Optimization of Molecular Dynamics Simulator with Cell Lists and Range-Limited Forces

Alex Melnick and Maura Mulligan
Boston University EC 527

Introduction to Molecular Dynamics

Molecular dynamics has many applications. At a high level, it is the study and computation of how molecules and atoms move in space over time. This requires modeling basic physics and interatomic interactions. This modeling is used across fields such as protein folding, synthesis of compounds, simulation of molecules interacting with light and electricity, chemical reactions, and many other areas [1].

As the number of molecules increases, more types of forces are considered, the time and resolution at which to be simulated increases, and other factors are adjusted, this simulation grows tremendously. This is why simulating molecular dynamics could greatly benefit from optimization and parallelization. Additionally, the repeated calculations for *every* atom in the simulation lends itself nicely to many of the techniques learned in this class. Through optimization, higher-fidelity simulations will be more feasible of a solution and more insightful in a timely manner.

Model Assumptions

Computing molecular dynamics equations can be treated as an initial value problem when given initial positions and velocities for all the particles. For the sake of simplicity the model limits what forces are taken into consideration. Liquid argon is used for our particles so the only interatomic force that is considered is the Lennard-Jones potential. The Lennard-Jones (LJ) potential is the sum of a close-range repulsive force due to electrons not being able to share shells and a long-range attraction as the molecules polarize [2], and it can be seen in Equation 1. All the constants and values used were normalized from the start to not result in overflow and underflow errors.

$$u(r) = 4\epsilon[(\frac{\sigma}{r})^{12} - (\frac{\sigma}{r})^{6}] \quad (1)$$

We can observe that the force diminishes rather quickly with the high exponent in the denominator. As a result, a cut-off is used for which ranges to consider when calculating the forces before they become negligible. Another assumption is that the simulated bulk medium is an infinite cube. This requires boundary checks/adjustments with each time step, but allows us to ignore the difference between surface and bulk behavior. Using these assumptions, time can be discretized and the initial value problem can be solved leveraging Newton's second law ($F=ma$) and Euler's method.

Baseline Serial Implementation

The baseline serial code was taken from University of Southern California's Scientific Computing and Visualization course. It derives a step of equations to update position and velocity over time. The specific algorithm they leverage for calculating the trajectories is the velocity-verlet which uses Taylor expansion. The velocity-verlet algorithm is similar to Euler's in that it discretized time and leveraged position, velocity, and acceleration all being derivatives of one another. This algorithm is structured so all the dependencies on trajectory values at a specific time are calculated prior to when they are needed. To calculate the acceleration the Leonard-Jones potential was used with a cutoff distance and fed into Newton's second law.

High-Level Algorithm

Initialization consists of distributing *n* atoms evenly in a lattice formation. The velocity of each is randomly generated but has a magnitude proportional to the temperature. Initial acceleration is calculated and then the velocity-verlet algorithm is applied. For each time step, the velocity is updated based on the acceleration, the position is updated based on the velocity, and the new acceleration is calculated. This algorithm is repeated for *k* time steps as set by the user. Figure 1 shows this approach. This is nearly identical to the code implementation, however a boundary check/adjustment for the infinite medium was added right before calculating the new acceleration.

(Velocity-Verlet algorithm for `StepLimit` steps)

Initialize $(\vec{r}_i, \vec{v}_i)$ for all i

Compute $\vec{a}_i$ as a function of $\{\vec{r}_i\}$ for all i—function `computeAccel()`

for `stepCount` = 1 to `StepLimit`

do the following—function `singleStep()`

$\vec{v}_i \leftarrow \vec{v}_i + \vec{a}_i \Delta / 2$ for all i

$\vec{r}_i \leftarrow \vec{r}_i + \vec{v}_i \Delta$ for all i

Compute $\vec{a}_i$ as a function of $\{\vec{r}_i\}$ for all i—function `computeAccel()`

$\vec{v}_i \leftarrow \vec{v}_i + \vec{a}_i \Delta / 2$ for all i

endfor

Figure 1: Implemented velocity-verlet algorithm [2]

Complexity Analysis

All of the updates are O(n) complexity as they only require iterating through each atom and each dimension once. The bulk of the computation is the computing of acceleration. This is $O(n^2)$ as each atom has a force applied to it by every other atom that must be accumulated. As the number of atoms, *n*, increases, this acceleration calculation quickly dominates all others for where the most time is spent. The arithmetic intensity (FLOPS per memory access) is 38/9 or 4.2 for acceleration calculations between those within the cut-off range. For those outside this region, it is 7/6 or 1.17 which is more likely to be bound by memory. While it depends on the size of the

simulation, typically the larger majority of atoms will be in the latter region where they will be more likely to be memory bound. Optimizations for this code will look at both computational speed up for the frequent (and costly) square root calculations, as well as how to best arrange memory to take advantage of the cache hierarchy.

Memory Analysis

What are the primary data structures? What is the memory reference pattern?
The way memory was structured was a "struct of arrays". Aside from constants and other smaller values to maintain for calculations, there were three primary arrays. There is an array for position, `r[NMAX][3]`, velocity, `rv[NMAX][3]`, and acceleration, `ra[NMAX][3]`. Each update calculation goes atom by atom, and dimension by dimension for each, which lends itself nicely to this memory structure. Outside of the initial compulsory miss, our algorithm benefits greatly with the sequential memory accesses. For the acceleration calculation, a similar memory reference pattern can be seen but within a nested for-loop.

The use of an "array of structs" was also considered. However, given that each step in our algorithm only considers two of the three arrays (i.e. updating velocity only cares about velocity and acceleration, updating position only cares about position and velocity, and calculating acceleration only cares about acceleration and position), this would waste ⅓ of each cache block. Especially when our working set exceeds the cache size and the end of a previous step is servicing a different set of atoms than the beginning of the next step.

Optimizations on CPU

Multiple optimizations were made to the serial CPU implementation that consisted of content from the course and algorithmic modifications. The first three introduced, while improvements, are still $O(n^2)$. The last optimization introduced is algorithmic and shows the largest speed up as the complexity is brought down to O(n).

Code Restructuring

Given the computational complexity of the calculations, especially the time required for a square root, moving around the code to minimize this resulted in some benefit. The distance calculation requires squaring each dimension and then taking the square root of the sum. By changing the comparison for being in the cutoff region from the cutoff to the cutoff-squared, the square root no longer has to be taken for *every* distance calculation. Given the majority are outside of this region, this saves noticeable time. Additionally, computation that is based on distance (varies with each pair) is split into smaller computations that are stored in local variables. This saves time when those smaller computations are reused in the final equations. Finally, any computations in logic control are pulled as far out as possible and computed there to minimize duplicate computations.

SIMD by Vectorization with AVX

The updating of each atom is an ideal application for using SIMD. For each dimension of each atom the same calculation is being performed. Either incrementing velocity by the product of half a time step and acceleration, or incrementing position by the product of a time step and velocity, the calculation is the same for every single element in the array. Since the algorithm uses doubles, AVX was used which can hold four elements in a vector register. The code for updating the position is

```
__m256d   m1;
__m256d* pos = (__m256d*)r;
__m256d* vel = (__m256d*)rv;
__m256d dt = _mm256_set1_pd((double)DELTAT);

int nloop = 3*nAtom / 4;

for(n=0; n<nloop; n++) {
     m1 = _mm256_mul_pd(*vel,dt);
     *pos = _mm256_add_pd(*pos,m1);
     pos++;
     vel++;
}
```

which is very similar to the updating of velocity. Similarly, updating the boundary conditions was also vectorized. This required a logical AND for checking the sign, and a logical XOR to flip the sign efficiently. The code to vectorize `if (x > 0) return v; else return -v;` looks like the following:

```
int n;
__m256d m1, m2, m3, m4;
__m256d* pos = (__m256d*)r;
__m256d regionH = _mm256_set1_pd((double)RegionH[0]);
__m256d region = _mm256_set1_pd((double)Region[0]);
uint64_t signbit[] = {0x8000000000000000, …};
__m256d negative = _mm256_loadu_pd(&signbit);

int nloop = 3*nAtom / 4;

for(n=0; n<nloop; n++) {
     m1 = _mm256_and_pd(*pos,negative); //find negatives
     m2 = _mm256_xor_pd(regionH,m1);    //flip if negative
```

```
        m3 = _mm256_sub_pd(*pos,region);
        m1 = _mm256_and_pd(m3,negative);
        m4 = _mm256_xor_pd(regionH,m1);
        m3 = _mm256_add_pd(m2,m4);
        *pos = _mm256_sub_pd(*pos,m3);
        pos++;
    }
```

These adjustments do add a little bit of overhead but they also are able to execute four operations in parallel. Given that these AVX adjustments were only made to update steps which run with O(n) complexity they allowed for some speed up, but were ultimately still limited by the $O(n^2)$ complexity of calculating acceleration.

With more time, vectorization of the acceleration function would like to be explored. Calculating the distance could benefit but would require a mask to only consider the first three elements. Another option would be to calculate the fourth element but discard it when accumulating. This would require it to be recalculated on the next iteration which wouldn't be ideal, but the vectorization still will benefit by computing three operations in parallel. Similarly, some loop-unrolling could also be added to help vectorize the acceleration calculation - however, this can pose some issues when divergence occurs due to some being within and some being outside the cutoff.

Newton's 3rd Law

Newton's third law states that "every action has an equal and opposite reaction". Therefore, only one interatomic force (LJ potential) needs to be calculated for each pair of atoms. This alone should nearly halve the total runtime as now only half the computations are being completed. When added to the other optimizations mentioned, it *more than* halves the total runtime.

This was implemented very easily by simply changing the bounds of the inner for loop from

```
for (j1=0; j1<nAtom; j1++) {
          for (j2=0; j2<nAtom; j2++) {
… }}
```

for the serial baseline to

```
for (j1=0; j1<nAtom; j1++) {
          for (j2=j1+1; j2<nAtom; j2++) {
… }}
```

for our “N3L” implementation. While this greatly improves the baseline, it does pose issues down the line for parallelization. Rather than each atom updating its own acceleration accumulator value, other atoms will be updating it as well causing read/write conflicts that must be managed.

Cell Lists

The use of cell lists showed the greatest improvement. This algorithmic change brought the complexity of calculating the acceleration down from $O(n^2)$ to $O(n)$. This works by dividing the simulated area into “cells”. These cells split the 3D area into smaller cubes with a length just longer than the cutoff distance. Since we only care about atoms within the cutoff region, we only need to parse the 26 neighboring cells (and its own - the 27th cell) for each atom. As the total number of atoms grows, the number of how many are relevant for calculating the acceleration stays constant. The previous optimizations mentioned can also be included with this such as Newton’s third law and code restructuring.

All the cells are stored in a single contiguous array to continue to take advantage of spatial locality. Again, separate arrays for position, velocity, and acceleration. These arrays have double the storage allocated, and it is distributed across each cell’s section to act as a buffer for atoms to move in and out of cells. Another array, the `head_tail[NCLMAX][2]` stores what the index into the main arrays of each cell’s start and end are. The start is fixed, and the end points to the first empty slot. The end pointer allows for atoms to be inserted into the cell if they move locations. Atoms that leave a cell are filled with an `EMPTY`, or -1, to alert our algorithm to ignore them. The algorithm must initially sort all atoms into their respective cells, and then every five time steps these arrays are condensed. This happens by filling the empty slots with the elements from the end to minimize the amount of memory movement. The five time steps threshold was chosen randomly to err on the side of caution and not overwrite another atom’s data. While there is some overhead in condensing the lists, it’s still a better implementation than linked lists which jumps erratically across memory.

Cell lists showed the most improvement which is expected when decreasing the complexity exponentially. However, the added buffers and empty slots made vectorization opportunities less inherent. For parallelization, it could be used as a way to partition the blocks and threads by, but the arbitrary size of each cell might result in the waste of some threads.

Optimization Through Parallelization

After optimizing the serial code on the CPU, attempts to further speed up the algorithm moved to multiple cores and GPUs. These consisted of interfacing with OpenMP and CUDA.

Parallelize with CPU: OpenMP

The first optimization used OpenMP to parallelize solely the acceleration computation - by far the heaviest part of each time step - in our three algorithms (baseline, Newton's 3rd Law, and cell list). Early experiments showed that trying to parallelize every phase neither paid off nor improved performance. This could be a combination of the acceleration computation time still dominating small O(n) improvements (similar to what we saw with AVX), or the overhead of OpenMP.

For the baseline implementation, the acceleration loop is wrapped in a `#pragma omp parallel` region. Each thread writes into its own private acceleration array, then an explicit barrier synchronizes them before all partial results are combined into the global array. All loop indices are declared private, and each thread also accumulates its own partial sum of potential energy with `reduction(+ : potEnergy)`.

The Newton's Third Law (N3L) implementation follows the exact same pattern as the parallel baseline: private acceleration buffers per thread, a barrier before reduction into the global accelerations, and per-thread energy sums via pragmas.

For the Cell List implementation, the same techniques from the other two were reused. In addition, when particles move between cells, we protect those updates with an atomic directive to avoid race conditions on shared cell-slot arrays. Finally, we collapse the three-nested loops over cells and neighbors into a single `#pragma omp for collapse(3) schedule(dynamic)` loop. That balances the uneven work - due to branching in the interaction checks - across threads efficiently.

Parallelize with GPU: CUDA

The baseline CUDA implementation mirrors the serial version, but parallelizes across atoms by assigning one thread per atom. All atom data - positions, velocities, and accelerations - are allocated a single time in GPU global memory, and each kernel reads and writes directly to these device buffers. We chose a fixed `BLOCK_SIZE` of 256 threads, and computed the required grid size to launch a one-dimensional grid of one-dimensional blocks.

Each major step of the simulation runs in its own kernel - velocity half-kick, position update, boundary-condition enforcement, acceleration computation, and final velocity half-kick. Between kernels, the host calls `cudaDeviceSynchronize()` to act as a global barrier, guaranteeing that all blocks from the previous kernel have finished before the next one begins. Because the data stays resident on the device, no host–device transfers occur between time steps, keeping overhead to a minimum. As a result, positions, velocities, and accelerations for all atoms

are updated in parallel every time step, with the host solely responsible for orchestrating kernel launches and global synchronization.

Discussion of Results

All results were verified to be working correctly by calculating the potential and kinetic energies every 10 time steps, and those values were compared against our baseline (truth). This computation was removed from the final runtime calculations after each method was verified.

Environment Specifications

Tests were conducted on Ubuntu after a full power cycle and an idle wait period to minimize background activity. All benchmarks were run on a Dell XPS 9520 laptop equipped with:

CPU**:** Intel Core i7-12700H

- Performance-cores (P-cores): 6 cores × 2 threads each, with 48 KiB L1 data + 32 KiB L1 instruction cache, private 1.25 MiB L2 per core, boost to 4.70 GHz
- Efficiency-cores (E-cores): 8 cores × 1 thread each, with 32 KiB L1 data + 32 KiB L1 instruction cache, organized in two clusters of four sharing 2 MiB L2 per cluster, boost to 3.50 GHz
- L3 cache: unified 24 MiB shared by all cores
- TDP: 35W to 115W (45W base)
- System memory: 64 GB DDR5-5600 MT/s

GPU**:** NVIDIA GeForce RTX 3050 Ti Mobile (Ampere GA107)

- Streaming Multiprocessors (SMs): 20
- Streaming Processors (SPs/CUDA cores): 2,560 total (128 per SM)
- Clock: 735 MHz base, 1,035 MHz boost
- Memory: 4 GiB GDDR6
- TDP: up to 35 W

Results

A comparison of all the implementations across different working set sizes (number of atoms) can be seen in Figure 2. The wall-time versus atom count for every implementation is plotted. "Cache filled" denotes the number of atoms required to fill a single P-core's (data) cache. The L1 region (< 683 atoms) is dominated by measurement noise: runtimes are in the millisecond range, so a single OS context switch can distort timing and occasionally inverts the expected ordering (e.g., the erratic AVX–N3L curve).

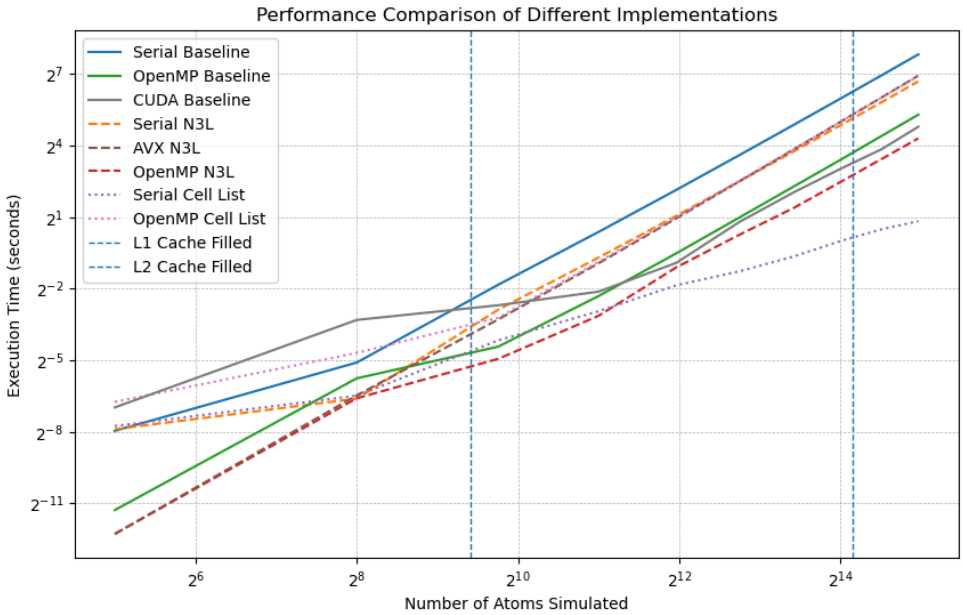

Figure 2: Performance results of all different implementations. Note that "Cache Filled" refers to the amount of data it would take to fill a single P-core's data or combined cache.

From 683 to 18,204 atoms - the L2 range - measurement variance falls away and each curve follows a clean power-law. Both $O(n^2)$ algorithms (Baseline and N3L) exhibit parallel slopes, confirming identical asymptotic cost. In contrast, the Cell-List versions show a visibly shallower slope, consistent with their $O(n)$ theoretical bound. Figure 3 shows this area zoomed in.

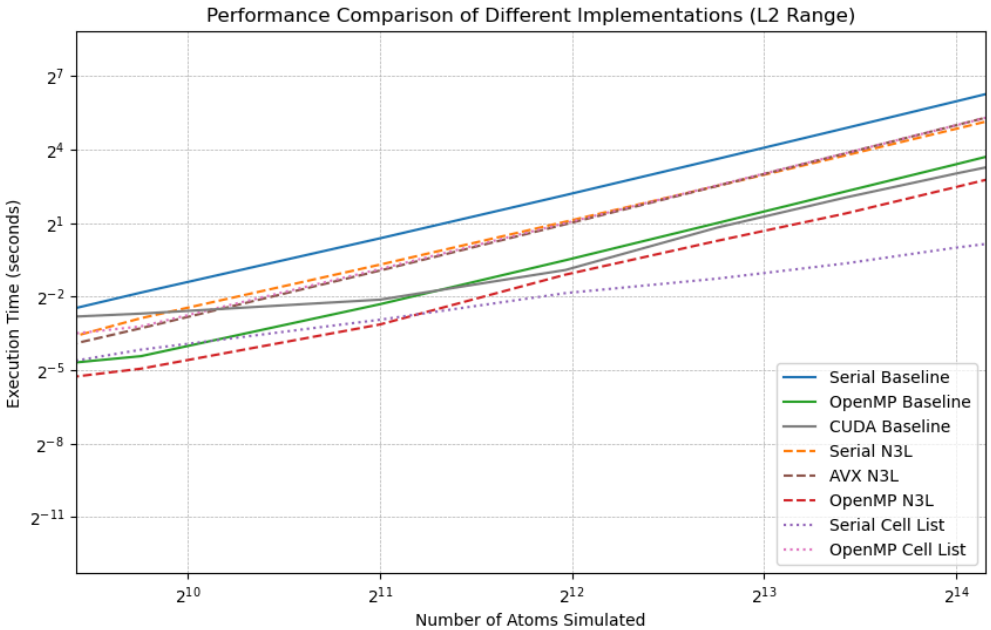

Figure 3: Performance results of all different implementations only in the L2 range (n=6 simulation sizes visible)

The log–log representation makes the slopes (exponent b in $y = ax^b$) explicit. Where $b \approx 2$ for Baseline and N3L (parallel lines), and $b \approx 1$ for Cell-List (lower, gentler slope). The near-perfect linearity indicates that our implementation is compute-bound in this range, with minimal cache-miss interference. Zooming in even further to see all the implementations across a single sample point, one can look at Figure 4. It extracts a single column at x = 4,000 (mid-L2).

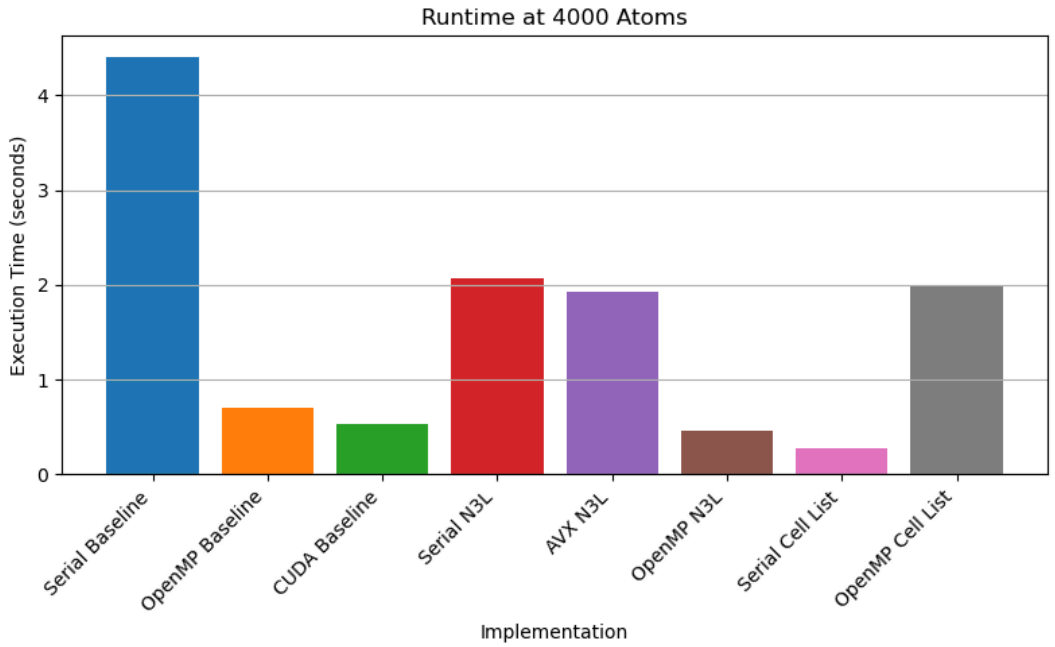

Figure 4: "Slice" of Figure 1 at x=4000 (~2^{12}) atoms. Note that colors of bars match those in Figure 3 and Figure 2.

Since the distance between the trend lines holds with larger simulations, looking at a slice of the data allows for clearer visualization of how each implementation compares with one another. We can first observe that Serial N3L halves the Baseline runtime, as expected from avoiding duplicate force evaluations. The AVX N3L shows some minor speed up for a smaller number of atoms over the serial implementation. As the number of atoms grows, however, the non-AVX compute acceleration ($O(n^2)$) dominates any AVX speedup elsewhere. Next, OpenMP Baseline/N3L achieve an additional 5-6 x speed-up over their serial counterparts on 20 CPU threads. While it's not strong scaling, it still is a significant improvement as the OpenMP N3L is the most efficient $O(n^2)$ implementation. Finally, the CUDA Baseline is only marginally faster than the OpenMP Baseline - both deploy 20-way parallelism (20 SMs vs. 20 CPU threads), so parity is reasonable.

The main anomaly is the OpenMP Cell-List. Although functionally correct, its runtime varies by up to an order of magnitude between identical runs and it occasionally crashes for large numbers

of atoms. It is not currently known why this is happening, however it appears to be memory related from the segfault messages so potentially not enough “buffer” space was allocated to allow for movement of atoms. There could also be an occasional race condition that evaded testing. Looking at Figure 5, it performs much better than the same setup for the Baseline and N3L OpenMP implementations. This further shows variability between runs.

Multithreading Results

Speed-up versus thread count for all OpenMP variants was also tested to examine scaling and can be seen in Figure 5.

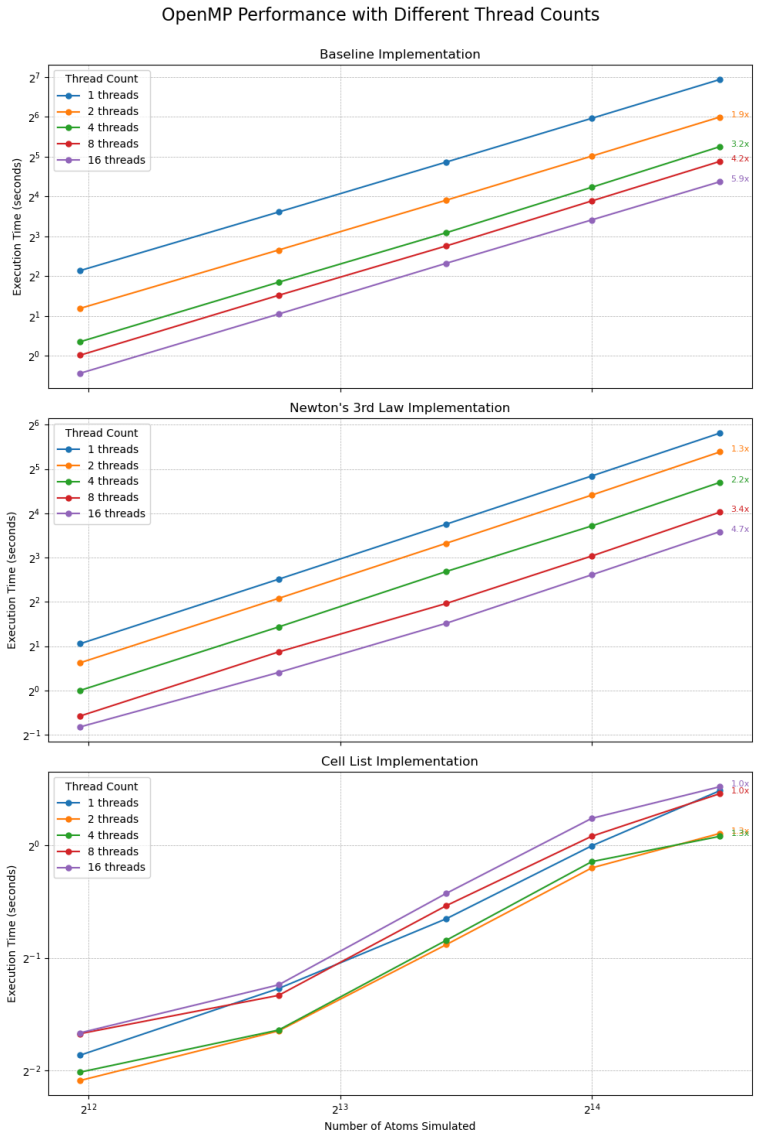

Figure 5: Plot of OpenMP implementations running on various numbers of threads. Speedup compared to the single-thread version indicated.

Scaling is sub-linear for two reasons: 1) Amdahl's Law - only the acceleration kernel is parallelized while the velocity and position updates remain serial, and 2) Memory bandwidth limits - the arithmetic intensity of LJ force evaluation is rather modest. A custom pThreads implementation to have more cache-aware memory access might alleviate some bandwidth pressure, but we expect continued diminishing returns beyond the 20-thread mark on different hardware.

Putting it All Together

Overall, nearly every optimization step behaves close to that predicted by theory. Newton's Third Law removes redundant computations by a factor of 2 and the runtime scales accordingly. This also scaled well with the runtimes. OpenMP and CUDA expose task-level parallelism (~5-6x on 20 lanes) which is mirrored in the runtime speedup. Cell-Lists deliver the only algorithmic change, reducing complexity from $O(n^2)$ to $O(n)$, but the multithreaded version still needs robustness work. Summarizing these gains, the fastest $O(n^2)$ implementation runs ~10x faster than the unoptimized baseline for 4,000 atoms, while the single thread $O(n)$ Cell List outperforms every $O(n^2)$ method at 6.9k atoms and beyond.

Conclusion

The optimization pipeline delivered exactly what theory forecast: vectorising inner loops and exploiting Newton's Third Law cut the scalar runtime in half, and a 20-thread OpenMP pass slashed another order of magnitude. Every one of those steps behaved predictably and reliably. The lone exception was the OpenMP Cell List, whose memory-competing design caused erratic timing and occasional crashes - even though the same algorithm runs flawlessly in single-thread mode.

A surprise at first glance was the near-parity between the CUDA Baseline and the OpenMP Baseline. The explanation is straightforward: both the RTX 3050 Ti (20 streaming multiprocessors) and the i7-12700H (20 logical CPU cores) have roughly the same degree of parallelism. With identical memory limits, identical throughput is the logical outcome. Beyond ~6.9k atoms, the single-thread Cell List's $O(n)$ scaling beats every $O(n^2)$ variant, underscoring the value of algorithmic improvements.

Overall, this project was extremely successful. We learned a great deal and were very happy with the final results. We also greatly enjoyed EC 527 and wish Prof. Herbordt a wonderful summer.

References

[1] SymbiosOpenMM, Stanford University. Introduction to Molecular Dynamics. (May 9, 2014). Accessed: Apr. 3, 2025. [Online Video]. Available: https://www.youtube.com/watch?v=_TiQYNWJwYg

[2] A. Nakano (2024). University of Southern California CSCI 596 Scientific Computing and Visualization: Molecular Dynamics Basics [Lecture notes]. Available: https://aiichironakano.github.io/cs596/01MD.pdf

[3] A. Nakano (2024). University of Southern California CSCI 596 Scientific Computing and Visualization: Linked-List Cell Molecular Dynamics [Lecture notes]. Available: https://aiichironakano.github.io/cs596/01-1LinkedListCell.pdf

Github

https://github.com/alexrmelnick/Molecular-Dynamics-Simulation