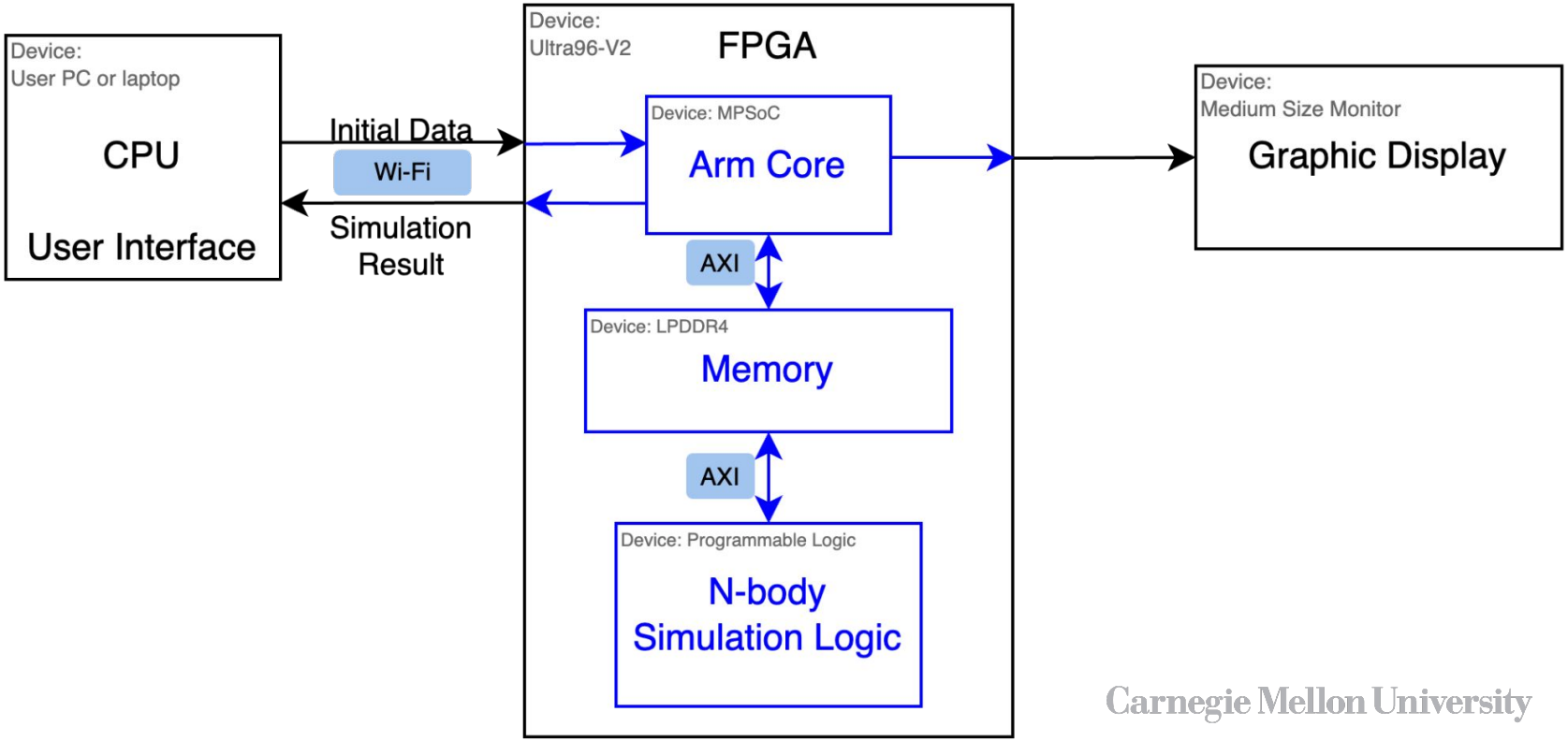
Team A3 N-Body

Abhinav Gupta, Rene Ravanan and Yuhe Zheng

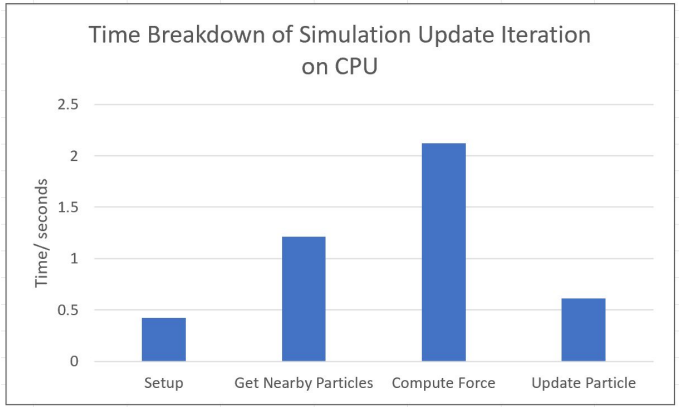
Use Case

- **Problem:** For physicists the N Body simulation is an important and computationally hard problem to solve, trying to run the algorithm in parallely on a CPU is simply not fast enough, and running on GPU is not power efficient.
- **Solution:** Run the N Body simulation on a FPGA and try to achieve a gain in performance.
- **Goal:** Achieve a **~10x Speedup** for a **10000 particle** 2D-simulation.
- **Achieved Speedup: ~40x** 

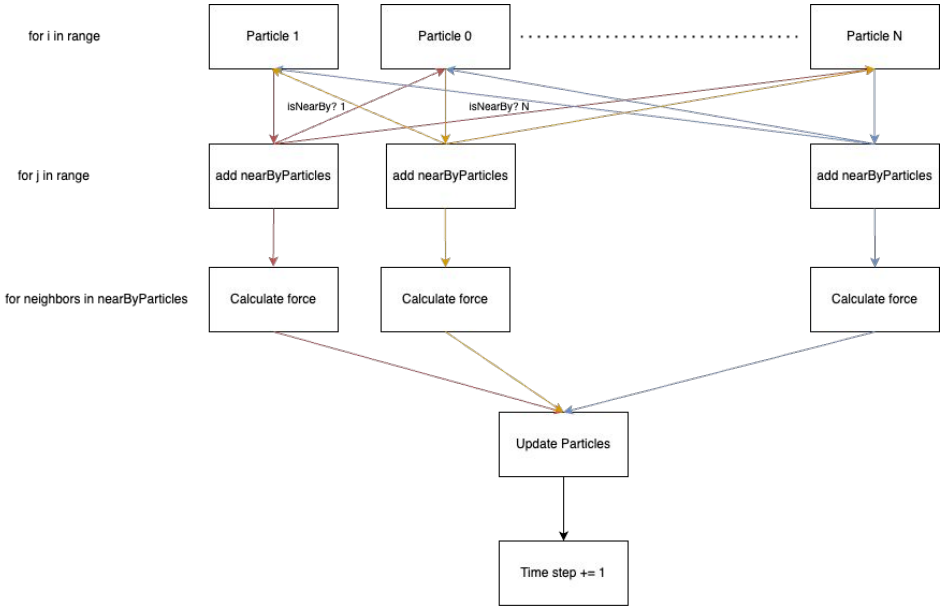
Solution Approach and System Overview



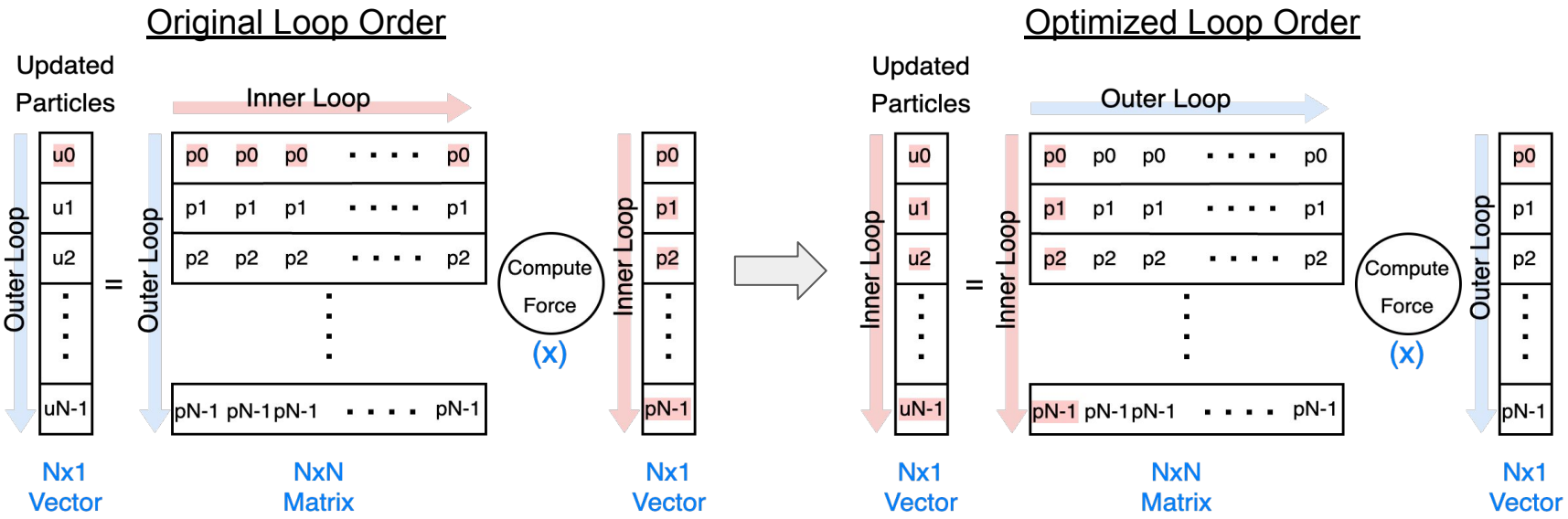
Base Line: N-Body Simulations on CPU



CPU Runtime: ~4.2s per iteration/time step
Running on i7-9700 3.0GHz, parallelised with the Quad Tree Approach



Solution Overview: Matrix-Vector Multiply Model



Inner loop:
- element-wise compute force
- accumulate forces across iterations

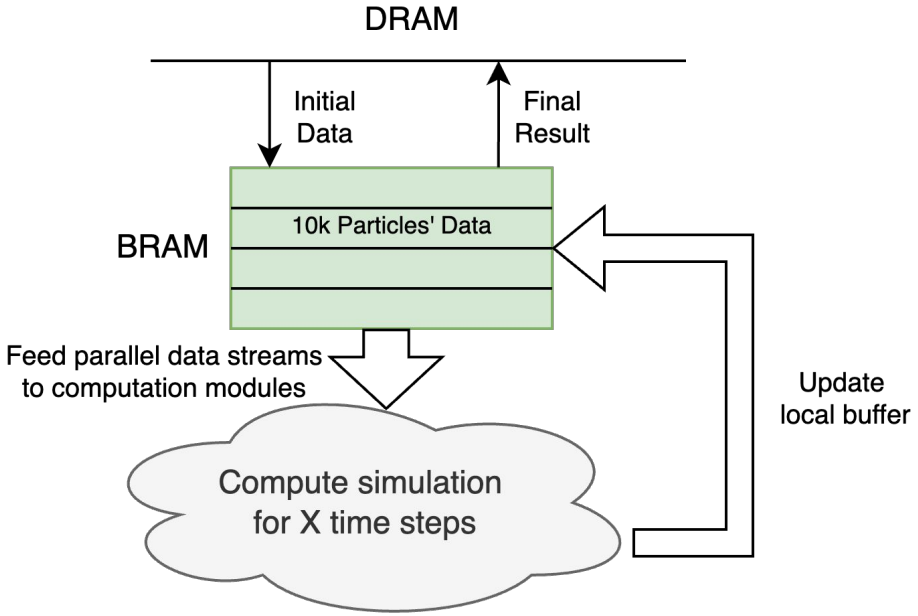
data dependency → pipeline stall

Inner loop:
- element-wise compute force
- element-wise add forces

Solution Breakdown - Minimize DRAM R/W

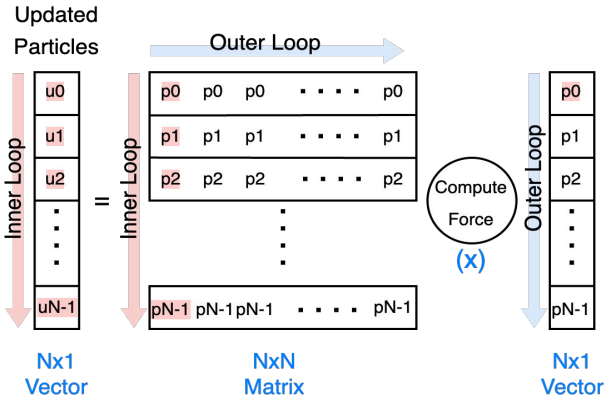
BRAM:

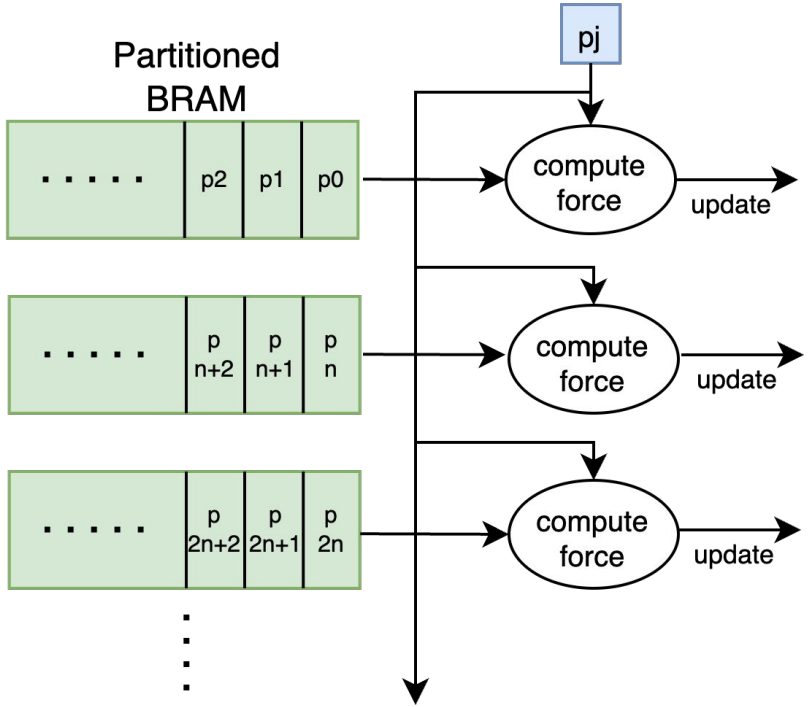
- The board has roughly 432 KB of BRAM, this allowed us to load our entire dataset (roughly 200KB) onto it.
- Block RAM reduced our R/W latencies to memory and also allowed for more concurrent access patterns.



Solution Breakdown - Batch Execute

Since iterations in the inner loop are element-wise, we can compute force on multiple particles at the same time



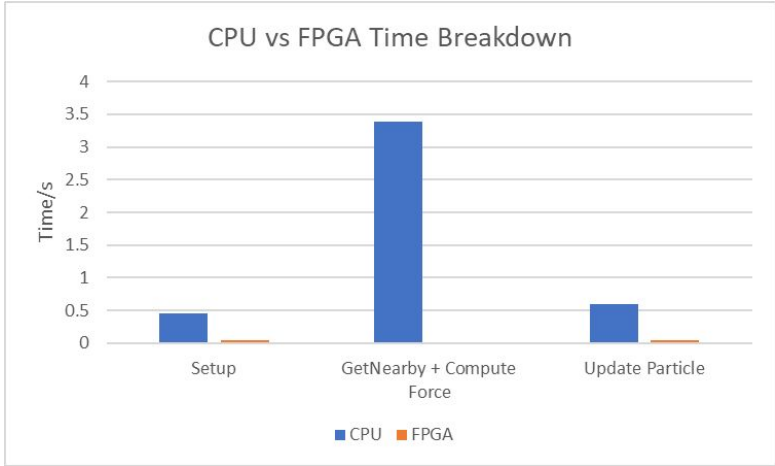


n = number of particles (N) / batch-size

Testing & Verification

Quantitative Verification:

- Correctness: For our use cases (Molecular, Astronomical etc.) a **90-95%** accuracy with our reference solutions would be sufficient for our use cases.
- Used a CPU implementation running on an i7 CPU using double precision floating point numbers to compute reference solution.
- Achieving this accuracy means that our final output should not differ from our reference by more than 10%.
- Compared to our Parallelised CPU implementation on the right, we achieved a **40x Speedup!**



$$\frac{\sum_{j=1}^{\text{Num Iter}} \sum_{i=1}^{10000} \left[\frac{\left| \frac{x_{ref} - x_{test}}{x_{ref}} \right| + \left| \frac{y_{ref} - y_{test}}{y_{ref}} \right|}{2} \right] \times 100}{\text{Num Iter} \times 10000}$$

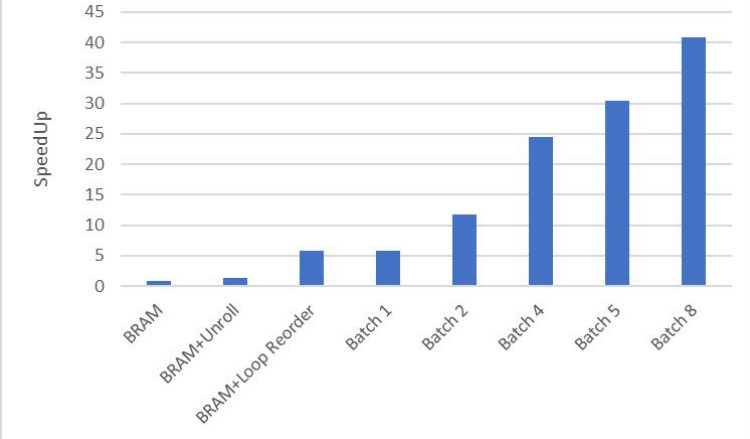
Approaches We Tried

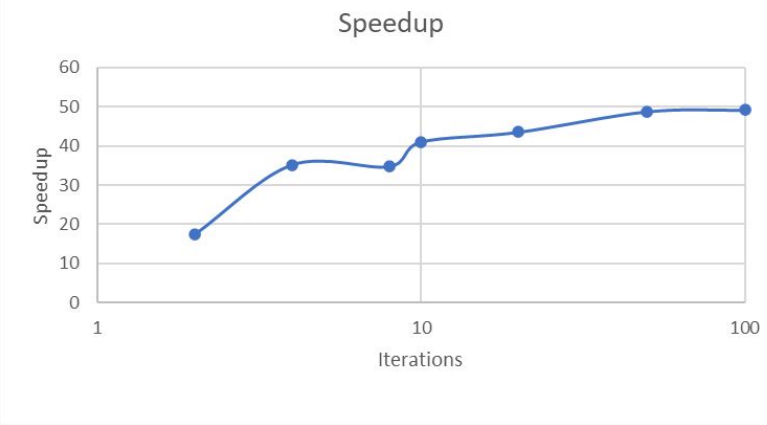
Approach	Success	Reason
Unrolling + Pipelining Alone	**NO**	This alone was not enough to gain any performance. This was because of our DRAM port not being wide enough to support several concurrent reads/writes.
DRAM Widening	**NO**	Due to limited port availability, this did not solve the issue above
Fixed Points	**NO**	While this did produce significant speedup, we later realised that this was not accurate for longer iterations due to loss of precision. Increasing bit width was not feasible for our resource utilisation
Structs for Particles	**NO**	While they made our code easier to read, they increased our hardware utilisation
BRAM	**YES**	Improved our R/W Latency
Batching + Unrolling + Pipelining	**YES**	Improved latency for our concurrent memory operations. This coupled with BRAM above, set up unrolling + pipelining

Our Results

- Given that we stuck to using floating point types, we ended up with a **96% accuracy** to our reference CPU implementation !
- Our CPU implementation (Running on i7-9700 with a parallelised QuadTree implementation) ran for roughly 4.2s per iteration, we produced roughly 0.102s per iteration giving us a **~40x Speedup** for 10 iterations!
- This scales non-linearly with number of iteration (time steps) due to Amdahl's Law

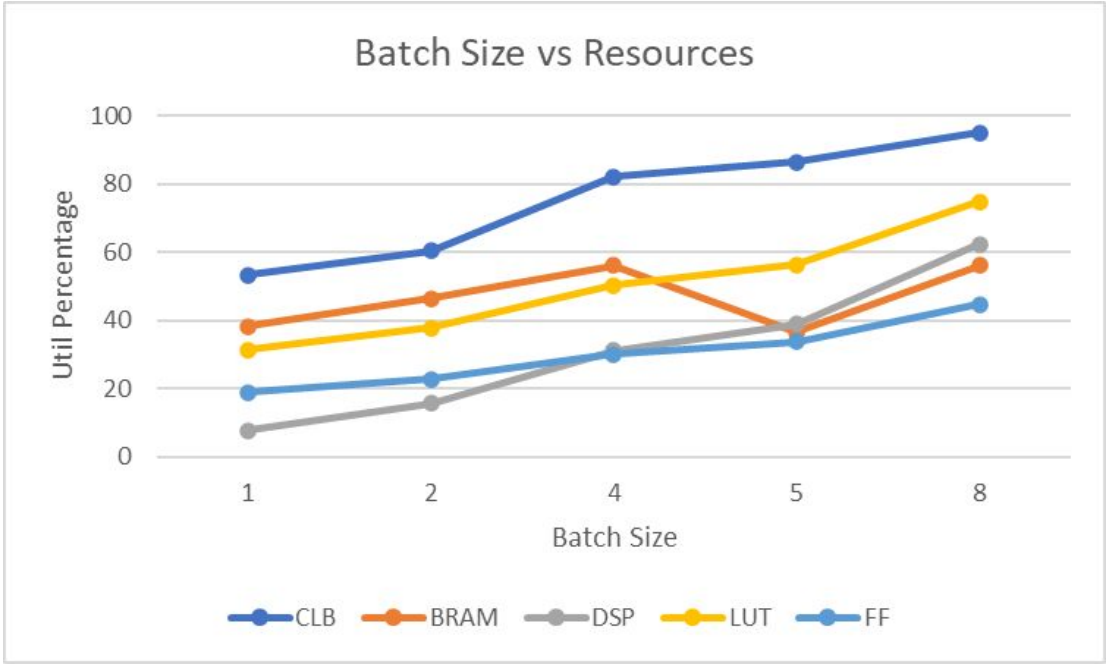
$$Speedup = \frac{1}{(1-p) + p/N}$$





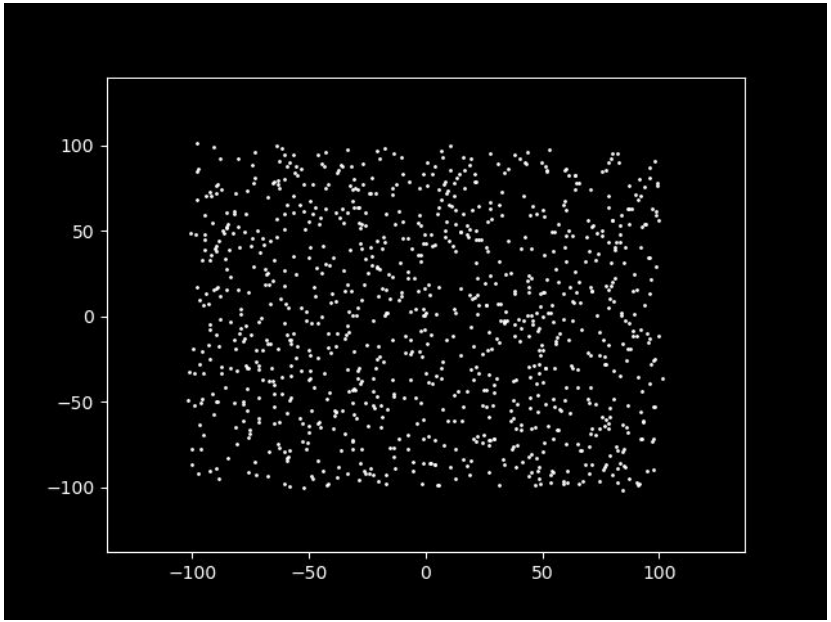
Trade Off

Speedup (batch size) vs Resources



Results (Graphics)

- We have 2 working scripts for graphical simulation (C++, Python)
- Getting the simulation displayed directly using on the FPGA has turned out to be very cumbersome
- In addition to running our display on the board, we are looking into other alternatives like the FPGA communicating with an AWS server to do the simulation.



Project Management