



Boston University
Electrical & Computer Engineering
EC464 Capstone Senior Design Project

Final Test Report

**High Performance Molecular Dynamics on Various
Parallel Architectures**

by

Team 17

Vance Raiti, Austin Jamias, Emika Hammond, Sora Kakigi, Fadi Kidess

Vance Raiti vraiti@bu.edu
Austin Jamias ajamias@bu.edu
Emika Hammond eth@bu.edu
Sora Kakigi skakigi@bu.edu
Fadi Kidess fadik@bu.edu

Testing Environment

GPU

- NVIDIA A40 provided by the SCC
- Simulator code (<https://github.com/md-hpc/cuda>)
- PDB files generated by our Python scripts
- Output csv files (containing particle positions)
- Output txt files (containing timing data)

CPU

- Linux terminal
- Simulator code (<https://github.com/md-hpc/CPU>)
- Visualization (Python)
 - Matplotlib
 - Numpy

FPGA

- Access to the Open Cloud Testbed (OCT) FPGA environment
- FPGA bitstream generated on the New England Research Cloud (NERC)
- Input particle files generated by our Python scripts
- Output txt files (containing particle positions)

Set-Up

CPU

- Git clone <https://github.com/md-hpc/CPU>
- `make -C vec` will create the simulation binary. `make -C ref` will create the reference binary (used only for validation).

GPU

- `git clone git@github.com:md-hpc/cuda.git`
- From home directory run `./scripts/batch_job.sh`
 - This will schedule jobs run on the SCC for N^2 , N^2 with shared memory, and N^2 with N3L implementations with various particle counts
 - Timestep count is 1 and timestep duration is 1 femtosecond (10^{-15} seconds)

- Can also do this all manually by typing “make” to compile the needed source files to test the different implementations

FPGA

- Run an experiment on the OCT containing a PC host connected to an Alveo U280 FPGA.
- Send the bitstream file and the host program from the NERC platform to the OCT PC host and set up the system requirements.

Testing Procedure

CPU

- Provide CLI arguments: `./sim [--save] [--universe-size <length>] [--particles <count>] [--resolution <timesteps per animation frame>] [--pdb <path to PDB file>]`
- To measure performance, only provide `--universe-size`. Particle count will be automatically set to appropriate density.
- To visualize, provide at least `--save` and `--resolution`. After the program exits, copy `particles`` to viz and run `viz.py``. This will produce a GIF of
- To validate, create a PDB file specifying initial conditions of the simulation. This file can be passed to both vec/sim

GPU

- You can manually execute the different binaries by typing `<executable> <input_file.pdb> <output_file.csv>` to get the positions of the particles at the last timestep.
- As of right now, the way we perform timing is by using the linux “time” command, but we plan to time only the timesteps in the future

FPGA

- Run the host program on the OCT host PC by running the ‘run.py’ program.
- Compare the results with that of the emulator for the given input.

Measurable Criteria

CPU

1. Programs must be correct WRT reference program (N^2)
2. Program performance must exhibit linear scaling WRT problem size
3. Program performance must exhibit scaling WRT processor count

GPU

1. Simulations output valid particle data
2. Shared memory and global memory accesses should be coalesced
3. Simulations must minimize atomic operations when possible

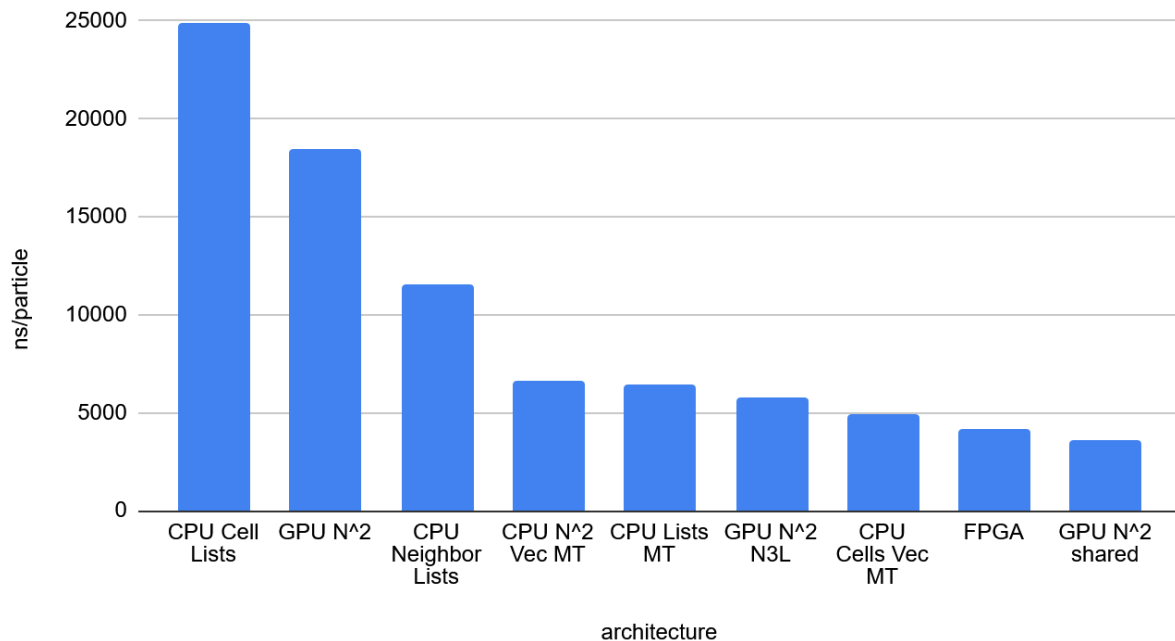
FPGA

1. Simulation outputs must not exceed 1×10^{-7} when compared to Python verification scripts (attributable to floating-point roundoff error).
2. Schematic must match the designed architecture with no error in connections.
3. FPGA resource utilization must not exceed the Alveo U280 resources when synthesizing the design

Results and Discussion

Overview

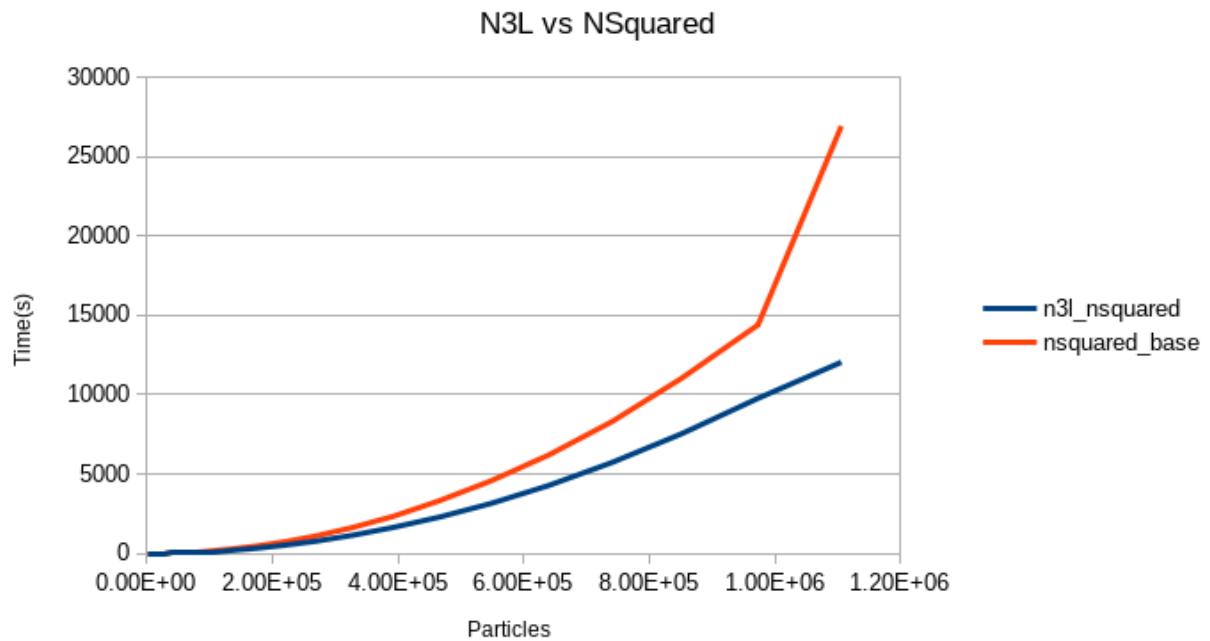
Per Particle Performance (3x3x3 Space, 1 timestep)



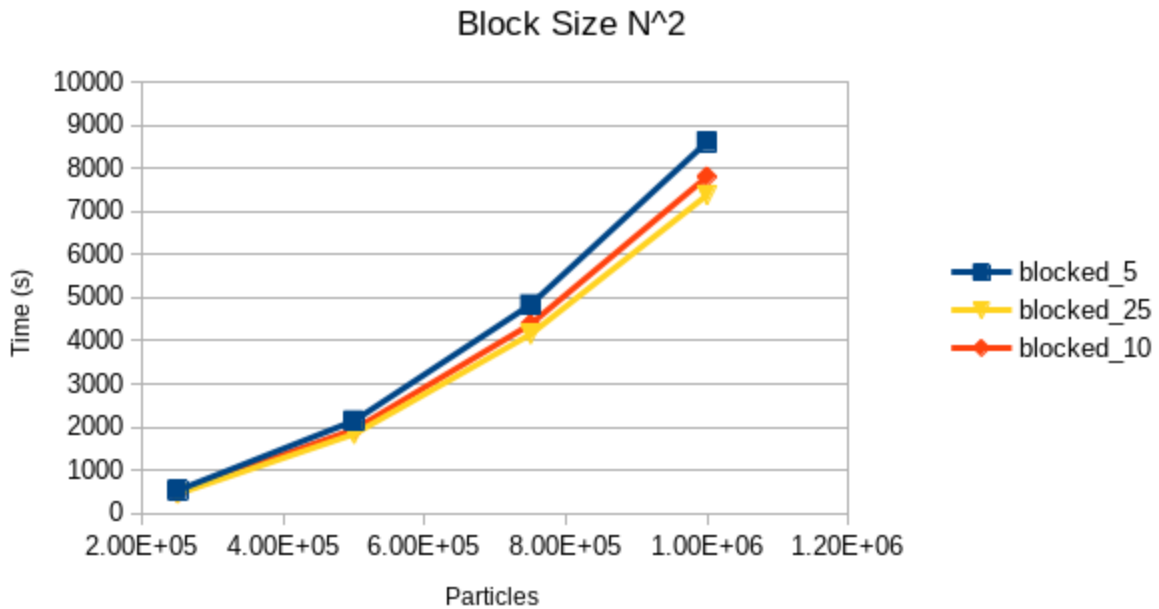
We find that we were able to get the most performance from GPU when using Shared Memory. Several algorithms are included for both CPU and GPU that optimize to varying degrees for algorithmic complexity (N^2 vs cell/neighbor lists), FP utilization (CPU vectorization), and memory (GPU shared memory). FPGA proved more challenging to implement, so only one design iteration is shown here.

Some surprising results include the relative competitiveness of CPU compared to GPU and FPGA, especially when shared memory is not used, and the performance of GPU as compared to FPGA. We had expected FPGA, being entirely optimized for the application, to be by far the most performant.

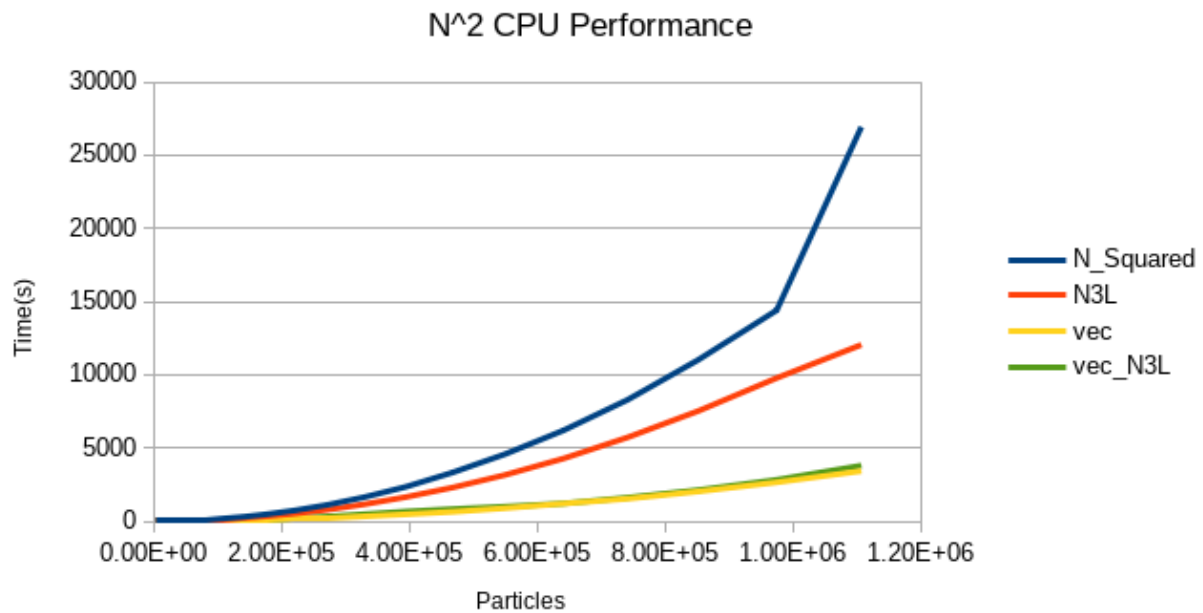
CPU



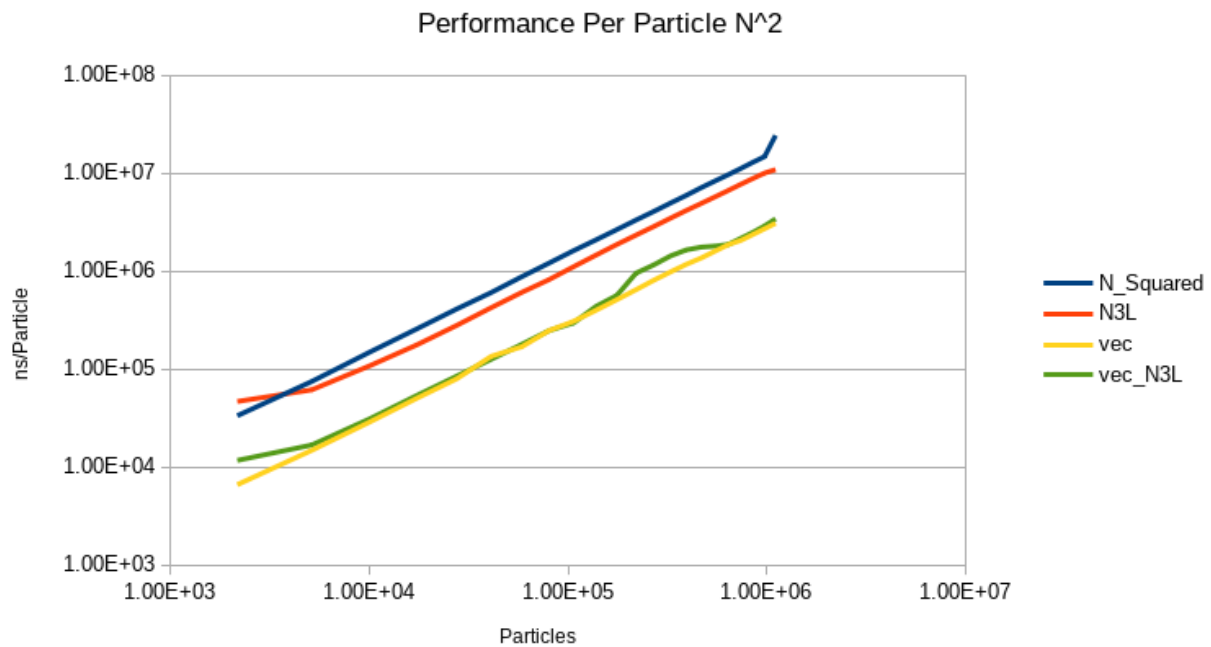
The Newton's Third Law (N3L) method allows us to effectively half the work done by the N Squared version. By converting the forces' inverse on every iteration, we update two particles instead of one from a single force calculation. This graph compares the two's performance and shows the large difference given by their respective times. N3L here looks to be more scalable to large datasets.



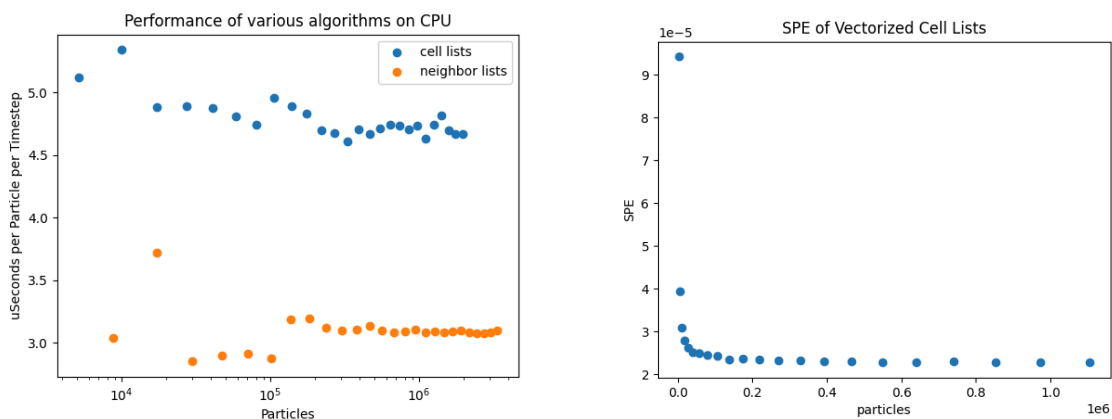
This graph shows the blocking method (where we split the main loops into smaller chunks), with different size blocks run on the N² implementation. Here, the improvement going from a 5-block size to a 10-block size is quite large, while the improvement from 10 -> 25-block size is noticeably smaller. However, in general the improvements from blocking is expected in this version, because the cache will be able to hold the smaller blocks much more effectively than trying to hold the entire space of particles at the same time. Blocking allows for much greater temporal locality than base N² especially as the particle count increases.



This graph compares the vectorized performance of the two versions. Here we can observe that the vectorized time is much faster than the non-vectorized time (approximately 5x). However, one interesting point is that the N3L for the vectorized version performed similarly to the regular vectorized version. One hypothesis is that the introduction of a second write introduces more time lost due to threading conflicts than worth the speedup of having to compute less.



This graph shows the per particle performance of the n^2 algorithms. Due to the nature of an exponentially increasing amount of interactions when increasing the particle count (universe size), the performance per particle starts (in small universes) at a reasonable ~6000 nanosecond per particle time, however quickly explodes to over 3 million nanoseconds, when the universe is 24^3 or 1.1 million particles. This shows the lack of scalability with n^2 methods, and displays the baseline for algorithmic approaches.

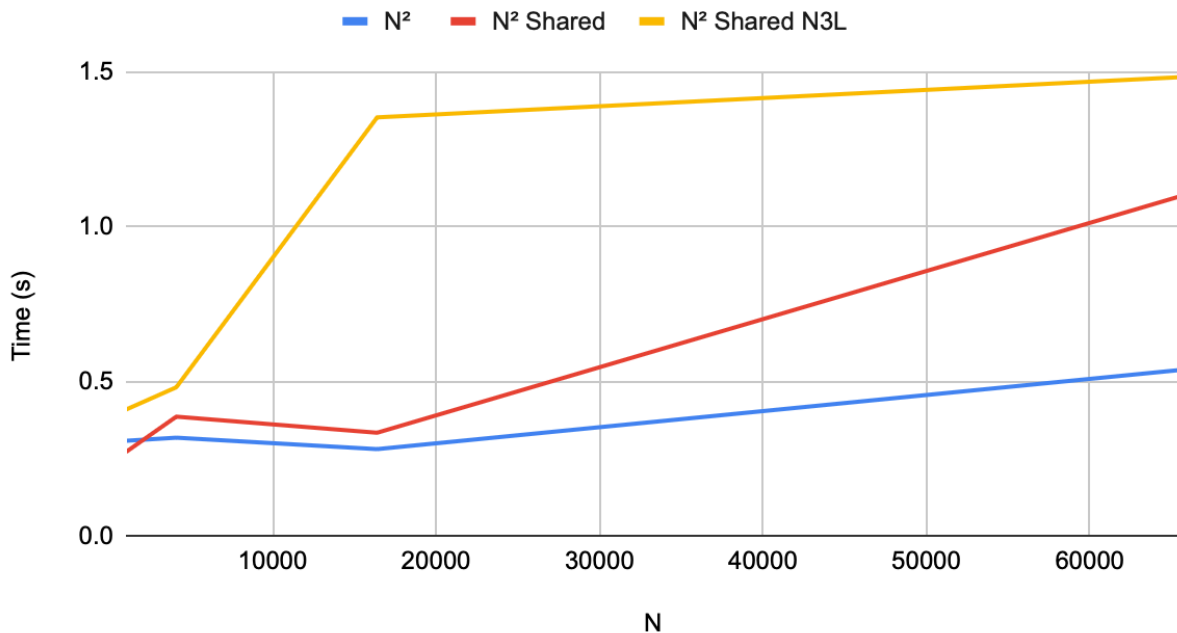


Benchmarking our three $O(n)$ algorithms, we can observe that the scaling is in fact linear, with neighbor lists being consistently more performant than cell lists, and SIMD cell lists being consistently more performant than neighbor lists. A surprising result is that the performance

seems to not depend on problem size: typically as the working set exceeds the size of L1, L2, and L3 cache you see spikes in runtime as the program must now store its working set in the slower memory. This is most likely due to compiler prefetching, as both programs make the majority of their memory accesses to sequential members of arrays, allowing the compiler to easily overlap computation with memory access.

GPU

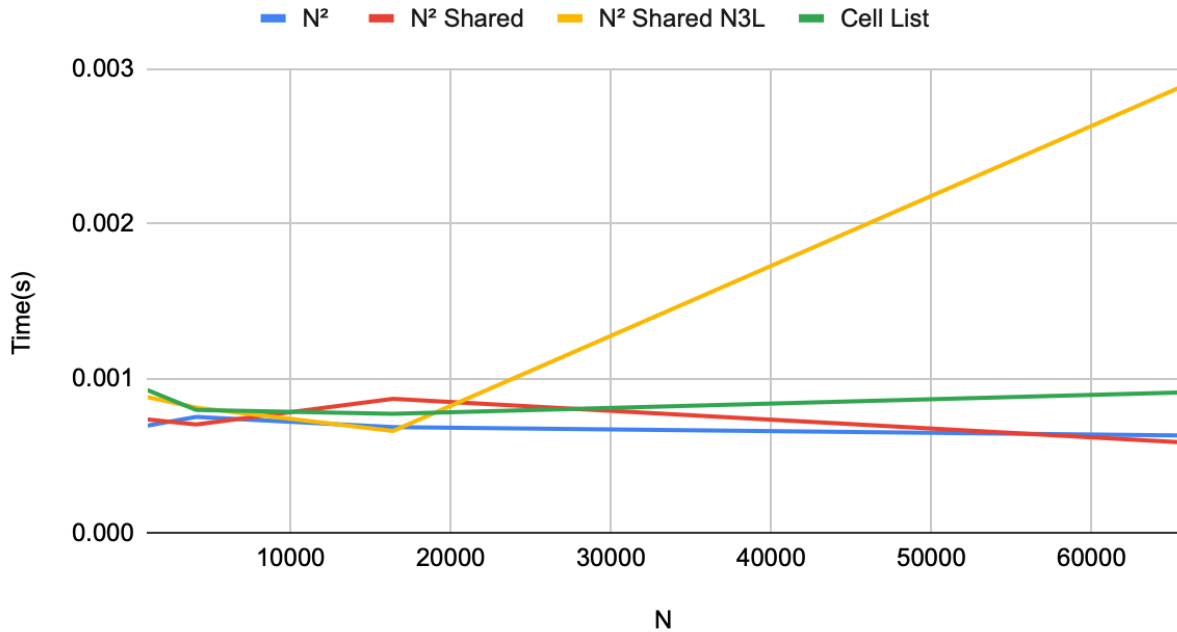
N^2 , N^2 Shared and N^2 N3L as a function of particle count (N)



Performance metrics of running different MD implementations on varying particle counts using Particle lists

In our previous implementation using an array of particles, N^2 and N^2 Shared have similar linearly scaling results while N^2 with N3L appears to be the slowest implementation. While we predicted that the N^2 with N3L implementation would be the most performant, it is possible that it runs longer because the host calls a new kernel to sum up the accelerations of each block, which means many more accesses to slow global memory. The results show that the N^2 Shared and N^2 with N3L need to be looked into and their less efficient runtime compared to N^2 point to design inefficiencies.

Simulation time as a function of particle count (N)



Performance metrics of running different MD implementations on varying particle counts using arrays

We were able to get N^2 Shared and N^2 N3L to run faster by decreasing the number of particles in a block to 32 which is the number of threads in a warp. Because warps move together, we were able to remove thread synchronization from our code which was taking a significant amount of time of our simulation. Later we made even more optimizations by completely switching data structures from an array of particles (which included particle id, x, y, z positions and x, y, z velocities) to arrays of floats (one array for each field). After switching to arrays of floats to store particle data you can see a significant performance increase, around 500X, compared to the previous graph.

Validation

We validated our simulation by proving conservation of momentum. The results are as follows:

momentum x: 0.000007151626

momentum y: 0.000000731088

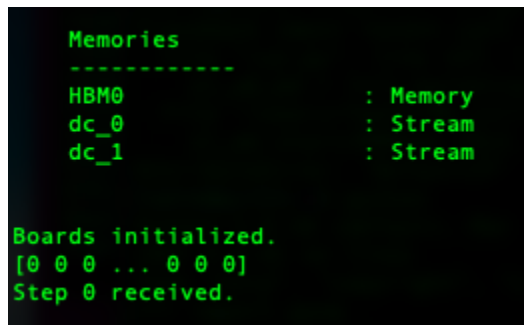
momentum z: -0.000001127366

This allows us to verify that our simulation follows physical laws, and shows that there is not much drift due to numerical errors.

FPGA

After running the simulation for multiple timesteps for 300 particles in a 3 by 3 by 3 universe, we found that each timestep for this problem size takes about 50,000 clock cycles. At a 30 MHz clock, that translates to 600 timesteps per second.

Our simulation results for each timestep matches that of the emulator. When running the bitstream on the FPGA, however, we faced issues with communicating to the PC. There are also some issues that arise when generating the bitstream caused by congestion issues. However, as shown in Figure 1, we are able to push the bitstream to the FPGA and establish a communication channel.



```
Memories
-----
HBM0      : Memory
dc_0      : Stream
dc_1      : Stream

Boards initialized.
[0 0 0 ... 0 0 0]
Step 0 received.
```

Figure 1. Output logs of an OCT FPGA experiment, where the bitstream was written to the device.

The populated netlist generated by Vivado's RTL analysis shows the physical implementation of our approach. As shown in Figure 2, our design's ring topology contains a series of Position and Velocity caches implemented as BRAMS, each connected to its neighboring cache. The motion update unit and compute pipeline are shown in the design, each containing dozens of instantiations of over 30 modules.

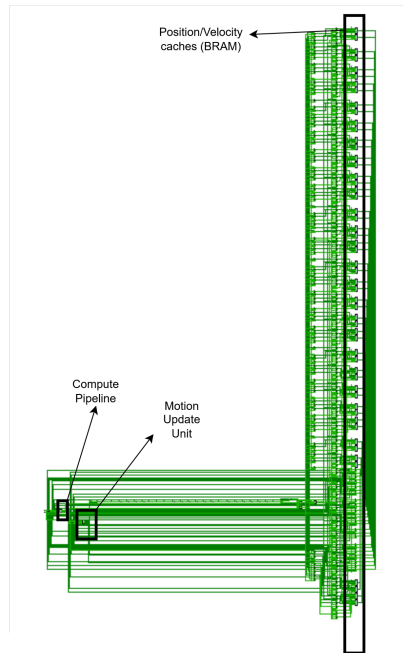


Figure 2. RTL netlist of our top module, showing the ring node of position and velocity caches, as well as the compute pipeline and motion update unit.

Based on synthesis reports, our design utilizes roughly 11 BRAMs per cell. Using a 3x3x3 simulation size, that corresponds to around 300 BRAMs out of the 1,776 BRAMs available. Our design utilizes around 60% of the FPGA's resources (including LUTs) when using the same parameters as the emulator.