Kyung Hwan ‘David’ Lee

EECE 4638: Advanced Digital Design

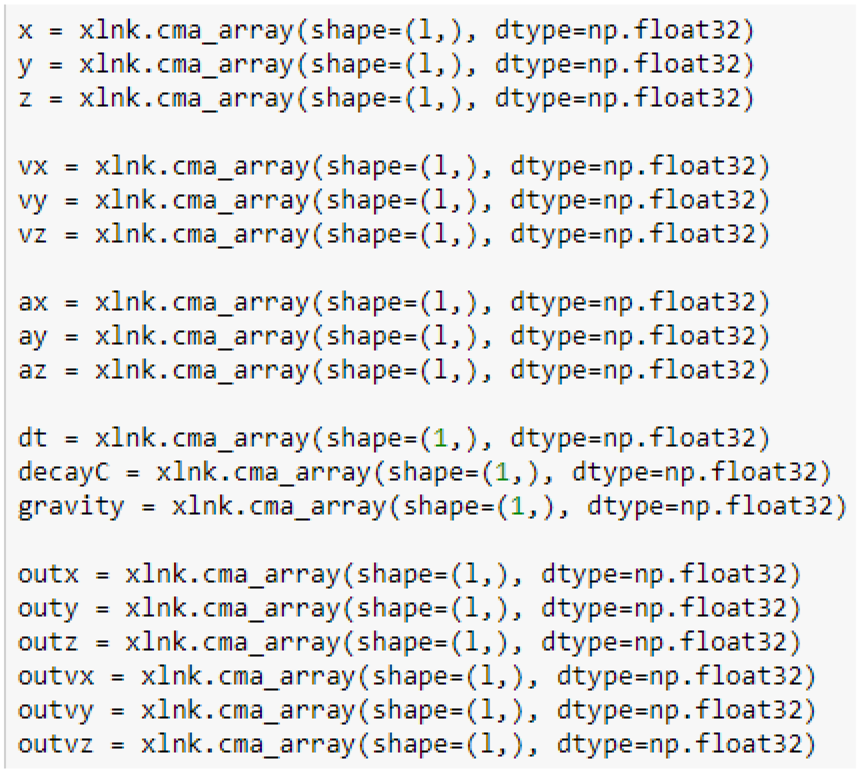
Project Report: *Hardware Acceleration of Physics Simulation*

This project implements a 3D particle simulation model onboard the PYNQ board. The code is original but takes *Jake Vanderplas' Matplotlib Animation Tutorial* [1] as a reference.

The submitted physics simulation python code can be broken up into three sections: Initialization, Update, and Animate. The hardware accelerated particle simulation also utilizes the same three sections. Primarily, the Update section utilizes the FPGA to hardware accelerate the floating-point calculations needed to enable the physics simulation. However, the Initialization section must be modified to allow the Programmable Logic (PL) to compute floating-point data types. The Animate section of both Python only and hardware accelerated simulator are the same. This section utilizes the Matplotlib *animate.FuncAnimation* included in its library.

Common to both Python only and Hardware Accelerated Simulator, the Initialization stage sets the physical properties of the simulation. This includes: the graph limits, the force of gravity, timestep per calculation, particle collision range (size of particle), and a parameterized decay constant which is used as a substitute for friction. The decay constant is a parameterized decay of acceleration and velocity which occurs due to friction or collision of particles. The decay constant is used to simplify calculations within the Update stage. The Initialization stage will also populate 9 arrays: x[i], y[i], z[i], v[x], vy[i], vz[i], ax[i], ay[i], az[i]. These arrays are the initial x, y, z positions, initial velocity in the x, y, z direction, and initial acceleration in the x, y, z direction. The total count of particles, *i,* can be specified in the Initialization section and these initial arrays are randomly generated. In the Python only simulator, these arrays are numpy floating point arrays.

The Hardware Accelerated Simulator requires these arrays to be initialized differently to be usable in the PL. This is because PYNQ only supports np.uint32 and np.uin64 register writing. Because we must use floating-point data types in our simulation, we must use the m_axi4 protocol and fetch data from DDR. To do this, the floating point numbers must be initialized using a nd.array. Below is how these arrays are initialized:



```
x = xlnk.cma_array(shape=(1,), dtype=np.float32)
y = xlnk.cma_array(shape=(1,), dtype=np.float32)
z = xlnk.cma_array(shape=(1,), dtype=np.float32)

vx = xlnk.cma_array(shape=(1,), dtype=np.float32)
vy = xlnk.cma_array(shape=(1,), dtype=np.float32)
vz = xlnk.cma_array(shape=(1,), dtype=np.float32)

ax = xlnk.cma_array(shape=(1,), dtype=np.float32)
ay = xlnk.cma_array(shape=(1,), dtype=np.float32)
az = xlnk.cma_array(shape=(1,), dtype=np.float32)

dt = xlnk.cma_array(shape=(1,), dtype=np.float32)
decayC = xlnk.cma_array(shape=(1,), dtype=np.float32)
gravity = xlnk.cma_array(shape=(1,), dtype=np.float32)

outx = xlnk.cma_array(shape=(1,), dtype=np.float32)
outy = xlnk.cma_array(shape=(1,), dtype=np.float32)
outz = xlnk.cma_array(shape=(1,), dtype=np.float32)
outvx = xlnk.cma_array(shape=(1,), dtype=np.float32)
outvy = xlnk.cma_array(shape=(1,), dtype=np.float32)
outvz = xlnk.cma_array(shape=(1,), dtype=np.float32)
```

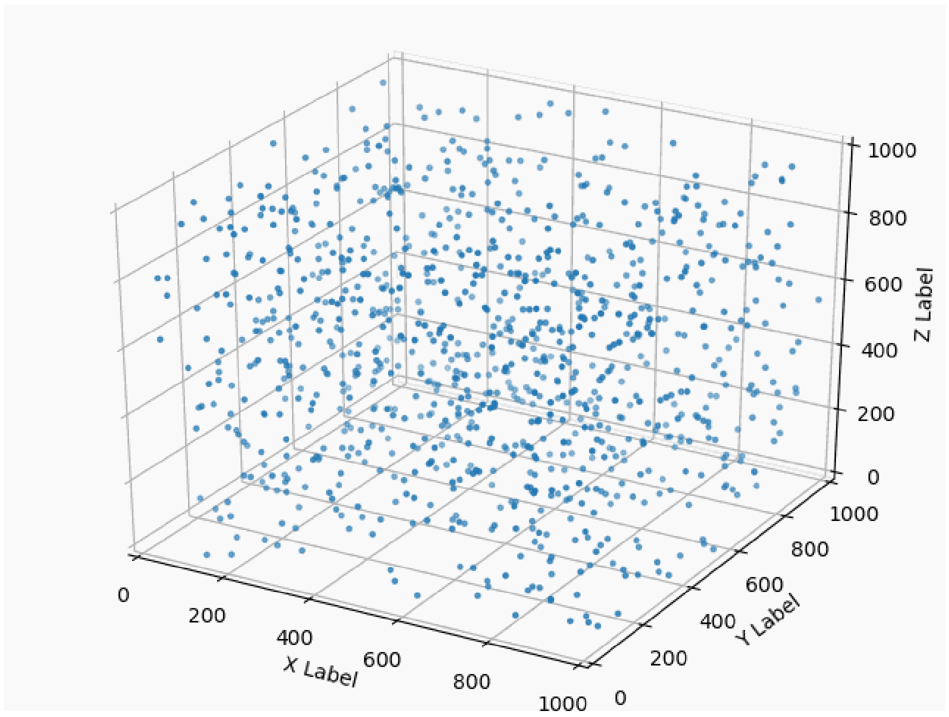
The Update stage includes all calculations needed to determine the next x, y, and z positions and velocities of each particle. This includes calculating the velocity from acceleration and position from the velocity at each time step interval specified during initialization. The update function will also calculate the next time step to check for any collisions with the graph limits or with other particles. The Update stage will adjust velocity, acceleration, and position of each particle experiencing collision. Then, the position, velocity, and acceleration of each particle in small values will be quantized to zero. This is done to ensure the animate function looks smooth and eliminates tiny movements which are unnecessary at this scale. The following figures are from the Update stage of the Python only simulator. However, the same implementation is used in the Hardware Accelerated Simulation.

```python
#Quantize small movements so that balls do not float outside of box
for i in range(count):
    if vz[i] < 1 and z[i] < 1:
        az[i] = 0
    if vy[i] < 1 and y[i] < 1:
        ay[i] = 0.0
    if vx[i] < 1 and x[i] < 1:
        ax[i] = 0

for i in range(count):
    vx[i] += (ax[i] * dt)
    vy[i] += (ay[i] * dt)
    vz[i] += (az[i] * dt)

for i in range(count):
    x[i] += (vx[i] * dt)
    y[i] += (vy[i] * dt)
    z[i] += (vz[i] * dt)
```

```python
#Check Boundary Collision Condition
for i in range(count):
    if xnext[i] < 0:
        vx[i] *= -1 * np.exp(-decayC)
    if xnext[i] > graphLimits:
        vx[i] *= -1 * np.exp(-decayC)

    if ynext[i] < 0:
        vy[i] *= -1 * np.exp(-decayC)
    if ynext[i] > graphLimits:
        vy[i] *= -1 * np.exp(-decayC)

    if znext[i] < 0:
        vz[i] *= -1 * np.exp(-decayC)
    if znext[i] > graphLimits:
        vz[i] *= -1 * np.exp(-decayC)
```

```python
#Collision with other particles
for i in range(count):
    for j in range (i+1, count, 1):
        if x[j] - d < x[i] < x[j] + d and y[j] - d < y[i] < y[j] + d and z[j] - d < z[i] < z[j] + d:
            ax[i] *= -1
            ay[i] *= -1
            az[i] *= -1
            vx[i] *= -1
            vy[i] *= -1
            vz[i] *= -1

            ax[j] *= -1
            ay[j] *= -1
            az[j] *= -1
            vx[j] *= -1
            vy[j] *= -1
            vz[j] *= -1
```

The Animate stage includes all steps required to produce a visual representation of the physics simulator. This is done with the python *Matplotlib* library's *animate.FuncAnimation* function. The results of this is seen below:



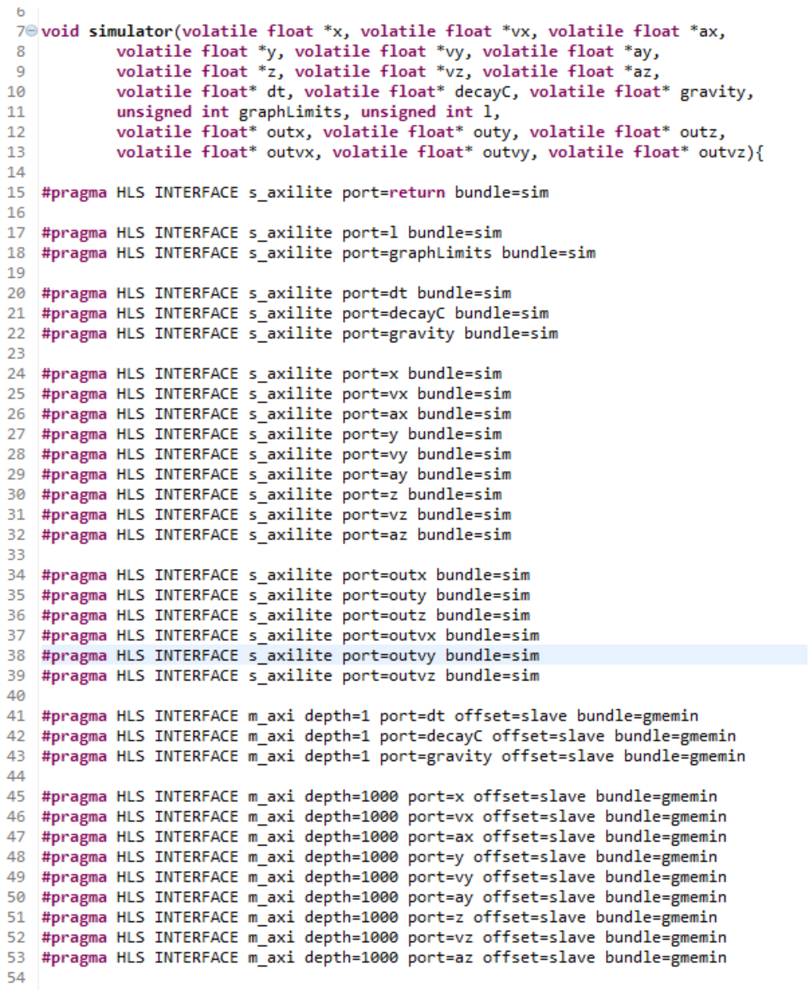
Some extra steps were taken to display this image on board the PYNQ board [2]. They are described in the included Jupyter notebook.

Additional steps were taken on the hardware side to accommodate floating-point data types. The following figure shows are the PS must communicate with the PL.

```
sim.write(0x58, dt.physical_address)
sim.write(0x60, decayC.physical_address)
sim.write(0x68, gravity.physical_address)
sim.write(0x70, graphLimits)
sim.write(0x78, l)

sim.write(0x10, x.physical_address)
sim.write(0x18, vx.physical_address)
sim.write(0x20, ax.physical_address)
sim.write(0x28, y.physical_address)
sim.write(0x30, vy.physical_address)
sim.write(0x38, ay.physical_address)
sim.write(0x40, z.physical_address)
sim.write(0x48, vz.physical_address)
sim.write(0x50, az.physical_address)

sim.write(0x80, outx.physical_address)
sim.write(0x88, outy.physical_address)
sim.write(0x90, outz.physical_address)
sim.write(0x98, outvx.physical_address)
sim.write(0xa8, outvy.physical_address)
sim.write(0xa8, outvz.physical_address)
```

```
sim.write(0x0, 1)
```

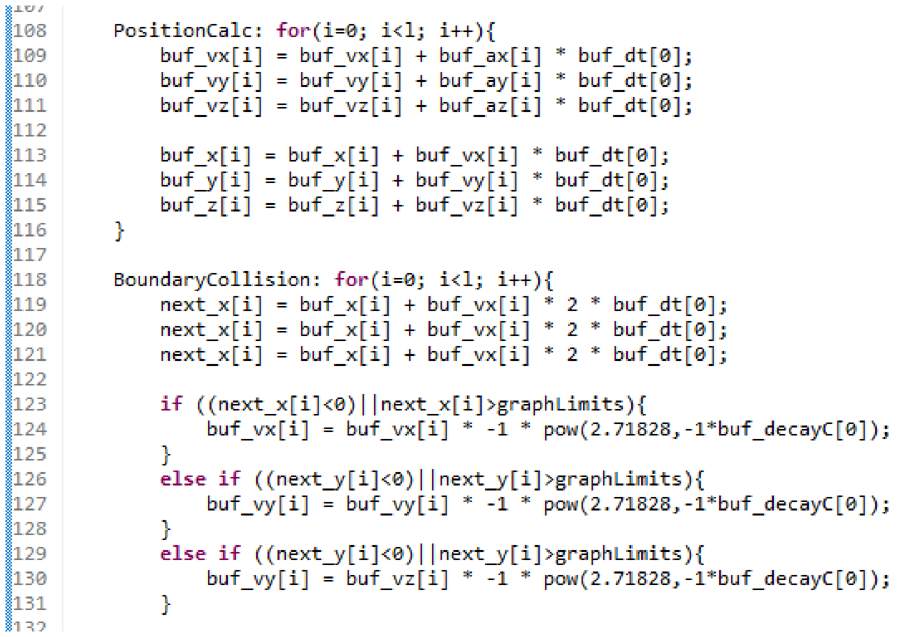
The physical address of the physical constraints and position, velocity, and acceleration of each particle are passed to the PL. The hardware must then buffer this data in order to calculate the next time step. The figures below show us how this is done.



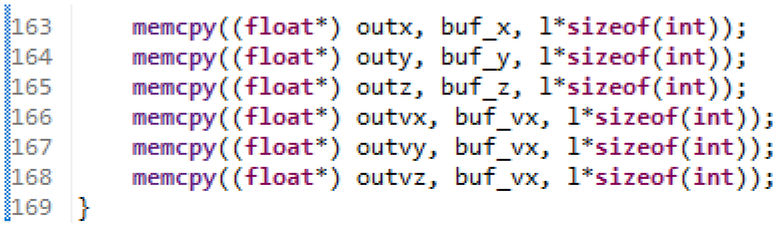
```

void simulator(volatile float *x, volatile float *vx, volatile float *ax,
        volatile float *y, volatile float *vy, volatile float *ay,
        volatile float *z, volatile float *vz, volatile float *az,
        volatile float* dt, volatile float* decayC, volatile float* gravity,
        unsigned int graphLimits, unsigned int l,
        volatile float* outx, volatile float* outy, volatile float* outz,
        volatile float* outvx, volatile float* outvy, volatile float* outvz){

#pragma HLS INTERFACE s_axilite port=return bundle=sim

#pragma HLS INTERFACE s_axilite port=l bundle=sim
#pragma HLS INTERFACE s_axilite port=graphLimits bundle=sim

#pragma HLS INTERFACE s_axilite port=dt bundle=sim
#pragma HLS INTERFACE s_axilite port=decayC bundle=sim
#pragma HLS INTERFACE s_axilite port=gravity bundle=sim

#pragma HLS INTERFACE s_axilite port=x bundle=sim
#pragma HLS INTERFACE s_axilite port=vx bundle=sim
#pragma HLS INTERFACE s_axilite port=ax bundle=sim
#pragma HLS INTERFACE s_axilite port=y bundle=sim
#pragma HLS INTERFACE s_axilite port=vy bundle=sim
#pragma HLS INTERFACE s_axilite port=ay bundle=sim
#pragma HLS INTERFACE s_axilite port=z bundle=sim
#pragma HLS INTERFACE s_axilite port=vz bundle=sim
#pragma HLS INTERFACE s_axilite port=az bundle=sim

#pragma HLS INTERFACE s_axilite port=outx bundle=sim
#pragma HLS INTERFACE s_axilite port=outy bundle=sim
#pragma HLS INTERFACE s_axilite port=outz bundle=sim
#pragma HLS INTERFACE s_axilite port=outvx bundle=sim
#pragma HLS INTERFACE s_axilite port=outvy bundle=sim
#pragma HLS INTERFACE s_axilite port=outvz bundle=sim

#pragma HLS INTERFACE m_axi depth=1 port=dt offset=slave bundle=gmemin
#pragma HLS INTERFACE m_axi depth=1 port=decayC offset=slave bundle=gmemin
#pragma HLS INTERFACE m_axi depth=1 port=gravity offset=slave bundle=gmemin

#pragma HLS INTERFACE m_axi depth=1000 port=x offset=slave bundle=gmemin
#pragma HLS INTERFACE m_axi depth=1000 port=vx offset=slave bundle=gmemin
#pragma HLS INTERFACE m_axi depth=1000 port=ax offset=slave bundle=gmemin
#pragma HLS INTERFACE m_axi depth=1000 port=y offset=slave bundle=gmemin
#pragma HLS INTERFACE m_axi depth=1000 port=vy offset=slave bundle=gmemin
#pragma HLS INTERFACE m_axi depth=1000 port=ay offset=slave bundle=gmemin
#pragma HLS INTERFACE m_axi depth=1000 port=z offset=slave bundle=gmemin
#pragma HLS INTERFACE m_axi depth=1000 port=vz offset=slave bundle=gmemin
#pragma HLS INTERFACE m_axi depth=1000 port=az offset=slave bundle=gmemin

```

```
float buf_x[count], buf_vx[count], buf_ax[count];
float buf_y[count], buf_vy[count], buf_ay[count];
float buf_z[count], buf_vz[count], buf_az[count];

float next_x[count], next_y[count], next_z[count];

float buf_dt[1], buf_decayC[1], buf_gravity[1];

float d = 5;


memcpy(buf_dt, (const float*) dt, sizeof(int));
memcpy(buf_decayC, (const float*) decayC, sizeof(int));
memcpy(buf_gravity, (const float*) gravity, sizeof(int));

memcpy(buf_x, (const float*) x, l*sizeof(int));
memcpy(buf_vx, (const float*) vx, l*sizeof(int));
memcpy(buf_ax, (const float*) ax, l*sizeof(int));
memcpy(buf_y, (const float*) y, l*sizeof(int));
memcpy(buf_vy, (const float*) vy, l*sizeof(int));
memcpy(buf_ay, (const float*) ay, l*sizeof(int));
memcpy(buf_z, (const float*) z, l*sizeof(int));
memcpy(buf_vz, (const float*) vz, l*sizeof(int));
memcpy(buf_az, (const float*) az, l*sizeof(int));
```

After the calculations are finished, the data must be pushed to DDR using the following method.

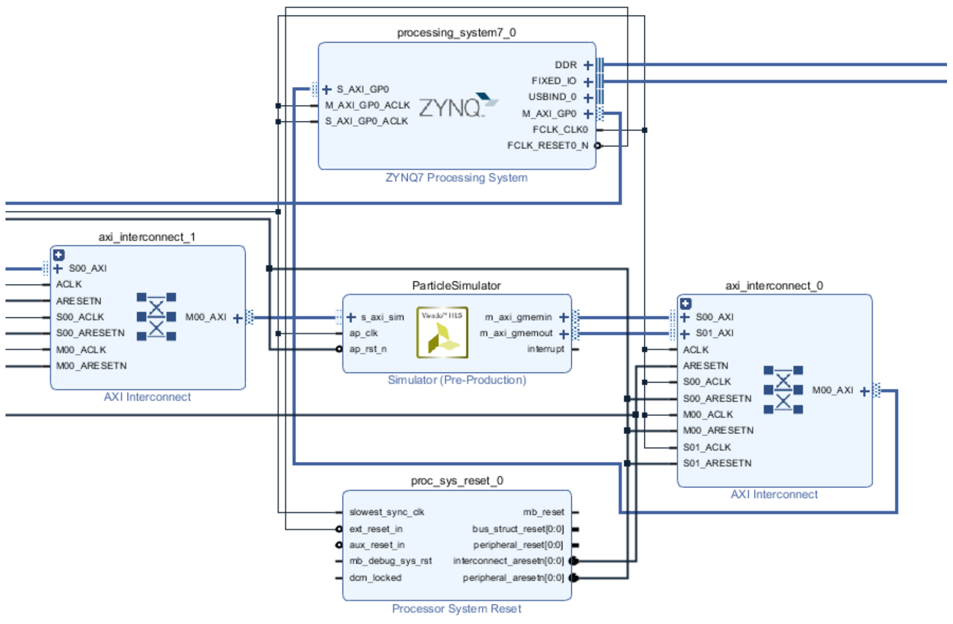


```
    PositionCalc: for(i=0; i<l; i++){
        buf_vx[i] = buf_vx[i] + buf_ax[i] * buf_dt[0];
        buf_vy[i] = buf_vy[i] + buf_ay[i] * buf_dt[0];
        buf_vz[i] = buf_vz[i] + buf_az[i] * buf_dt[0];

        buf_x[i] = buf_x[i] + buf_vx[i] * buf_dt[0];
        buf_y[i] = buf_y[i] + buf_vy[i] * buf_dt[0];
        buf_z[i] = buf_z[i] + buf_vz[i] * buf_dt[0];
    }

    BoundaryCollision: for(i=0; i<l; i++){
        next_x[i] = buf_x[i] + buf_vx[i] * 2 * buf_dt[0];
        next_x[i] = buf_x[i] + buf_vx[i] * 2 * buf_dt[0];
        next_x[i] = buf_x[i] + buf_vx[i] * 2 * buf_dt[0];

        if ((next_x[i]<0)||next_x[i]>graphLimits){
            buf_vx[i] = buf_vx[i] * -1 * pow(2.71828,-1*buf_decayC[0]);
        }
        else if ((next_y[i]<0)||next_y[i]>graphLimits){
            buf_vy[i] = buf_vy[i] * -1 * pow(2.71828,-1*buf_decayC[0]);
        }
        else if ((next_y[i]<0)||next_y[i]>graphLimits){
            buf_vy[i] = buf_vz[i] * -1 * pow(2.71828,-1*buf_decayC[0]);
        }
```

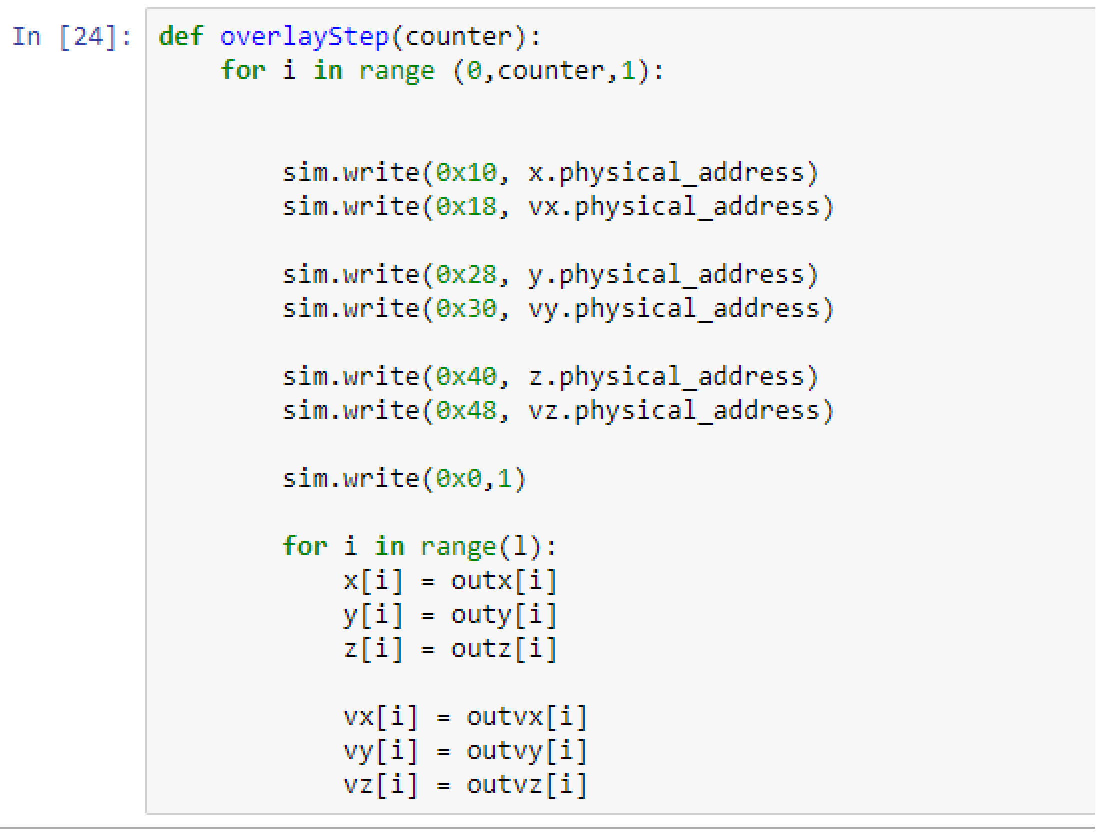


```
    memcpy((float*) outx, buf_x, l*sizeof(int));
    memcpy((float*) outy, buf_y, l*sizeof(int));
    memcpy((float*) outz, buf_z, l*sizeof(int));
    memcpy((float*) outvx, buf_vx, l*sizeof(int));
    memcpy((float*) outvy, buf_vx, l*sizeof(int));
    memcpy((float*) outvz, buf_vx, l*sizeof(int));
}
```

The hardware block diagram is shown below.



To correctly evaluate the hardware acceleration, we must evaluate the Update stage separate from the Animate and Initialization stage. This is because the Animate stage includes graphical processing which is not being evaluated in this project. Two new functions are defined to evaluate only the Update stages in a separate Jupyter notebook. These functions execute the loop *j* number of times. Below is the Hardware Accelerated test function.



```
In [24]: def overlayStep(counter):
             for i in range (0,counter,1):


                 sim.write(0x10, x.physical_address)
                 sim.write(0x18, vx.physical_address)

                 sim.write(0x28, y.physical_address)
                 sim.write(0x30, vy.physical_address)

                 sim.write(0x40, z.physical_address)
                 sim.write(0x48, vz.physical_address)

                 sim.write(0x0,1)

                 for i in range(l):
                     x[i] = outx[i]
                     y[i] = outy[i]
                     z[i] = outz[i]

                     vx[i] = outvx[i]
                     vy[i] = outvy[i]
                     vz[i] = outvz[i]
```

The hardware was also optimized using HLS pragma directives and the loops were manually merged. A detailed report of these optimizations, their FPGA usage, and their resulting speed are detailed in sheet 1 of the included excel sheet. The test function had a particle count of 5 and the loop ran 1 time.

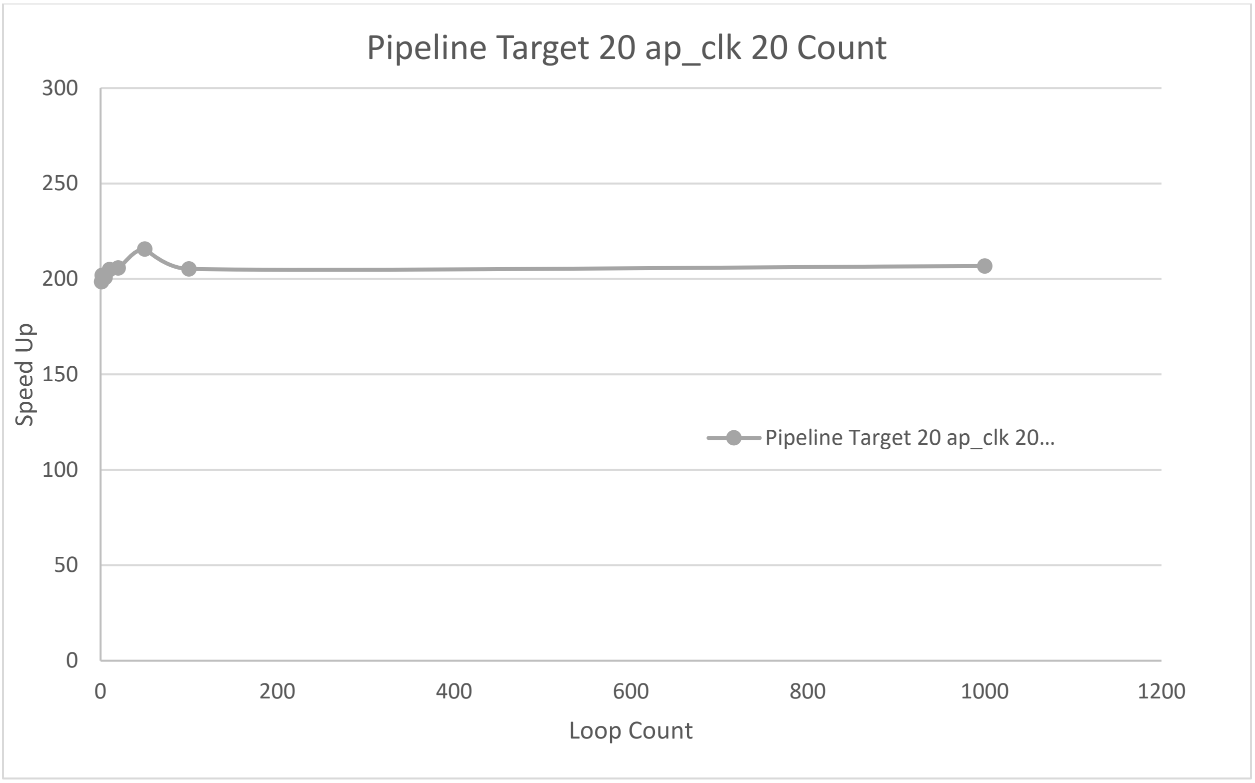
The following observations were made. In the original code, the for loops checked *l* elements in the array. This *l* is the total number of particles in the simulation and is passed from the PS. Because of the dynamic range of the loops, the Loop Unroll Pragma is suspected to be functioning incorrectly (This is determined by the HLS Synthesis Warning Message). The *l* variable was replaced with a constant 1000 value. This value is chosen because the m_axi4 bus was configured to have a depth of 1000. With this done, it is suspected that the Loop Unroll Pragma will unroll correctly (HLS Synthesis Warning Message does not appear). It is observed that with this done, the area of usage does not increase significantly.

It can be observed that unrolling and pipelining increases the used footprint of the FPGA without much change in the speed. Manually merging the for loops resulted in the same or slightly better speed with substantially less area used within the FPGA. Therefore, the manually merged loops were utilized. After trying a few optimizations, the final hardware design utilized a manually merged loop which has been pipelined, unrolled by a factor of 2, and had a target clock of 20 ns (ap_clk is measured at 18.671 in HLS). This resulted in the best performance and usage area. This is documented in the included excel file.

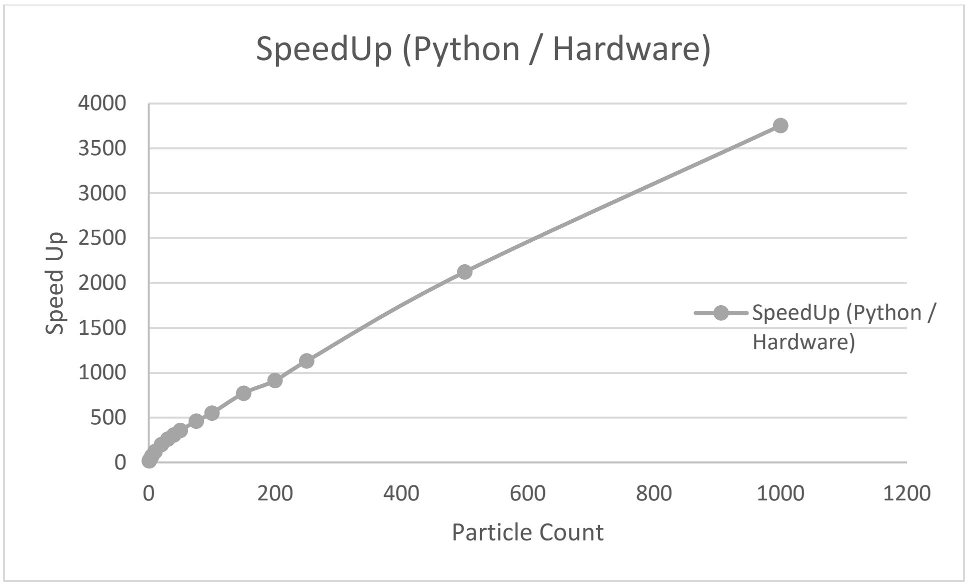
We use the test functions to observe changes in performance due to changing particle counts and loop numbers. All these tests are recorded in sheet 2 of the included excel file.
Hardware speedup is defined by the following:

$$SpeedUp = \frac{PythonOnlyTime}{HardwareAccTime}$$

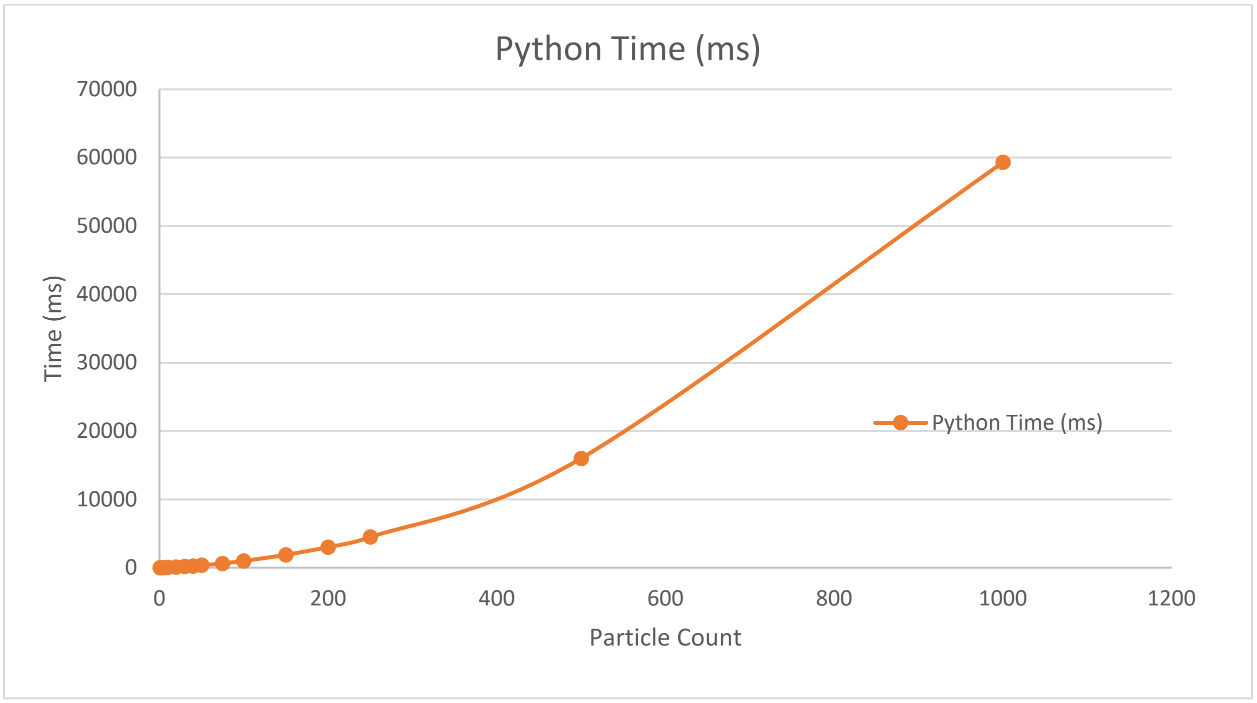
It is observed that increasing the loop count does not affect the hardware speedup. Below is a graph detailing the hardware speedup changes due to loop count.

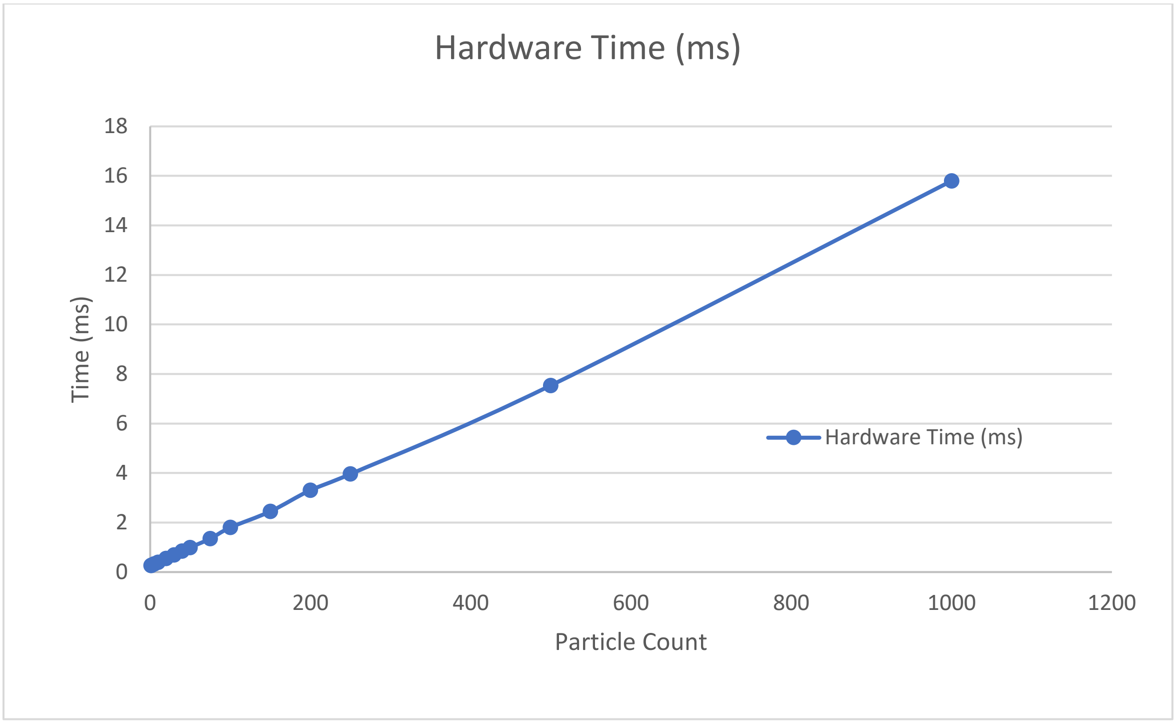


However, there is a significant change of hardware speedup due to changing particle counts. These tests complete only one loop. Below is a graph detailing the hardware speedup due to changing particle counts.



Below are graphs detailing the time it takes to complete one loop at different particle counts for Python only and Hardware Accelerated Update functions.





It is observed that as the particle count increases, the hardware speedup increases. It is suspected that the Python only functions take exponentially longer to complete one loop as the particle count increases.

The speed the Python only function takes to complete one loop exhibits why hardware acceleration can be very useful. In this case, the animation must complete one loop at least every 30 Hz (1/30 seconds) to produce a smooth animation (30 Hz = ~ 33.3 ms per loop). However, at increasing particle counts, the python only function can take approximately 1 minute. With the python only function the 30 Hz per loop constraint can only be achieved at a particle count less than 10. The Hardware Accelerated Simulation loop can achieve greater than 1000 particles. This is detailed in the included excel sheet.

During this project, it was surprising to see that passing floating point data types through the AXI Bus was so difficult. Ideally, the AXI streaming protocol would be more efficient as it can continuously stream position, velocity, and acceleration. This would eliminate writing to memory, reading from memory, and buffering within the programmable logic which must be used in the above hardware acceleration. It was also interesting to see that the above update function was taxing even on a desktop computer (running an i7). At particle counts near and above ~1000, it was utilizing ~70% of CPU usage to generate an animate function. Although the calculations were simple, the sheer volume and floating-point data type calculation inefficiencies (vs integer calculations) resulted in a very hardware intensive application. The previous tests exhibit the need for hardware acceleration in this application.