGA_VS monitor the horizontal and vertical syncs, respectively.

References

1. Intel Corporation. “Nios II Custom Instruction User Guide”
2. “Leapfrog Integration” *Wikipedia, The Free Encyclopedia*, Wikimedia Foundation, 15 April 2025, https://en.wikipedia.org/wiki/Leapfrog_integration.