UNIVERSITY OF COPENHAGEN

MSc in Computer Science

Scientific Simulation in Futhark

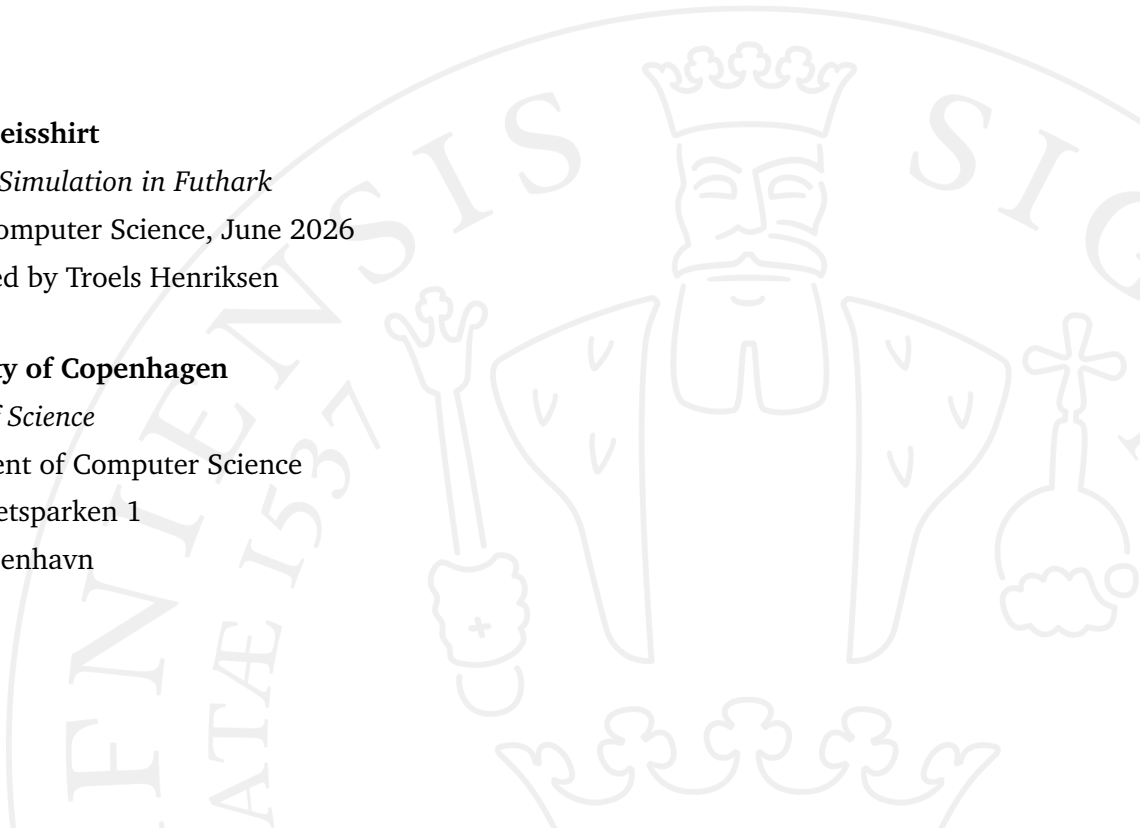
Svante Geisshirt

Supervised by Troels Henriksen

June 2026

Svante Geisshirt
Scientific Simulation in Futhark
MSc in Computer Science, June 2026
Supervised by Troels Henriksen

University of Copenhagen
Faculty of Science
Department of Computer Science
Universitetsparken 1
2100 København



Abstract

Scientific simulation allows researchers to explore system behavior without the need for large-scale experiments. Traditionally, such simulations have been implemented in C or Fortran, which offer great control and therefore performance but come with the price of a high burden on the programmer as they must manage memory and threads manually. This thesis investigates how Futhark, a high-level data-parallel functional language, compares to traditional languages in both runtime performance and programmer experience.

A molecular dynamics simulator governed by Newton's laws of motion and the Lennard-Jones potential was the basis of the project and was implemented in six languages with varying abstraction levels: C, OpenMP, CUDA, NumPy, JAX, and Futhark. Using Futhark's compilation flexibility, it was compiled to three backends: sequential C, multicore CPU, and parallel CUDA.

The implementations were benchmarked using end-to-end runtime at different system sizes, and it was found that while not faster, Futhark is competitive with its handwritten counterparts. Although faster, it was found that the experience of writing and maintaining three separate code bases (C, OpenMP, and CUDA) was a significant burden compared to a single high-level code base in Futhark.

Contents

1 Introduction

Scientific simulation enables researchers to explore system behavior and test their hypotheses numerically without waiting or having to pay for large experiments. Particle simulations governed by Newton's laws of motion and the Lennard-Jones (LJ) potential will be the basis of this project. The LJ potential models interactions between neutral spherical particles, with a short-range repulsive and a long-range attractive force (Lenhard *et al.*, 2024). Evaluating the LJ naïvely requires $\mathcal{O}(N^2)$ computations, where N is the particle count, as every other particle interacts. Some optimizations exploited in this project will bring it down to $\mathcal{O}(N)$.

Traditionally, high-performance scientific software has been written in low-level imperative languages such as C, Fortran, and CUDA, which allow for high control of memory and thread assignment, leading to well-optimized code (Allen and Tildesley, 1998). Such code demands hardware understanding and is error-prone. Futhark, a high-level, data-parallel, functional language, makes it easier to write parallel programs (Elsman *et al.*, 2018), making it well-suited for computationally intensive, data-parallel algorithms such as particle simulations. Futhark saves the programmer the burden of manual memory management, and the compiler takes responsibility for generating efficient parallel code as a series of bulk array transformations. With support for compilation to multiple backends, a single Futhark codebase can be run sequentially, with multiple cores, and in parallel on an Nvidia GPU. This thesis investigates how Futhark compares to traditional languages for scientific simulation in both runtime performance and programmer experience.

This thesis is structured as follows: Chapter 2 introduces the scientific background, Chapter 3 discusses hardware-aware optimization strategies, Chapter 4 describes the implementations, Chapters 5 and 6 present the performance modeling and results, Chapter 7 reflects on the qualitative experience of programming across the abstraction levels, and finally, Chapter 8 concludes the project. The code for the project is publicly available on GitHub at `https://tinyurl.com/3xfvavtp`.

2 Scientific Simulation

Scientific simulation is the imitation of real-world physical or chemical systems using a mathematical model, executed on a computer, which is typically expensive or impractical to observe or measure (Winsberg, 2022). This chapter aims to provide the reader with some background on what scientific simulation is and how it has been used in this project. Molecular dynamics (MD) simulations are a computational method for modeling how a system of particles behaves and interacts over time (Binder *et al.*, 2004). The particles can represent an atom or a molecule, however, their motions are governed by Newton's laws of motion. The goal of an MD simulation is thus to calculate the macroscopic thermodynamic (physical or chemical) properties of the system, e.g., heat capacity, temperature, and pressure. This is done by computing the positions and velocities of a system as it interacts over time, given a set of initial positions and velocities.

2.1 Mathematical Model

The system in this project is a three-dimensional cube of length L containing N particles. Every particle has a position, $\mathbf{x}_i = (x_i, y_i, z_i)$, and a velocity, $\mathbf{v}_i = (v_{xi}, v_{yi}, v_{zi})$. To eliminate boundary effects, *periodic* boundary conditions are applied, allowing a particle to exit the cube on one side only to re-enter on the opposite side. The distance between the particles is then computed using the *minimum image convention* (Hloucha and Deiters, 1998), which gives the shortest periodic distance. For instance, if the cube has length $L = 10$, and a particle is at $(0, 0, 0)$ and another is at $(0, 0, 9)$, the distance between the two particles is one. The interaction between the particles, in this project, is modeled using the *Lennard-Jones potential* to obtain the forces acting on them.

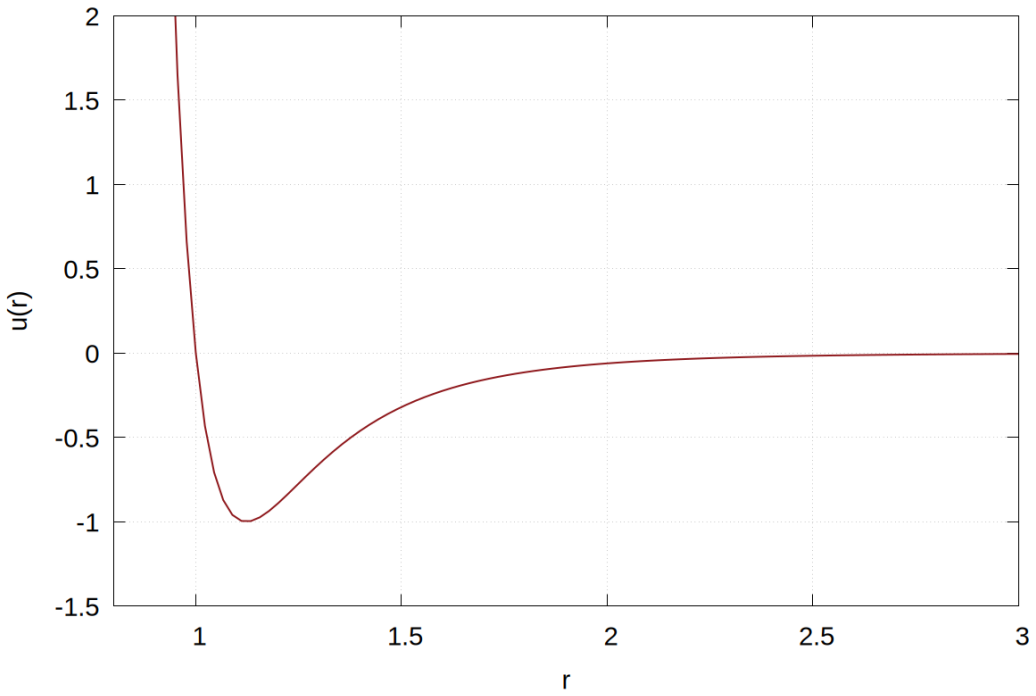


Figure 2.1.: Graph of the Lennard-Jones 12-6 potential between particles $u(r)$ as a function of the distance r as seen in equation 2.1.

2.1.1 Lennard-Jones Potential

The Lennard-Jones (LJ) potential can be used to model the interactions between spherical, neutral particles such as argon and methane. The potential describes two effects, namely a repulsive force and an attractive force. In the general form, the repulsive potential is $1/r^n$ and the attractive potential is $-1/r^m$ where r is the distance between the two particles. $n = 12$ and $m = 6$ are widely used as parameters and are referred to as the *Lennard-Jones 12-6 potential* (Tee *et al.*, 1966). $m = 6$ is chosen to match the London dispersion force[1], and $n = 12$ is a computational convenience. If we have already computed r^6, then squaring it gives us r^{12}, which is a good approximation to the exponential repulsion. Using the 12-6 potential, we see the repulsive term dominates at a short distance, and the attractive term dominates at a long distance. The LJ potential combines the two effects by summation into a single expression:

$$u(r) = 4\varepsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^{6} \right] \tag{2.1}$$

[1] A temporary attraction force that occurs when two electrons occupy the same position.

where σ is the distance at which the potential is zero and ε is the depth of the *potential well*. The function is plotted and shown in Figure 2.1, and the well is the minima at $r = 2^{1/6}$. At the minima, the particles are at an equilibrium, meaning they are not repelling nor attracting each other.

In molecular dynamics simulations, the force between particles is required rather than the potential itself. The force is the negative gradient of the potential:

$$F(r) = -\frac{\mathrm{d}u}{\mathrm{d}r} \tag{2.2a}$$

$$= -\frac{\mathrm{d}}{\mathrm{d}r}\left[4\varepsilon\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right] \tag{2.2b}$$

$$= \frac{24\varepsilon}{r}\left[2\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right] \tag{2.2c}$$

Using equation 2.2c between all pairs of particles, the force acting on each particle can be found. In Figure 2.1, we notice that the function, $u(r)$, rapidly decreases with distance and has an asymptote at zero. We can use this to our advantage by assuming that any distance $r \geq 2.5$ is practically zero, and hence we only need to compute the force between particles that are close to each other.

2.1.2 Newton's Laws of Motion

MD simulations are based on classical mechanics, where the motions of particles are governed by Newton's laws of motion, particularly the second law (Allen and Tildesley, 1998). As a reminder, it states that the net force acting on a particle is related to its mass and acceleration as:

$$F = m \cdot a \tag{2.3}$$

where F is the force, m is the mass, and a is the acceleration. In the simulation, the force arises from pairwise interactions between particles as described by the LJ force.

Newton's third law of motion is also used in this project, and it states that the force between pairs is equal and opposite, meaning the force from particle i applied on j is equal and opposite to the force of j applied on i, i.e.:

$$F_{ij} = -F_{ji} \tag{2.4}$$

This symmetry can be exploited to compute the force on one particle and reuse it for the other.

2.2 Numerical Time Integration

Newton's second law of motion governs the motion of each particle in the system. For a system of N particles, the total force on particle i is the sum of the pairwise interactions:

$$\mathbf{F}_i(t) = \sum_{j \neq i} \mathbf{F}_{ij}(t). \tag{2.5}$$

where $\mathbf{F}_{ij}$ represents the force between particle i and j as defined by the Lennard-Jones potential (Section 2.1.1). Each pairwise interaction depends on the positions of the two interacting particles, and thus the total force depends on the positions of all the particles in the system. Applying Newton's second law, where the acceleration is given by the second derivative of the particle's position, we get a system of second-order ordinary differential equations:

$$\frac{\mathrm{d}^2\mathbf{x}_i(t)}{\mathrm{d}t^2} = \frac{1}{m} \sum_{j \neq i} \mathbf{F}_{ij} \left(\mathbf{x}_1(t), \mathbf{x}_2(t), \ldots, \mathbf{x}_N(t) \right) \tag{2.6}$$

where $\mathbf{x}_i$ is the position of particle i at time t. This system is nonlinear and strongly coupled, since the acceleration of each particle depends on the positions of all other particles. For systems with three or more interacting particles, no analytical solution exists and is known as the N-body problem. Consequently, the time evolution of the system must be approximated numerically by discretizing time into small steps, Δt, and computing the positions and velocities at each time step, iteratively. The method used for advancing the system from t to $t + \Delta t$ in this project is the *Velocity Verlet Algorithm*.

2.2.1 The Velocity Verlet Algorithm

The Velocity Verlet algorithm is derived from a Taylor expansion of the particle's motion. It provides a scheme for updating the particles' positions and velocities for each time step while maintaining energy conservation (Omelyan *et al.*, 2002). To obtain an expression for the position at $t + \Delta t$, we make use of a Taylor expansion of the position function $\mathbf{x}_i(t)$ around t. The Taylor expansion around t is given by:

$$f(t + \Delta t) = \sum_{n=0}^{\infty} \frac{f^{(n)}(t)}{n!} (\Delta t)^n \tag{2.7}$$

And we know the first derivative of position with respect to time is velocity and the second derivative is acceleration, i.e.:

$$\frac{\mathrm{d}\mathbf{x}_i(t)}{\mathrm{d}t} = \mathbf{v}_i(t), \quad \frac{\mathrm{d}^2\mathbf{x}_i(t)}{\mathrm{d}t^2} = \mathbf{a}_i(t) \tag{2.8}$$

where $\mathbf{v}_i(t)$ is the velocity of particle i and $\mathbf{a}_i(t)$ is the acceleration at time t. Hence, expanding the position of a particle in a Taylor series around t yields:

$$\mathbf{x}_i(t + \Delta t) = \mathbf{x}_i(t) + \Delta t \mathbf{v}_i(t) + \frac{1}{2}\Delta t^2 \mathbf{a}_i(t) + \mathcal{O}\left(\Delta t^3\right) \tag{2.9}$$

where the error term, $\mathcal{O}\left(\Delta t^3\right)$, will be ignored since it is negligible. Notice that the expression depends on the current position, velocity, and acceleration. Similarly, a Taylor expansion of the velocity yields:

$$\mathbf{v}_i(t + \Delta t) = \mathbf{v}_i(t) + \Delta t \mathbf{a}_i(t) + \mathcal{O}\left(\Delta t^2\right) \tag{2.10}$$

and, again, the error term, $\mathcal{O}\left(\Delta t^2\right)$, will be ignored since it is negligible. Using equations 2.9 and 2.10 directly is impractical since the acceleration at time $t + \Delta t$ depends on the updated positions, which at the current step are unknown. To conquer this problem, the Velocity Verlet method performs *half steps*. Here, the velocities are evaluated at intermediate half steps used to compute the positions, which in turn are used to compute the new acceleration and complete the final velocity update.

First, the forces are computed at time $t = 0$ before the time loop even begins. When the loop begins, the velocity is updated using the current acceleration as:

$$\mathbf{v}_i\left(t + \frac{1}{2}\Delta t\right) = \mathbf{v}_i(t) + \tfrac{1}{2}\Delta t\ \mathbf{a}_i(t) \tag{2.11}$$

Then the positions are updated using the intermediate velocity:

$$\mathbf{x}_i(t + \Delta t) = \mathbf{x}_i(t) + \Delta t\ \mathbf{v}_i\left(t + \frac{1}{2}\Delta t\right) \tag{2.12}$$

The forces are then recomputed based on the updated positions and velocities, resulting in the new acceleration $\mathbf{a}_i(t + \Delta t)$. Finally, the velocity is updated with the second step, which is computed using the updated forces:

$$\mathbf{v}_i(t + \Delta t) = \mathbf{v}_i\left(t + \frac{1}{2}\Delta t\right) + \frac{1}{2}\Delta t\ \mathbf{a}_i(t + \Delta t) \tag{2.13}$$

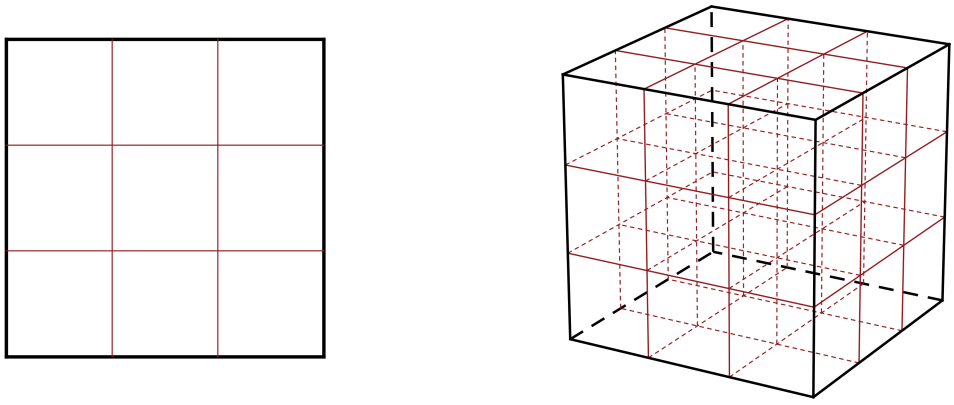
With that, the scheme uses equations 2.11, 2.12, and 2.13 to update the positions and velocities for each timestep. Although the higher-order terms are ignored in this project, the Velocity Verlet algorithm does exhibit a numerical error that accumulates globally with the time step size. By picking a small Δt, however, the accuracy and stability of the simulation will be improved.

2.3 Efficient Force Computation

To compute the forces between the particles, there are optimizations to consider. The first is to introduce a *cutoff* distance as mentioned in Section 2.1.1. Here we can choose some r_{cutoff} such that the practical potential is close to zero. Choosing $\sigma = \varepsilon = 1$ we get $u(r_{\text{cutoff}} = 2.5) = -0.01$. This allows us to only compute the forces between the particles within the cutoff, greatly reducing the amount of interaction, assuming the particles are uniformly distributed. This is a reasonable approximation for homogeneous systems, e.g., gases.

2.3.1 Naïve Pairwise Approach

The simplest way to compute the forces between the particles is to evaluate the force between all pairs. This approach, however, requires $\frac{N(N-1)}{2}$ interactions for N particles resulting in a computational complexity of $\mathcal{O}\left(N^2\right)$. Even with the *cutoff* optimization described above, this approach still has to consider



(a) Two-dimensional example with a neighborhood of $3 \times 3 = 9$ cells.

(b) Three-dimensional example with a neighborhood of $3 \times 3 \times 3 = 27$ cells.

Figure 2.2.: Illustration of the cell decomposition to exploit the short-range nature of the LJ potential for efficient force evaluation. Figure (a) is a simplified two-dimensional example, and Figure (b) is a more convoluted example in three dimensions, which resembles the actual structure.

all pairs before disregarding the ones further away than the cutoff, and the complexity remains quadratic, i.e., only compute the cheaper distance and not the expensive force calculation. This approach is quite easy to implement, but becomes impractical even at a relatively small N.

2.3.2 Cell List Approach

A different approach is to exploit the short-range nature of the LJ potential further by constructing a grid of smaller cells within the box. An illustration of the decomposition is shown in Figure 2.2. These cells will have a side length greater than or equal to the cutoff radius. Each particle is then assigned to the cell it is currently in based on its position. This allows us to only check the neighboring cells since we know for certain that, since the cell length is chosen to be at least the cutoff radius, any interacting particle pairs must be in the same cell or neighboring cells. In a three-dimensional system, there are $3^3 = 27$ (including the current) cells to consider.

This approach reduces the *candidate* interactions per particle from $\mathcal{O}(N)$ to $\mathcal{O}(1)$ if we assume uniform particle distribution. The overall computational

complexity is then $\mathcal{O}(N)$, and by applying Newton's third law, the number of cells to check can be reduced by a factor of two.

3 Hardware-Aware Optimizations

While asymptotic complexity indicates scalability, it does not capture a program's practical runtime. In many cases, significant performance improvements can be achieved through implementation choices without changing the complexity of the algorithm/program. In MD simulations, most of the computational cost stems from evaluating forces on a large number of particles. This results in efficiency being dependent on how the data is stored in memory and how well the program fits the specification of the underlying hardware.

In this project, the simulators are designed for both CPUs and GPUs, which have different considerations when programming them efficiently. The physical hardware differs in how they execute instructions, perform parallel code, and, perhaps most importantly, access memory. This chapter aims to provide the reader with what to consider when optimizing computational performance without changing the overall complexity, i.e., obtaining a better performance with the same algorithm.

3.1 Memory Hierarchy and Locality

Most modern architectures use a memory hierarchy to close the gap between fast processors and slow main memory (RAM). This hierarchy relies on small but fast *caches* (L1, L2, and L3) which store frequently accessed data. The latency of fetching data from RAM is often orders of magnitude slower than a cache hit, which is why software performance is frequently limited by memory bandwidth rather than raw arithmetic throughput. This is known as the *memory wall* (McKee, 2004).

An important design consideration is therefore how data is stored in memory and how we access it. In this project, the data consists of particle positions

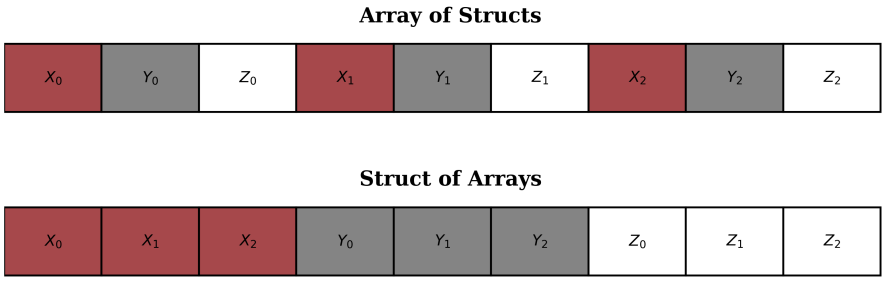


Figure 3.1.: Illustration of memory layouts: Array of Structs (AoS) vs. Struct of Arrays (SoA).

and velocities, and each of these has components in the x, y, and z directions. There are two main ways to organize this in memory:

- **Array of Structures (AoS):** Store the individual particles as structs, e.g., `struct {x, y, z, vx, vy, vz} particles[N]`. While this representation is intuitive and provides good temporal locality for a single particle, it leads to inefficient cache usage in the simulators. This stems from the force evaluation, which only requires the positions and not the velocities. We end up filling the cache with data we neither want nor need (the velocities), resulting in the small cache being full sooner, requiring more expensive fetches while also paying for the fetches of the velocities we don't need.

- **Structure of Arrays (SoA):** Alternatively, each component can be stored in its own array, `double x[N], y[N], z[N], vx[N], vy[N], vz[N]`. This layout provides good spatial locality during the force evaluation since only the positions are fetched into the cache, allowing the hardware prefetcher to stream contiguous memory more effectively. This layout is also useful for SIMD instructions.

The SoA layout is therefore a better choice for this project, as the most computationally intensive parts of the simulation are the force evaluation, which is performed on particle positions. By storing these values contiguously, the memory accesses are more predictable and efficient, reducing cache misses and improving overall performance. The difference between SoA and AoS is illustrated in Figure 3.1.

While the SoA layout allows for efficient access to the individual components, the MD simulator's performance follows the spatial locality of the particles. Due to the nature of the LJ potential's short range, a particle only interacts with particles in close proximity. Suppose the neighbors of a particle are scattered in memory, we will have to chase around for the coordinates we need, resulting in frequent cache misses, which are expensive. To avoid this, the simulation domain is divided into a grid of cube-shaped cells with a side length of the cutoff radius, r_c. The particles are assigned to a cell depending on their spatial coordinates, and by sorting the position arrays with respect to their cell index, we ensure that physical particle neighbors become neighbors in the address space in the memory, which leads to an improved temporal locality. Now, when we are evaluating the forces for particle i in some cell, the data for all neighboring particles in that cell and its neighboring cells are loaded into the cache. These neighboring cells will also interact with particle $i + 1$, and now the data is already in the cache, reducing the need to fetch from the slow main memory, to only when we consider a new cell.

3.2 Branching and Arithmetic Cost

Besides optimizing the memory access, one can optimize the flow of executed instructions and the arithmetic operators. In this project, the forces between interacting particles are evaluated many times, meaning small overheads can lead to significant bottlenecks.

Mathematical operations vary in how many instructions they require. Addition and multiplication are handled in a single clock cycle, while other operations are more expensive. For instance, general purpose divide routine may take up to 80 cycles (Magenheimer *et al.*, 1987). The LJ force, as seen in equation 2.2c, involves operations such as distances, powers, and divisions, which are all expensive operations. Thus, a naïve implementation is quite slow but can be optimized. Here, the distance evaluations require division and square roots, which are expensive operations. In this project, the square root is mitigated by keeping all distances squared and comparing with r^2_{cutoff}.

Furthermore, exponents in the LJ force would naïvely depend on a *power* function, which is slow in most languages and all of which are used in this

project. However, since the force only requires integer powers, the r^{-6} term can be computed as first assigning $a = r^{-2}$ and then computing $a \cdot a \cdot a$, which saves divisions and exponents and brings it down to a single division and some multiplication. Applying this again, we can compute r^{-12} from $r^{-6} \cdot r^{-6}$.

3.3 Parallelism

While memory layout, branching, and arithmetic efficiency optimizations can potentially maximize the efficiency of individual instructions, parallelism can optimize the entire program's throughput. The nature of the MD simulator in this project offers opportunities for concurrency since the force interactions are entirely *independent*. This means the potential and resulting force between a pair of particles, (i, j), is dependent solely on the coordinates of particles i and j. This allows the simulation to go from a series of steps to a large collection of simultaneous operations.

This project considers both sequential and parallel execution models, and in the sequential execution model, the simulator iterates through pairs one by one. Since no pair calculation depends on the result of another, the workload can be decomposed across any number of processing units on both a multicore CPU and a parallel GPU. However, parallel the force evaluation is, the forces must still be aggregated to a total force vector for each particle, which isn't parallel. Consider the forces between particles $(1, 2)$ and $(1, 3)$, and here we see they both contribute to particle one. This results in a write conflict or a race between the threads, since both pairs need to update particle one.

The race condition exposes a fault in the implementation technique rather than the underlying physics informing the simulation. Luckily, several solutions exist within a variety of programming paradigms, including the one considered in this project:

- **Imperative paradigm:** In languages like `C`, the shared state is focused through synchronization, which typically includes *atomics* that can lock a specific memory address to perform an update.

- **Functional paradigm:** This approach treats the MD simulation like a series of data transformations. Instead of updating a global array with atomics or locks, a functional approach will employ the *map-reduce* strategy. Here, the forces between the particle pairs are evaluated as a collection of independent and parallel calculations (the map part), and then combined using a conflict-free operation (the reduce part). The reduction step is conflict-free because each thread produces its own result as a partial force, and the whole thing is combined like a tree structure, ensuring no threads ever conflict.

4 Implementation of the Molecular Dynamics Simulator

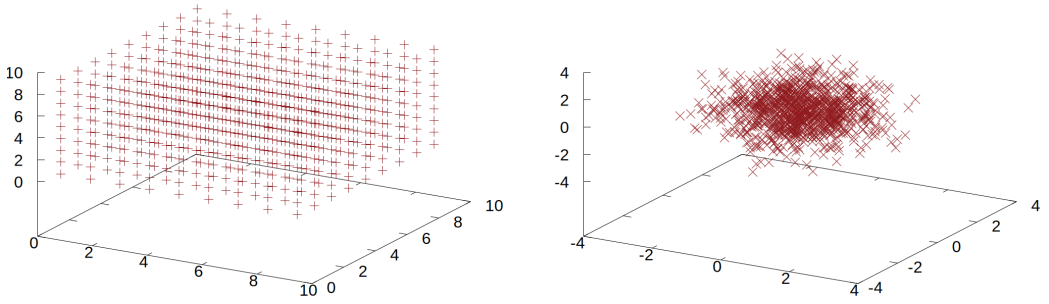
This chapter presents the implementation details of the MD simulator developed in this project. Each simulator, which includes C, OpenMP, CUDA, NumPy, JAX, and Futhark, implements the same algorithm and is based on the same physical and mathematical model, however, they differ in programming language paradigms and optimizations. These languages were chosen to provide a range of abstraction levels enabling the comparison with Futhark for both performance and programmer experience.

The chapter first describes how the input is represented across the implementations and how it is generated. Then the general algorithm is outlined before language-specific implementation details are presented.

4.1 Generating the Input

The particles in the system must have initial values, hence the need for an input generator. The input is generated via a Python script, which constructs the particle configuration for all N particles and the global simulation parameters. The goal of the generator is to produce physically reasonable and reproducible initial values so that the simulation can be properly validated and its performance benchmarked across the implementations.

For each simulator to read the generated input, the generator writes to a plain-text file. Here, the first four lines are global parameters for the system: the number of particles N, the simulation box length L, the number of steps, and the step length Δt. In this project, the simulation box length is not kept constant but is instead a function of the system's density, which is then kept



(a) Initial uniformly distributed particle positions.

(b) Initial normally distributed particle velocities.

Figure 4.1.: Initial conditions of a 1000 particle system: (a) positions sampled from a uniform distribution, and (b) velocities sampled from a normal distribution.

constant throughout the simulation. With the density, ρ, the box length L can be found with:

$$L = \left(\frac{N}{\rho}\right)^{\frac{1}{3}} \tag{4.1}$$

One can pick a density to simulate, e.g., a gas or a liquid, and the default value chosen for this project is $\rho = 0.8$.

Following the header, each particle is described in the plain text initialization file on its own line with six floating-point values: first the spatial coordinates (x, y, z), then the three velocity components (v_x, v_y, v_z). This makes it easy to read from most languages and all that are used in this project, using formatted input functions to read the file sequentially. An example of the header and the first 15 particles in an input file is shown in Appendix A. If there are $N = 1000$ particles, the plain input text file will have 1004 lines, four for the header and 1000 for the particles. The initial particle positions are placed in a uniform grid within the simulation cube. The spacing between particles is given by $L/\sqrt[3]{N}$ in each dimension. This ensures that the entire simulation box is uniformly filled, as shown in Listing 4.1a. The initial velocity components are sampled independently from normal distributions with zero mean, resulting in particle speeds that follow a Maxwell–Boltzmann distribution, which characterizes a thermal system at equilibrium (Huang, 1987). This is shown in Listing 4.1b.

4.2 The General Algorithm

All the simulator implementations follow the same high-level algorithm to ensure consistency, which is important for validating and benchmarking them. For this reason, all variables use double-precision floating-point numbers across the implementation as well. This section outlines the algorithm's core phases, and the language-specific details of those phases are described in the subsequent sections.

4.2.1 Initialization Phase

Before starting the simulation loop, each implementation must read and parse the plain text input file, which contains the global parameters as described in Section 4.1. Furthermore, it sets the LJ specific parameters, $\varepsilon = \sigma = 1$ and $r_{\text{cutoff}} = 2.5$. It then reads all the particles into position and velocity arrays, and particles are assigned to cells based on their spatial coordinates and sorted accordingly. With the cells sorted, an initial call to compute the force is performed at $t = 0$ to provide the needed acceleration for the first Velocity Verlet half step.

4.2.2 Advancing Time using the Velocity Verlet Algorithm

Once the initialization is done, the main simulation loop repeats a user-defined number of times. Each iteration follows the following scheme to advance time with the mathematical model described in section 2.2.1:

- **First velocity half-step.** Update the velocities to the midpoint using the current acceleration from the force as equation 2.11.

- **Position update.** Update the positions of the particles with their intermediate velocities as shown in equation 2.12.

- **Apply boundary conditions.** Using the periodic boundary condition with a modulo operation to ensure the particle leaving the box enters the opposite side.

- **Re-sort cells.** Re-assign particles to spatial cells and sort them to maintain good spatial locality in the memory.

- **Force computation.** Recompute the force using the updated positions and velocities.

- **Second velocity half-step.** Complete the velocity update as shown in equation 2.13.

A high-level pseudocode is shown in Algorithm 1. Additionally, at 5000 steps, the current particle state is written to a file as the system's initial state for validation, as described in Section 5.1. This point is chosen to allow the system to move from its artificial input grid to a natural equilibrium.

4.2.3 Termination

Once the loop has completed the user-defined number of iterations, the final state is written to another file, which is also used for validation along with the initial state.

4.3 Imperative Implementations

The imperative implementations in this project are written in C and further extended with OpenMP, which is an API for parallel programming on shared-memory systems in C (Chandra, 2001). Furthermore, a GPU parallel simulator was implemented in CUDA. All three implementations are very similar, and OpenMP merely needs some directives on top of the C code and small modifications to parallelize the code in the C implementation. CUDA, however, is structured using individual kernels, which are controlled from the CPU but executed directly on the GPU. In the following sections, the sequential C code

Algorithm 1 Main Simulation Loop

1: **procedure** SIMULATE($N, L, steps, dt, rc$)
2: $v, x \leftarrow$ READINITIALSTATE($input.txt$)
3: $cells \leftarrow$ CELLINDEX(x, L)
4: $f \leftarrow$ COMPUTEFORCES($x, L, rc, cells$)
5: **for** $step = 1$ to $steps$ **do**
6: **if** $step == 5000$ **then**
7: WRITESTATE($intial.txt, x, v$)
8: **end if**
▷ First Velocity half-step.
9: **for** $i = 1$ to N **do**
10: $v[i] \leftarrow v[i] + \frac{1}{2} \cdot dt \cdot f[i]$
11: **end for**
▷ Position update and apply boundary conditions.
12: **for** $i = 1$ to N **do**
13: $x[i] \leftarrow x[i] + dt \cdot v[i]$
14: $x[i] \leftarrow$ PERIODICBOUNDARY($x[i], L$)
15: **end for**
▷ Sort particles by cell.
16: $cells \leftarrow$ SORTBYCELL($x, v, cells$)
▷ Re-compute forces.
17: $f \leftarrow$ COMPUTEFORCES($x, L, rc, cells$)
▷ Second Velocity half-step.
18: **for** $i = 1$ to N **do**
19: $v[i] \leftarrow v[i] + \frac{1}{2} \cdot dt \cdot f[i]$
20: **end for**
21: **end for**
22: WRITESTATE($final.txt, x, v$)
23: **end procedure**

is described, how it was converted to a parallel OpenMP version, and finally, the CUDA implementation.

4.3.1 Sequential C

The C language requires the programmer to take full control of the memory management, including layout, allocation, and deallocation. As described in Section 3.1, this project uses SoA to improve locality, so each coordinate component is dynamically allocated in its own array using the `malloc` and `calloc` functions and released with `free`.

After reading the global data from the input and constructing the particle data corresponding to lines 2 and 3 of Algorithm 1, the algorithm calls `cellIndex`. In the C implementation, this call assigns particles to a spatial cell based on their coordinates. The number of cells per dimension is determined by the integer division of the box length by the cutoff radius. This ensures each cell has a side length greater than or equal to the cutoff, which in turn guarantees that if two particles are close enough to interact, they are either in the same cell or in one of the 26 neighboring cells. The particles are then assigned to cells based on their coordinates by dividing each coordinate by the cell size, and the floor of this number is the cell index. To adhere to the boundary conditions, the indices are wrapped so that they remain in valid bounds, and the three-dimensional cell index is flattened to a one-dimensional array. This is shown in Listing 1, and here the modulo operations implement the boundary conditions, and the flattened one-dimensional index is assigned to `particle_cells`. With

```
1  for (int i = 0; i < N; i++) {
2      int cx = (int)floor(x[i] / cell_size);
3      int cy = (int)floor(y[i] / cell_size);
4      int cz = (int)floor(z[i] / cell_size);
5
6      cx = (cx % nc + nc) % nc;
7      cy = (cy % nc + nc) % nc;
8      cz = (cz % nc + nc) % nc;
9
10     particle_cells[i] = cx + nc * (cy + nc * cz);
11 }
```

Listing 1: Assigning particles to cells, applying boundary conditions, and flattening cell indices to be one-dimensional.

the particles, now assigned to cells, they are sorted according to their cell index to obtain better spatial locality. This is `SortByCell` in Algorithm 1, and the C implementation achieves it in three steps:

1. The number of particles in each cell is counted.

2. The starting offsets in each cell are found using a prefix sum.

3. The particles are reordered into a new array according to their cell indices using the offsets, as shown in Listing 2.

```
for (int i = 0; i < N; i++) {
    int c = particle_cells[i];
    int index = cell_offsets[c] + temp_counts[c];
    sorted_particles[index] = i;
    temp_counts[c]++;
}
```

Listing 2: Reordering particles belonging to the same cell to be contiguous in memory, to improve spatial locality.

The `ComputeForces` step in Algorithm 1 is implemented in two stages. First, a pair list of all pairs of particles in the same and 26 neighboring cells is constructed by iterating over all cells. The corresponding particle is retrieved using the sorted particle array using the offsets and counts. To avoid double-counting, interactions within the same cell are only considered for particle pairs where the second index is greater than the first, and interactions between cells are only processed when the neighbor cell index is greater than or equal to the current cell index.

For each pair, the distance is computed using the minimum image convention, and if this distance is less than the cutoff, the pair is added to the pair list. With the pair list, the force is computed by iterating over it and applying the LJ force and updating both particles using Newton's third law, i.e., $F_{ij} = -F_{ji}$.

To save some computation, common tricks are used, e.g., keeping distances squared for comparison to save a square root and exploiting the fact that $x^6 \cdot x^6 = x^{12}$ to save additional exponent computations where it is applicable.

4.3.2 OpenMP Parallelization

To parallelize the C code, it is extended with OpenMP, which allows the code to run on multiple cores using shared memory on CPUs with two or more cores, i.e, multicore CPUs. The overall implementation remains the same as the C version described above, and this implementation focuses on distributing work to threads while still achieving the correct result.

Fortunately, many parts of the C implementation can be parallelized directly using directives, including the loops that update velocities and positions and apply boundary conditions. These loops are inherently parallel and can be parallelized with `#pragma omp for`, which distributes iterations across available threads without synchronization, as shown in Listing 3. The cell assignment is

```
#pragma omp parallel for
for (int i = 0; i < N; i++) {
    vx[i] += 0.5 * dt * fx[i];
    vy[i] += 0.5 * dt * fy[i];
    vz[i] += 0.5 * dt * fz[i];
}
```

Listing 3: Pragma directive for distributing loop iterations across available threads.

partially independent, which requires a bit more work to optimize. First, the cell index computation is fully independent and is parallelized using a loop directive as above. The count of particles and computing the prefix sum must remain sequential since they depend on shared data.

In the force computation, each particle interaction is independent, however, naïvely updating them would cause race conditions since multiple threads may attempt to update the force of the same particle simultaneously. To counter this, the implementation uses a *thread local accumulation strategy* where each thread maintains its own local force array, accumulating force contributions. This is shown in Appendix B. This now avoids all write conflicts and is safe to parallelize with a pragma directive. Once all threads have completed, which can be guaranteed with `#pragma omp barrier`, their share of force computations in the local forces is reduced to combine the forces into a global force array. The reduction is a parallel loop over the particles where the contributions from all the threads are summed.

The construction of the pair list is also parallelized using the OpenMP loop collapsing and dynamic scheduling feature. Each thread constructs its own local list of interacting pairs, which are merged, using an atomic operation, into a global pair list.

4.3.3 CUDA

Parallelism can also be exploited on the GPUs, and CUDA allows general-purpose programming to run Nvidia components through explicit parallel programming (NVIDIA, 2026a). To further accelerate the simulator, a CUDA implementation was developed. Unlike the parallelism in the OpenMP implementation, which relies on a small number of threads, CUDA provides thousands of lightweight threads, and since the MD program is largely independent (in each particle interaction), it will naturally increase the program throughput.

The CUDA implementation resembles the previous implementation, but with decomposed GPU kernels, where a thread is explicitly in charge of a small task, e.g, updating a particle position. The kernels use a thread per particle (or pair) and use the global thread index to determine which one. This scheme is shown in Listing 4.

```
1 int i = blockIdx.x * blockDim.x + threadIdx.x;
2 if (i >= N) return;
```

Listing 4: CUDA thread indexing, to particles (or particle pairs) with bounds checking.

This allows the work to scale with the given GPU and its available cores. First, it must build the pair list, which is done with the following GPU kernels and pipeline, which are called from a CPU function:

Assigning particles to cells. Like the CPU implementation written in C, the CUDA implementation uses a cell list to reduce the computational complexity from $\mathcal{O}(N^2)$ down to $\mathcal{O}(N)$. Here, the particles are assigned to a spatial cell, which is done fully in parallel on the GPU. A single thread handles one particle independently and assigns it to the appropriate cell by its three-dimensional coordinate to a flat one-dimensional cell array.

Sorting. The particles are then sorted using the Thrust library (NVIDIA, 2026b) by their cell indices, as shown in Listing 5 (for radix sort, this will run in $\mathcal{O}(k \cdot N)$ time where k is the number of its processed bits per key). This, like in the CPU implementations, groups the particles belonging to the same cell, improving the spatial locality since the particles can then be traversed through contiguous access. First, the memory pointers, `d_cell_id`,

```
thrust::device_ptr<int> t_cell(d_cell_id);
thrust::device_ptr<int> t_pid(d_sorted_pid);
thrust::sort_by_key(thrust::device, t_cell, t_cell + N, t_pid);
```

Listing 5: Sorting particles using the Thrust library.

which store the cell index for each particle, and `d_sorted_pid`, which contains the corresponding particle ID, are wrapped in Thrust device pointers, which allows them to be used in the Thrust parallel primitives. The call to `thrust::sort_by_key` then sorts the particles by cell indices (`t_cell` is the key, and `t_pid` is the reordering).

Computing cell offsets. After particles have been sorted, the start and end offsets for each cell are computed. Here, each thread compares itself with the neighboring entries. If the cell index of the previous particle differs, we know it must be the start, and similarly, if the cell index of the next particle differs, it must be the end.

Building the pair list. Each thread iterates over the thread-indexed particle's $3^3 = 27$ neighbor cells and checks if the particles in those cells are within the cutoff radius. If a particle is within the radius, it is inserted into a global pair list using an atomic operation, as shown in Listing 6. This allows the pair list construction to be fully parallel while still avoiding duplicate pair evaluations. While atomic operations avoid race conditions, they can greatly bottleneck the runtime if many threads attempt to update the same address simultaneously.

To further parallelize the implementation, the following GPU kernels also exist:

Zeroing forces. Before evaluating the forces, the force arrays must be reset to avoid the force accumulating over time. Since the resetting has to be done completely across all particles before any new accumulation begins, this must

```
double r2 = ddx*ddx + ddy*ddy + ddz*ddz;
if (r2 < rcut2) {
    int k = atomicAdd(npairs_out, 1);
    if (k < max_pairs) {
        ipair[k] = i;
        jpair[k] = j;
    }
}
```

Listing 6: Inserting a particle into the global pair list if the particle is within the cutoff radius.

be its own kernel, as CUDA doesn't allow kernels to synchronize globally across thread blocks.

Evaluating the LJ force. Each thread is assigned to an interacting particle pair in the pair list, using the thread indexing shown in Listing 4, and for each pair, the minimum image convention is applied to obtain their distance from each other. With the distance, the LJ force is computed, and to further reduce runtime, Newton's third law is exploited as shown in Listing 7, where `dfx` is the delta force in the x direction.

```
atomicAdd(&fx[i],  dfx);
atomicAdd(&fy[i],  dfy);
atomicAdd(&fz[i],  dfz);
atomicAdd(&fx[j], -dfx);
atomicAdd(&fy[j], -dfy);
atomicAdd(&fz[j], -dfz);
```

Listing 7: Exploiting Newton's third law, when applying the force.

Performing the velocity half step. Since both half-steps are identical, the same kernel is invoked twice per simulation loop. It works very similarly to the C implementation, but here, a thread updates only one index.

Updating the positions. Again, similar to the C implementation, but here each thread index is assigned a particle, which updates the position and applies boundary conditions on each coordinate component.

4.4 Pythonic Implementations

Python is a *batteries included* language that relieves the programmer of explicit manual memory management while providing an extensive standard library. In this project, two external libraries, namely NumPy (NumPy, 2026) and JAX (JAX, 2026), have been used to implement high-level implementations of the MD simulator.

NumPy provides vectorized multidimensional array operations, which makes it a good choice for data-parallel algorithms such as the one used in this project. JAX extends the NumPy implementation with just-in-time (JIT) compilation, which compiles selected functions into optimized machine code for both CPUs and GPUs. The following section describes how the MD algorithm was implemented using these two Pythonic frameworks.

4.4.1 NumPy

Similar to the imperative implementations described above, the NumPy implementation decomposes the spatial domain through a cell list, however, the operations are array-oriented. Instead of explicitly looping over the individual particles, the computations are delegated to kernels that operate on the entire array, i.e., the operations are expressed as vectorized array transformations. This reduces the overhead in the interpreter associated with explicit loops and allows for optimized low-level internal functions.

As in the imperative implementation, the pair-list construction is done by traversing neighbor cells and identifying interacting particles based on their distances and the cutoff radius. However, instead of the nested loops, the NumPy implementation computes the distances through *broadcasting*, which is a mechanism for expressing how arithmetic operations should be performed on arrays of different sizes. The use of the `None` keyword in the example shown in Listing 8 is an example of NumPy broadcasting. This allows NumPy to automatically compute pairwise distances between particles in neighboring cells simultaneously. Although the pair-wise distances themself are vectorized in the cell-list, the traversal of the cells requires Python iteration due to the

```
rij = x[pair_i] - x[pair_j]  # Distance.
rij -= L * np.round(rij / L) # Minimum image.
r2 = np.sum(rij**2, axis=1)
```

Listing 8: Computing distance though NumPy broadcasting.

varying particle count. In essence, the cells are iterated over, but the distances are evaluated through vectorization.

The indices of the pairs are stored in temporary arrays and finally merged into global arrays using concatenation such that the same index of the two arrays `pairs_i` and `pairs_j` represent particles that interact. A small optimization is implemented, which simply skips a cell if it is empty.

The force computation is also evaluated using vectorized array operations. Here, the LJ force is evaluated simultaneously for all pairs in the pair list using Newton's third law, as shown in Listing 9. Then, the forces can be

```
inv2 = 1.0 / r2
sr2 = (sigma**2) * inv2
sr6 = sr2**3
force_scalar = 24 * epsilon * inv2 * (2 * sr6**2 - sr6)
fij = force_scalar[:, None] * rij
np.add.at(f, pair_i, fij)
np.add.at(f, pair_j, -fij)
```

Listing 9: Evaluating the LJ force simultaneously for all pairs in the pair list using Newton's third law.

accumulated using a scatter-like reduction through `np.add.at`. This ensures the force contribution from all the interacting pairs is summed correctly while still allowing for the vectorized partial forces.

While this implementation uses data-parallel expressions, the broadcasting operations produce a large number of intermediate arrays. This contributes to more memory traffic, which might bottleneck the runtime significantly.

4.4.2 JAX

The JAX implementation is structured in a similar manner to the NumPy implementation described above, but due to the static nature of the JIT compiler in JAX, every array is required to be a fixed length at compile time. The main simulation loop and cell construction in the JAX implementation are identical to those in the NumPy implementation, with the exception of the cell list being built within the force computation function. This is due to JAX tracing the function at compile time, and hence all operations depending on particle positions must be expressed within the JIT-compiled section.

In the NumPy implementation, the pair list can grow dynamically when a new interaction is found as the simulation progresses. To accommodate this, the JAX implementation eliminates the pair list and allocates a flat array of fixed size $N \times 27$. Here, each entry represents a particle paired with one of the 27 neighbor cells. At runtime, `lax.fori_loop` iterates from zero to the maximum number of particles found in any cell and sums the force contributions. This and the neighbor cell lookup are shown in Listing 10.

```
# Neighbor cell index.
neighbor_cell = nx + nc * (ny + nc * nz)

# Look up start and count for each particles in the cell.
start = starts[neighbor_cell]
count = counts[neighbor_cell]
max_count = jnp.max(counts)

# Accumulated forces.
tx, ty, tz = lax.fori_loop(
    0,
    max_count,
    interact,
    (tx0, ty0, tz0)
)
```

Listing 10: Accumulating interaction forces using a neighbor cell lookup.

In the LJ force computation, a guard is implemented to prevent interactions that exceed the cutoff radius, self-interactions, and a bound to ensure the particle is within the cell. The mask and LJ force computation is shown in Listing 11. With the forces accumulated across the $N \times 27$ entries, they

```
# Lennard-Jones force.
mask = (r2 < rc2) & (r2 > 1e-12) & (k < count)
inv2 = jnp.where(mask, 1.0 / r2, 0.0)
sr2 = sigma**2 * inv2
sr6 = sr2**3
f = 24 * epsilon * inv2 * (2 * sr6**2 - sr6)
```

Listing 11: LJ force computation with conditions to exclude particles beyond the cutoff radius, avoid self-interactions, and ensure valid particle indices within the neighboring cell.

are reduced back to particles. In the NumPy implementation, this was done with a scatter reduction using `np.add.at` but since all $N \times 27$ entries are indexed by their source particle, we can use a segmented sum to sum all force contributions sharing the same particle index into a single output array of length N. The particles are then scattered back to their original ordering, essentially reversing the sort from the cell construction. The reduction and scatter are shown in Listing 12.

```
# Reduce the forces.
flat_forces = jnp.stack([tx, ty, tz], axis=1)
forces_sorted = jax.ops.segment_sum(flat_forces, particle_ids, N)

# Back to original order
f = jnp.zeros_like(x)
f = f.at[sorted_idx].set(forces_sorted)
```

Listing 12: Reducing the accumulated force contribution using a segmented sum over particle indices and then scattering the reduced forces back to the original particle ordering.

4.5 Functional Implementation in Futhark

Unlike all the previous implementations, the Futhark implementation expresses the simulator as a set of data-parallel array transformations that do not require manual thread and/or memory management or synchronization[1]. Instead, constructors like `map`, `scan`, `scatter`, and `reduce` are used, which allows the Futhark compiler to generate efficient binaries. With support for different backends like *multicore*, a single codebase can be multi-threaded on the CPU

[1]The Pythonic implementation did not require any memory or thread management.

and parallel with the CUDA backend on GPUs. The Futhark code is compiled to a C library using one of the backends, which a small C *host* program can invoke. The host program handles stuff like reading the input file, allocating memory, and writing the initial and final particle states to a file.

The Futhark implementation follows the same algorithm as described in Section 4.2, including the domain decomposition. However, the Futhark implementation is different with respect to how the force computation is structured as a fully independent per-particle evaluation rather than a pair-list.

The implementation uses the same domain decomposition by dividing the simulation domain into cubic cells, and particles are assigned to a cell depending on their coordinates. After assigning the particles to cells, they are sorted. This implementation depends on a radix sort primitive as shown in Listing 13. Noteworthy is the let-expression binding `bits` which computes the minimum

```
1 let bits = (64 - i64.clz n + 1)
2 let sorted = radix_sort_int bits
3     (\bit i -> (cell_ids[i] >> bit) & 1)
4     (map i32.i64 (iota n))
5 let cell_s = map (\i -> cell_ids[i]) sorted
```

Listing 13: Sorting the particles by cell index using the Futhark radix sort primitive.

bit width needed to represent the cell indices. This optimizes the radix sort by a constant factor since it will have to perform fewer iterations in the radix sort algorithm. This is mostly a concern with the CUDA backend, as the overhead of memory fetching will dominate the runtime on the CPU.

The biggest difference from the other implementations lies in how the force computation is structured. In the Futhark implementation, each particle computes its own net force independently by traversing its neighboring cells and examining all particles in them. Since each particle can do this independently, there is no need for a global shared pair list. In the other implementations, Newton's third law is exploited to avoid double computing the same force twice (one is negative of the other), and then updating the particles with an atomic operation. This isn't feasible in Futhark as it would require a coordinated scatter, which would break the parallel model. Therefore, in this implementation, both equal and opposite forces are computed, and although

doubling the evaluations, it is beneficial for GPU execution, since the use of atomics is avoided.

The Futhark implementation expresses the forces in a bulk parallel operation where the per particle, per neighbor-cell work is structured in a flat one-dimensional array for iterating over with size $27N$. This means each element represents a particle interacting with a neighboring cell, and here the particle index can easily be recovered with division and modulo operations as shown in Listing 14.

```
let max_neighbors = 27i64
let flat_forces = map (\idx ->
    let si = idx / max_neighbors -- index of particle in sorted
    ↪  order.
    let off_idx = idx % max_neighbors -- which neighbor cell?
    (...)
) (iota (n * max_neighbors))
```

Listing 14: Recovery of particle and neighbor cell index from the flattened $27N$ interaction array.

Each task in the flat array accumulates the force contributions from all particles in the target neighbor cell with a sequential loop. This results in a new array of partial forces, i.e., one per particle/neighbor cell pair.

For each particle, 27 partial forces contribute to it, one for each neighboring cell, meaning we can sum them to obtain the net force. This is done with a reduction using `reduce_by_index`, which, given two arrays of equal length, one containing values and the other the destination indices, sums values that share destinations. Here, the destinations are easily obtained by dividing the flat index by 27, as shown in Listing 15. Since all these operation where

```
let forces_s = reduce_by_index (replicate n (0.0, 0.0, 0.0))
    (\(ax,ay,az) (bx,by,bz) -> (ax+bx, ay+by, az+bz))
    (0.0, 0.0, 0.0)
    (map (\i -> i / max_neighbors) (iota (n * max_neighbors))) --
    ↪  repeat each particle 27 times.
    flat_forces
```

Listing 15: Reduction of the 27 partial forces using `reduce_by_index`.

performed, the forces are in sorted order, which means they must be re-ordered back to the original order before they can be used in the main simulation loop. This is done with a `scatter` operation, which takes the sorted forces and them back to their corresponding entry in the sorted array, essentially reversing the sort.

The rest of the implementation resembles the other implementation greatly: using the Velocity Verlet algorithm, which updates the positions and velocities using the half stepping scheme and is vectorizable by nature.

5 Performance Modeling

This chapter presents the methods for modeling and reasoning about the performance of the implementations. These metrics were chosen to achieve the goal of comparing how Futhark compares to the traditional languages in a scientific simulation in terms of performance. It begins by presenting a strategy for validating the correctness of the physical system based on energy conservation, then moves to outline a runtime measurement protocol. The Roofline model is introduced to help determine if the implementations are memory or compute bound before finally presenting Gustafson's law, which evaluates the scalability of the parallel implementations.

5.1 Validating the Implementations

While not the main focus of this project, the validity and correctness of the simulators are still important, as it is a fundamental requirement for any scientific simulation. For the simulator to be valid, it must follow the same physical laws we aim to simulate. In the system of the project, the number of particles, the volume of the box, and the energy are all constant, which means the total energy of the system is preserved over time, and thus the final state must hold the same energy as the initial state. Hence, for validating the system, this project measures energy conservation over time.

The total energy, E_{total}, of the system is defined by its Hamiltonian, which is the sum of the kinetic and potential energy (Griffiths and Schroeter, 2018):

$$E_{\text{total}} = E_{\text{kinetic}} + E_{\text{potential}} \tag{5.1}$$

where the kinetic energy is derived from the mass, m, and the velocities, v, of the N particles as:

$$E_{\text{kinetic}} = \sum_{i=1}^{N} \left(\frac{1}{2} m_i \mathbf{v}_i^2 \right) \tag{5.2}$$

and the potential energy is the LJ potential, $u(r)$, based on pairwise distances r_{ij} between particles i and j:

$$E_{\text{potential}} = \sum_{i<j} u(r_{ij}) \tag{5.3}$$

where $u(r)$ is defined as equation 2.1. Hence, the total energy is:

$$E_{\text{total}} = \sum_{i=1}^{N} \left(\frac{1}{2} m_i \mathbf{v}_i^2\right) + \sum_{i<j} \left(4\varepsilon \left[\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right]\right) \tag{5.4}$$

While the numerical integration of the Velocity Verlet algorithm introduces a local error at each time step due to the time being discretized, it limits the long-term energy drift. This means that the observed fluctuations remain bounded by a constant rather than diverging over time.

To validate the implementations with this measurement, the simulators will output their total energy at two different times. First, at the 5000$^{\text{th}}$ step, and second, at the final step. The energy at the 5000$^{\text{th}}$ step is the *initial* energy despite not being at the initial time step. Because the simulation starts from the initial positions and velocities from the generated input, which are set at an unnatural uniform grid, it must be allowed some time to relax and reach an equilibrium state. The transformation will henceforth be referred to as the *equilibrium phase*, in which the particles are transforming from the input state to a natural equilibrium state. With the initial energy, E_{initial}, and final energy, E_{final}, the relative energy fluctuation is found as:

$$\Delta E = \left| \frac{E_{\text{final}} - E_{\text{initial}}}{E_{\text{initial}}} \right| \tag{5.5}$$

This metric will be used for all simulators to demonstrate their energy conservation properties over time to confirm periodic boundary conditions, the minimum image convention, and that the force computations are mathematically consistent in Section 6.1 and should remain constant over time.

5.2 Measuring Runtime

To assess the performance of the simulators, it is necessary to have a consistent benchmarking protocol. There are several ways to measure the runtime of

a program, and this Section aims to outline some of them and go over the rationale behind them.

- **End-to-end runtime.** This method measures the entire running time of the program, from when it gets invoked until it terminates. This includes allocating the initial memory, parsing the input, the simulation loop itself, and writing the final state to a file. If it is a GPU-executed program, the time it takes to transfer data to and from the GPU device is also included. This method represents the practical *wait time* for a user, i.e, how long do we wait until the result is here.

- **Kernel only.** This method isolates the performance from the overhead, meaning it measures only the time spent on the simulation loop. The input parsing, initial memory allocation, and writing the final state are ignored. This is particularly helpful when determining if the GPU is reaching its theoretical limit, since it only measures the intensive parts of the program.

- **Time per step.** Instead of measuring the entire simulation loop, this method measures the average time for a single time step. This helps identify how the performance changes over time, such as if the particles suddenly cluster in a single cell, causing a load imbalance. Since the $\mathcal{O}(N)$ complexity of the cell list approach assumes uniform particle distribution, a cluster would cause a localized $\mathcal{O}(N^2)$ bottleneck.

- **Interactions per time.** This metric counts the number of pairwise particle interactions per unit of time (e.g., million interactions per second). This is effectively a normalized runtime, which allows for direct comparison across different simulations with varying sizes of N and densities, where all other methods described require these to be constant for comparison.

In this project, the end-to-end method is chosen as the runtime metric because it provides a realistic assessment of a simulator's utility. By measuring the runtime of the entire execution, the overhead associated with the high-level languages (kernel loading, JIT compilation, buffer allocation) is accounted for. The end-to-end method produces a metric for the total runtime that better

supports the aim of the project on how Futhark compares to the performance of traditional languages.

A notable concern of the end-to-end method is the influence of the system state, i.e., the OS and hardware may cache frequently used memory pages. If a program is executed multiple times, a later run may appear faster because the input is already parsed and residing in the cache. To mitigate this concern, a *warm-up* procedure is used. Here, an initial run is performed with the real input, but the result is discarded, so the actual performance is recorded instead of the one time warm-up time.

The other methods mentioned above are, however, valuable for performance tuning and can be of great help to identify bottlenecks like load imbalance and memory latency. *Kernel only* and *interactions per time* let us differentiate between the overhead of the language runtime and the efficiency of the generated code, which are useful in the optimization phase of development.

5.3 The Roofline Model

The *Roofline Model* identifies the factors limiting a program's performance on specific hardware, i.e., how well the program utilizes the hardware (Williams *et al.*, 2011). This metric can reveal whether a program has a bottleneck in compute throughput or memory bandwidth.

The model is defined as the relationship between the *peak arithmetic throughput*, P_{peak}, and *peak memory bandwidth*, B_{peak}. P_{peak} is measured in Floating Point Operations per Second (FLOPs/s), and B_{peak} is measured in bytes/s. The attainable performance, P, is given by:

$$P = \min \begin{cases} P_{\text{peak}} \\ \text{OI} \times B_{\text{peak}} \end{cases} \tag{5.6}$$

where OI is operational intensity, which is defined as the ratio of arithmetic operations, W, to the total bytes of data transferred to and from main memory, Q, i.e., $\text{OI} = {}^{W}/_{Q}$. This relationship is plotted as a graph where the memory roof is a diagonal line and the compute bound is a horizontal line correspond-

ing to the peak throughput of the specific hardware. Each program is then represented as a single point at its measured OI and achieved performance. A point lying on the roofline indicates a program that is fully utilizing the hardware at its given OI, while the vertical distance between a point and the roofline represents the gap between what the hardware could deliver and what the program actually achieves. The gap can be interpreted as the wasted potential of the implementation. Programs with low OI are memory-bound, and programs with high OI are computation-bound.

While the Roofline model can indicate what bounds a program, it relies on several assumptions. For instance, it assumes peak memory bandwidth and peak compute throughput at the time of running, which is typically not attainable on a real machine. Peak performance is especially a concern on shared resources like an HPC cluster, where one can request and get full performance of a CPU core, but the memory bus remains shared with the rest of the users on the shared machine, meaning the available bandwidth will be lower than the theoretical peak at any given moment.

Applying the Roofline model in this project helps to determine what the programs are bounded by, and to do so, the OI must be quantified. This will be done in Section 6.3 along with an analysis of the results.

5.4 Gustafson's Law

To evaluate the performance of parallel programs, the pure time speedup is not the only factor to consider, but also how they scale with more resources. In essence, how well does the implementation scale when we increase the number of available cores? To evaluate this, Gustafson's law is applied, which provides a way for analyzing the scalability of how the workload can grow with the number of processing units while maintaining a constant execution time (Shi, 1996).

Gustafson's law assumes constant total execution time while the program size, in this case, the number of particles N, increases with the number of available CPU cores. The speedup is given by:

$$S(p) = p - \alpha(p - 1) \tag{5.7}$$

where S is the speedup, p is the number of cores, and α is the fraction of the computation that is sequential. Hence, with a small α, a near-linear speedup can be achieved with the number of cores.

In the simulators of this project, the majority of the computational cost comes from the evaluation of pairwise forces and the update of particle positions and velocities as stated in the implementation descriptions in Chapter 4. These operations are inherently parallel, however, certain parts remain sequential or require synchronization, including prefix sums in cell construction, particle reordering, and the reduction of force contributions. These components contribute to the sequential fraction α and limit the achievable scalability.

To measure the scaling empirically, the number of particles N is scaled such that they target a constant execution time. For example, if the number of cores doubles, it may not be possible to maintain a constant runtime while doubling the particle count. If the program scales perfectly, the execution time remains flat as both N and p increase. However, if the program fails to keep a constant execution time when N and p grow, then this indicates when the sequential fraction, α, is becoming the binding constraint, which is likely caused by the memory bandwidth or synchronization overhead associated with multicore programs. This experiment is conducted for both the OpenMP and Futhark multicore implementations in Section 6.4, allowing a direct comparison between a manually parallelized and a compiler-parallelized approach to the same algorithm.

6 Performance Results and Analysis

This chapter presents an analysis of the empirical results for all the implementations. It begins by confirming the correctness using energy conservation, then compares the end-to-end runtimes at different system sizes. The operational intensities are found for the implementation and applied in the Roofline model to help determine if they are memory or computation bound. Finally, an empirical scaling analysis is applied along with Gustafson's law to evaluate how the parallel implementation behaves when given additional resources.

6.1 Validation Results

Figure 6.1 shows the kinetic, potential, and total energy of the C implementation over $20,000$ steps with 1000 particles. Overall, the energy is conserved, although there are big fluctuations in the first ~ 3000 steps in kinetic and potential energy. This is caused by the equilibrium phase, where the system goes from the unnatural initial uniform positions to a natural configuration. After the equilibrium phase, the system enters a stable state, and since no further energy is injected or dissipated, the total energy is flat within numerical precision. Since long-term energy drift remains bounded rather than diverging, this validates and confirms the implementation is correct as presented in Section 5.1. The remaining seven implementations produce near identical results and are shown in Appendix C.

6.2 End-to-End Runtime

The end-to-end runtimes for all eight simulators across system sizes ranging from $N = 1,000$ to $N = 1,000,000$ particles, plotted on a log-log scale, are shown in Figure 6.2 and the data shown in Appendix D. These were run on an

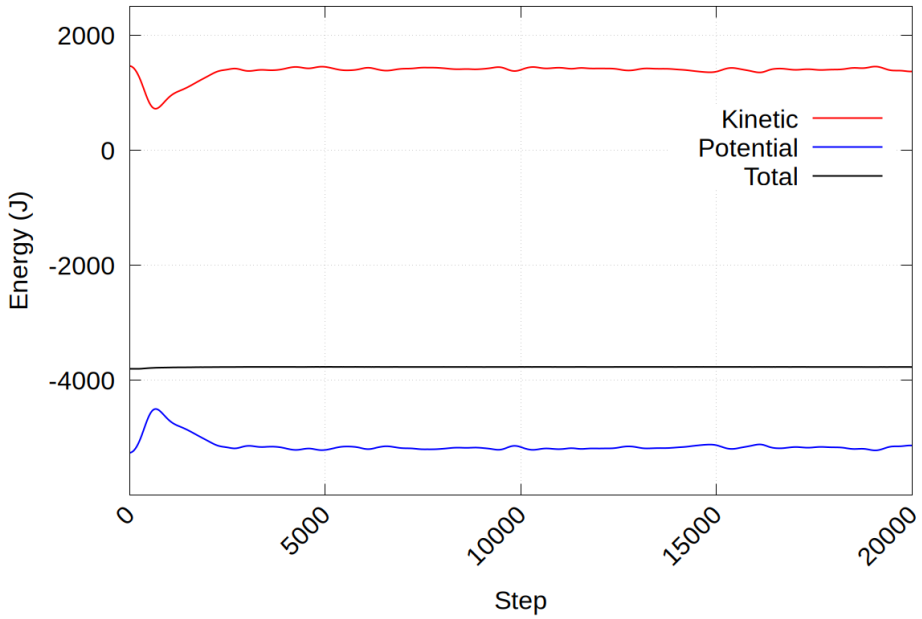


Figure 6.1.: Kinetic, potential, and total energy over time in the C implementation.

HPC cluster with an AMD EPYC 7352 CPU and an Nvidia A100 GPU. While each job is allocated a dedicated GPU and CPU cores, the memory bus is shared across all users on the node. Due to the memory bus being shared, the available bandwidth is almost certainly lower than the theoretical peak at any given time, which introduces some variability in the runtimes.

NumPy has been excluded from the largest particle system of one million particles because the runtime was too long to run on the HPC cluster. All the implementations show a near-linear relationship, consistent with the expected $\mathcal{O}(N)$ complexity of the algorithm, as further supported by the coefficient of determination (R^2) values for the linear fit, all greater than 0.992. However, most coefficients of determination are one, and the lower coefficient in NumPy likely stems from the overhead of intermediate array allocations, which become more prominent at larger system sizes.

The GPU-based implementations: CUDA, JAX, and Futhark-CUDA, are the fastest simulators at every system size except the very first, where OpenMP out-performs JAX. CUDA achieves the lowest runtime across all measured system sizes. Among the CPU-based implementations, the multicore simulators, OpenMP, and Futhark-multicore are the fastest, with the handwritten OpenMP steadily outperforming Futhark-multicore. The sequential implementations, C, NumPy, and Futhark-C, are the slowest, with handwritten C being the fastest

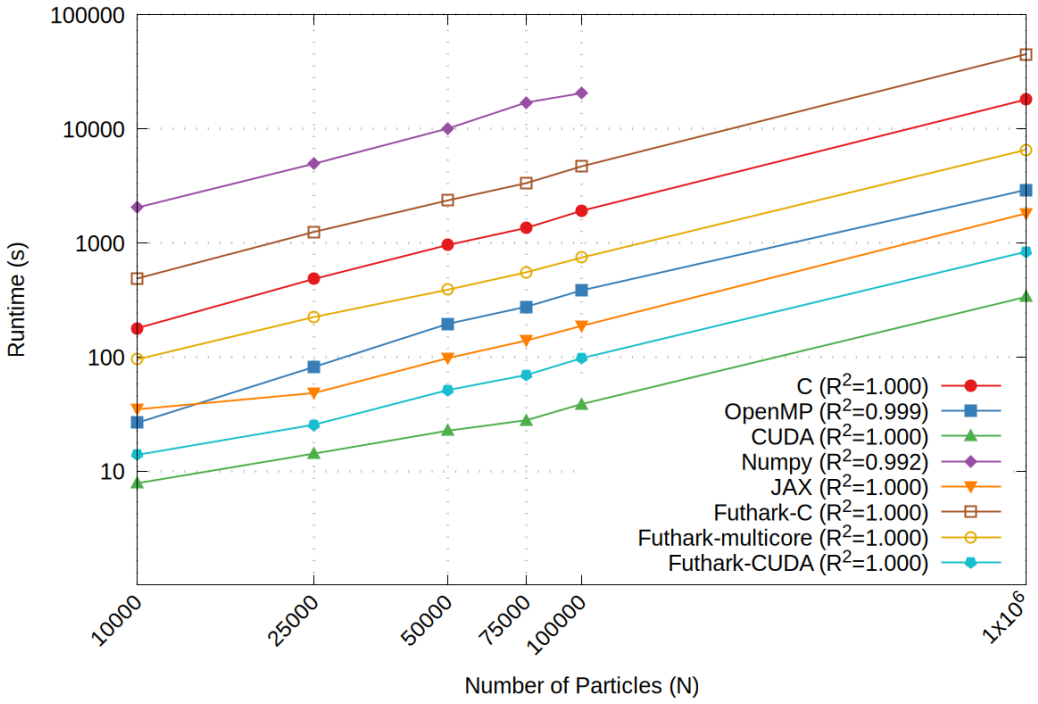


Figure 6.2.: End-to-end runtime vs. number of particles N for all eight simulators plotted on a log-log scale. Particle system ranges from $N = 1,000$ to $N = 1,000,000$ particles except NumPy, which is excluded at one million particles due to long runtime.

of the three. These results are expected, with handwritten implementations being faster across all categories (sequential, multicore, and GPU), since the programmer can apply hardware-aware optimizations that the compiler cannot automatically infer.

These results show that while Futhark is not faster than the traditional counterparts, it remains competitive.

6.3 Roofline Model Interpretation

Since some of the simulators in this project are executed on a CPU and others on a GPU, the Roofline model is interpreted accordingly. The five CPU-based implementations (C, OpenMP, NumPy, Futhark-C, and Futhark-multicore) were evaluated on the AMD EPYC 7352 CPU, and the three GPU-based implementations (CUDA, JAX, and Futhark-CUDA) were evaluated on the Nvidia A100 GPU.

6.3.1 Hardware Roof

As previously mentioned, the HPC cluster used in this project contains an AMD EPYC 7352, which runs at 2.3 GHz with AVX2 vector and fused multiply-add (FMA) support, which processes double-precision floating-point numbers simultaneously. Each Zen 2 core can perform two 256-bit AVX2 FMA instructions per cycle, and since a 256-bit AVX2 vector holds four double-precision floating-point numbers and each FMA performs one multiply and one add operation, the peak throughput per core is $4 \times 2 \times 2 = 16$ FLOPs per cycle for each core. The processor provides 24 cores per socket with two sockets, hence a total of 48 cores, which means the throughput is:

$$48 \text{ cores} \times 2.3 \text{ GHz} \times 16 \text{ FLOPs/cycle/core} \approx 1{,}766.4 \text{ GFLOPs/s} \tag{6.1}$$

The memory bandwidth was measured using the STREAM Triad benchmark (Advanced Micro Devices, Inc., 2026) with all 48 threads, yielding 86.3 GB/s. The GPU used is an Nvidia A100 and its theoretical peak double-precision throughput is $9{,}700$ GFLOPs/s and its HBM2 memory bandwidth is $1{,}935$ GB/s (NVIDIA, 2024).

6.3.2 Measuring the Operational Intensity

The operational intensity was obtained differently for the CPU- and GPU-based implementations. For the CPU, the FLOPs and bytes were both measured using hardware performance counters, giving the exact measurements. However, the hardware performance counter was not available on the HPC cluster, so the FLOPs and bytes were derived analytically from the source code directly. Since the main computation is performed in the force evaluation, this function is the base of the FLOP and byte count. The OI was found as described below.

CPU. In the same simulator run, the FLOPs and bytes transferred in memory were both recorded. The FLOPs were measured using the AMD hardware counter, `rFF03`, and collected via the Linux tool `perf stat`. The bytes transferred in memory were measured using the counter `r1F43`, which counts 64-byte cache-line fills from DRAM. The OI is then found as:

$$\frac{\text{rFF03 count}}{\text{r1F43 count} \cdot 64 \text{ bytes}} \tag{6.2}$$

which for each CPU-based simulator is shown in Table 6.1.

Implementation	OI	P_{CPU}
C	7.23	624
OpenMP	2.08	179
Numpy	0.043	3.7
Futhark-C	11.66	1005
Futhark-multicore	3.19	275

Table 6.1.: Operational intensities for each of the CPU-based implementations, where P_{CPU} is computed using equation 5.6 with $P_{\text{peak}} = 1{,}766$ GFLOPs/s and $B_{\text{peak}} = 86.3$ GB/s.

GPU. For the CUDA implementation, the force kernel performs 35 FLOPs per pair and accesses six double-precision positions, two integer indices, and six atomic writes, totaling 106 bytes. The force computation in JAX performs 33 FLOPs per candidate pair and moves 96 bytes. Futhark-CUDA performs 38 FLOPs and uses double precision throughout the implementation, giving it 96 bytes. The OI is then given by $^{\text{FLOPs}}/_{\text{bytes}}$, as shown in Table 6.2 for each GPU-based simulator.

Implementation	OI	P_{GPU}
CUDA	0.322	623
JAX	0.344	66
Futhark-CUDA	0.395	764

Table 6.2.: Operational intensities for each of the GPU-based implementations, where P_{GPU} is computed using equation 5.6 with $P_{\text{peak}} = 9{,}700$ GFLOPs/s and $B_{\text{peak}} = 1{,}935$ GB/s.

6.3.3 Roofline Results

The Roofline plots for the CPU and GPU are shown in Figures 6.3 and 6.4, respectively.

In Figure 6.3, we see that all five CPU implementations lie below the point of being bounded by computation. This confirms the MD simulations are memory-bound on the CPU. The parallel implementations achieve lower OI than their serial counterparts: OpenMP (2.08) versus C (7.23), and Futhark-multicore (3.19) versus Futhark-C (11.65). This happens because the cores simultaneously access the memory bus, resulting in the DRAM traffic increasing faster than the FLOP count scales with the core count. However, the parallel

implementations obtain an overall higher performance, with Futhark-multicore achieving 14.26 GFLOPs/s and OpenMP achieving 9.09 GFLOPs/s compared to their serial counterparts at 2.45 and 1.31 GFLOPs/s, respectively. This means that the speedup achieved from parallelizing the implementation is due to exploiting available memory bandwidth rather than improving on arithmetic efficiency. It is also worth noting that the NumPy implementation is a clear outlier with an OI of only 0.043 and a measured performance of only 0.24 GFLOPs/s. This reflects the overhead of the vectorized array model that Python provides with its allocation and discarding of larger intermediate arrays, which generates far more DRAM traffic per FLOP than the compiled implementations. The predicted ceilings P_{CPU}, as shown in Table 6.1 confirms this as every

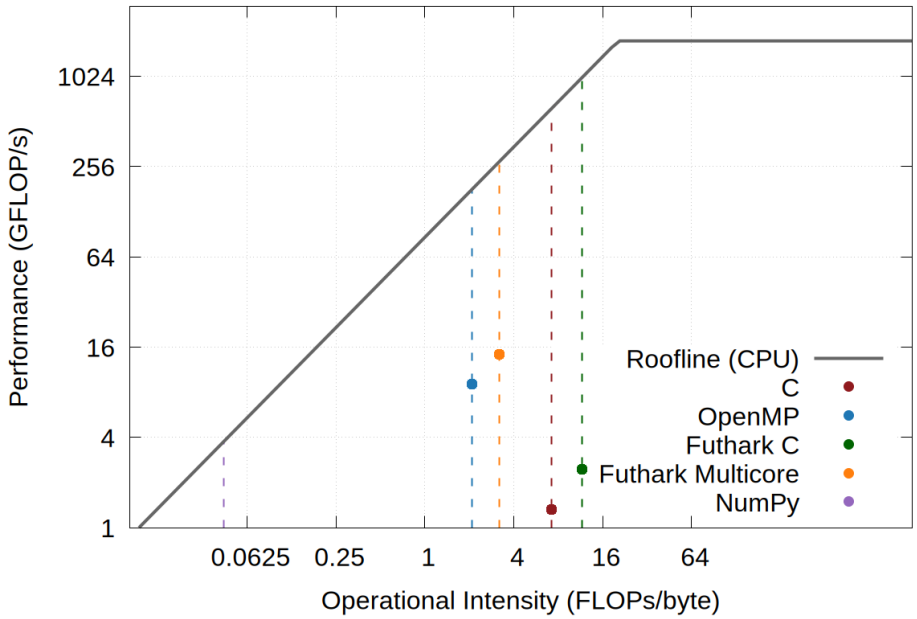


Figure 6.3.: Roofline Model for CPU implementations on AMD EPYC 7352. The memory-bandwidth ceiling, measured with the STREAM Triad, is $86.3\ \text{GB/s}$ and the compute ceiling is $1{,}766\ \text{GFLOPs/s}$.

prediction falls below $P_{\text{peak}} = 1{,}766\ \text{GFLOPs/s}$. Futhark-C had the highest ceiling at $1{,}005\ \text{GFLOPs/s}$, which reflects more arithmetic work and not more efficient memory access.

All three of the GPU implementations, as shown in Figure 6.4, are bounded by memory as well, further confirming that the MD simulator is memory-bound. The Futhark-CUDA implementation has the highest OI (0.395) and the highest measured performance at 40.78 GFLOPs/s, followed by JAX with an OI of 0.344 and 18.61 GFLOPs/s, and finally CUDA with an OI of 0.322

and 1.95 GFLOPs/s. This ordering in OI reflects the memory access patterns of the implementations: the CUDA implementation processes only active pairs using a pre-built neighbor list, which reduces the number of evaluated pairs per simulation time step, while JAX and Futhark-CUDA both use the cell list structure, which evaluates all $N \times 27$ potential pairs per force call. This increases both the FLOP count and the number of bytes accessed, but the higher potential pair count results in a slightly better OI than the neighbor-list approach. The large gap in measured performance between CUDA (1.95 GFLOPs/s) and the other two GPU implementations is notable and likely reflects the overhead of the atomic operations used in the pair-list-based force accumulation, which can bottleneck throughput when many threads contend for the same memory address. The predicted ceilings P_{GPU}, as shown in Table

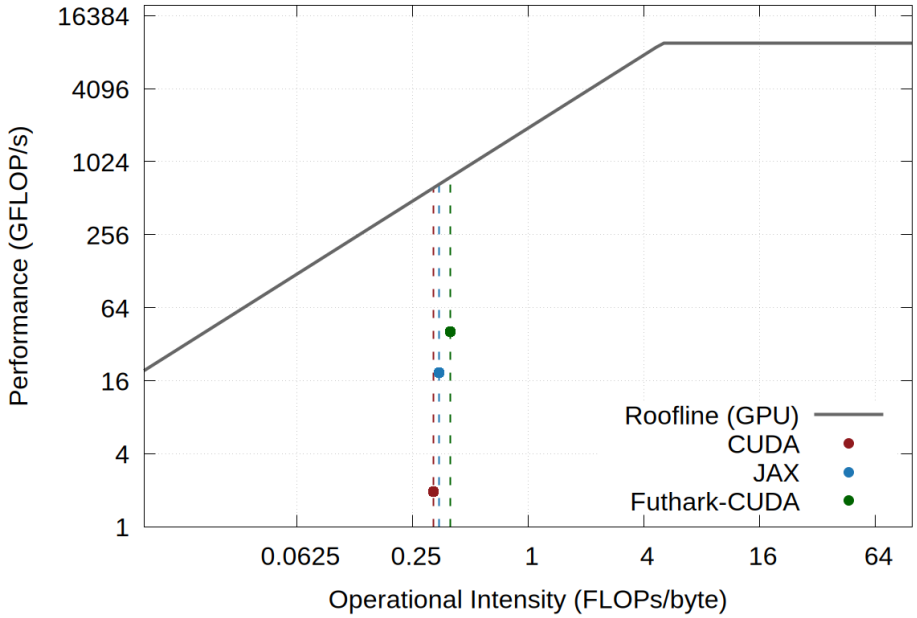


Figure 6.4.: Roofline Model for GPU implementations on Nvidia A100 with analytically derived operational intensities. The memory-bandwidth ceiling $1{,}935$ GB/s and the compute ceiling is $9{,}700$ GFLOPs/s.

6.2, are similarly below the $9{,}700$ GFLOPs/s for all three GPU simulators, further confirming the GPU implementations are memory-bound.

6.4 Empirical Scaling Analysis

To evaluate the parallel scaling of the simulators implemented in OpenMP and Futhark with the multicore backend, an empirical experiment was conducted.

As mentioned in Section 5.4, if the program is completely parallel, it scales perfectly and the execution time remains constant. In this experiment, the number of particles, N, was increased with the number of processing cores, targeting a constant execution time of ∼17 seconds for OpenMP and ∼35 seconds for Futhark on the HPC cluster. The performance is reported as millions of particle pair interactions per second (MPPIS), such that the runs are normalized with the different sizes of N. The results are shown in Table 6.3.

Cores	OpenMP N	OpenMP MPPIS	Futhark mc N	Futhark mc MPPIS
1	1000	7.4	1000	7.8
2	2000	15.4	2000	13.1
4	3500	30.0	3500	27.5
8	5700	51.6	5700	37.4
16	6100	55.7	6100	47.9
32	4600	35.7	9000	66.8
64	3500	27.3	7000	59.7

Table 6.3.: Scaling results for OpenMP and Futhark multicore (mc) maintaining a target runtime of ~ 17 seconds for OpenMP and ~ 35 seconds for Futhark multicore. MPPIS = Million Particle Pair Interactions per Second

OpenMP From one to four cores, the throughput scaled close to linear, increasing from 7.4 MPPIS to 30 MPPIS. This is a growth factor of 4.1 for a quadruple increase in cores. From four to eight cores, however, the throughput only grows from 30 to 51.6 MPPIS, which is a factor of 1.72 for a doubling of cores. This implies a scaling bottleneck, which is likely due to the memory bus being saturated. As the core count increases, the memory demand increases as well, but the available memory bandwidth does not increase, which is more apparent when going from eight to sixteen cores. Here, the throughput only grew marginally from 51.6 to 55.7 MPPIS, and beyond sixteen, the throughput decreases, and the particle count even had to be decreased to keep the constant target runtime. Applying Gustafson's law on the well scaling parts from one to sixteen cores, the speedup is $S = {}^{55.7}/_{7.4} \approx 7.5$. The the sequential fraction, α, of the OpenMP implementation is then:

$$\alpha_{\text{OpenMP}} = \frac{p - S}{p - 1} = \frac{8 - 7.5}{8 - 1} \approx 0.07 \tag{6.3}$$

i.e. 7%. This suggests the program is very parallel and the *wall* observed beyond eight cores is due to hardware, which aligns with the Roofline model

result discussed in Section 6.3. Regardless of available cores, the memory bus gets starved due to the high demand for memory access.

Futhark multicore The Futhark implementation shows a different and more well-behaved scaling profile for higher core counts. From one to four cores, the throughput was increased from 7.8 to 27.5 MPPIS, a factor of 3.5, i.e., close to linear as well. From four to eight cores, however, it only increased from 27.5 to 37.4 MPPIS, a factor of 1.36, similar to OpenMP. However, unlike OpenMP, Futhark continues to improve the throughput beyond eight cores and only starts to decline after 32 cores. Applying Gustafson's law on core count from one to 32, the speedup is $S = {}^{66.8}/_{7.8} \approx 8.6$. The the sequential fraction, α, of the Futhark multicore implementation is then:

$$\alpha_{\text{Futhark}-\text{multicore}} = \frac{p - S}{p - 1} = \frac{32 - 8.6}{32 - 1} \approx 0.75 \tag{6.4}$$

i.e. 75%. The higher sequential fraction compared to the OpenMP implementation suggests that the Futhark multicore implementation carries some additional overhead besides the memory bandwidth. This is likely from the compiler-generated parallelization of Futhark, which must generate its own scheduler and work distribution strategy, which introduces some synchronization and/or load balancing costs that the OpenMP implementation avoids by letting the programmer do it manually.

Despite the single-core results being nearly identical, 7.4 vs 7.8 MPPIS, the two implementations scale differently. OpenMP is faster up to sixteen cores but decreases drastically after that point, while Futhark is able to scale up to 32 cores before decreasing. The compiler-generated parallelism in Futhark therefore produces a better scaling profile than the handwritten OpenMP at the cost of the lower throughput at low to mid count of cores. For workloads that require scaling beyond sixteen cores, Futhark is the better choice on this specific hardware.

7 Discussion

This chapter reflects on the qualitative findings of the project. First, it presents how the abstraction levels of the implementation's ability to express the physical system and their trade-offs between programmer effort and program performance. Second, it discusses how the abstraction levels influenced correctness and the debugging experience, along with the type of errors that arose. Finally, the chapter then reflects on Futhark's multi-backend compilation, which allows for a single code base to target sequential and parallel CPU execution as well as GPU execution, are presented.

7.1 Abstractions Across the Implementations

All the implementations in this project express the same system of particles interacting via the Lennard-Jones potential, however, the languages imposed fundamentally different ways of thinking and reasoning about it. In C and OpenMP, the programmer is close to the hardware and has to make decisions about memory management, loop ordering, and parallelization. For instance, as mentioned in Section 3.1, the choice of using the structure of arrays rather than an array of structures was not an automated language feature but instead a manual optimization. Such optimizations require the programmer to have an understanding of CPU cache and how it behaves.

CUDA has a similar abstraction level to C and OpenMP, but further requires the programmer to have an understanding of the GPU. The GPU kernels written in CUDA are performing from the perspective of the hardware threads, and the programmer has to be explicit about thread assignment, warp behavior, and memory management both on the CPU and the GPU. The additional workload imposed by CUDA can be demanding for the programmer, and the mental

overhead of tracking which code executes on the CPU and GPU, combined with the manual memory management, makes it easy to introduce subtle errors.

NumPy sits at a higher level of abstraction and was by far the easist to implement because the broadcasting syntax closely resembles mathematical notation. With the ease of implementation comes the price of runtime, as NumPy was the slowest of the implementations. The vectorized array approach allocates and discards a large number of intermediate arrays, flooding the memory traffic and dominating the runtime. The memory waste, however, is invisible to the programmer as the memory management is automated.

JAX was used to improve runtime while maintaining a NumPy-like level of abstraction. JAX cleverly uses a JIT tracer, which, for the most part, is hidden from the programmer. However, the tracer did cause problems with the particle pair building as the tracer requires all arrays to have a fixed shape at compile time, and the particle pair arrays grow dynamically as the simulation advances. To work around this, the pair list was replaced by a fixed-sized $N \times 27$ array, with each entry representing a particle paired with one of its 27 neighboring cells. While this workaround works and is correct, it was not motivated by the physical system but was necessary due to the constraints of the JAX framework.

Futhark does not require the programmer to have an understanding of the hardware, as the language expresses what is to be computed rather than how to compute it. These computations are expressed as data parallel array transformation, which is closer to how one would describe the physics than a traditional program. For instance, the force computation is structured as a map over the particle array instead of an explicit loop, which feels more purposeful and natural in the context of implementing a physical system. However, this abstraction comes with some constraints, for instance, Newton's third law (particle i exerts the same and equal force on particle j as j on i) is a natural optimization but requires a coordinated scatter, which breaks the data parallel model, which introduced additional overhead and runtime, and so it was abandoned.

Reflection on these experiences, a pattern emerges: low-level languages like C, OpenMP, and CUDA make performance visible but put the full burden of optimization on the programmer. The high-level implementation of NumPy

reduces this burden, but its runtime is entirely too slow to be feasible for large array operations due to the memory traffic of intermediate arrays. JAX recovers the runtime with JIT compilation while keeping the burden low, but it introduces constraints in how the algorithm can be implemented in non-obvious ways. The high-level implementation in Futhark allows the programmer to reason about the physical system without concern for hardware-dependent optimizations while keeping the runtime low.

7.2 Debugging Across Abstractions

The correctness of any program is important, and in this project is no exception. The abstraction levels across the implementations dictated how errors and bugs arose and how they were found and dealt with. In the low-level implementations, the errors would result in an obviously wrong result or crash the program entirely with, e.g., a segmentation fault. Such errors typically included an out-of-bounds array index, race conditions, or a missing boundary wrap, and these were easily found with a simple print statement.

In the high-level implementations, however, the errors were harder to locate since the abstraction hid the mechanics behind the execution. For instance, Futhark would not crash or throw an error if the scatter operation did not reverse the sort correctly, instead the force would just be applied to the wrong particles, resulting in an energy drift.

Despite being harder to debug, Futhark offered a valuable guarantee: being purely functional. Here, race conditions, synchronization barriers, aliased writes, etc., are impossible. This means bugs and errors related to threads, which are common and easy to make in OpenMP and CUDA, are eliminated in their entirety.

Despite relying on a high-level language, the Pythonic implementations still exposed the underlying mechanics of MD well enough to facilitate easy debugging. However, JAX gave some cryptic errors when the arrays did not match the expected shape at trace time.

Across the implementations, the energy conservation checker proved to be valuable for debugging. Here, the correctness of an implementation could be checked, guaranteeing it worked beyond simply compiling. Without it, subtle bugs would easily have been introduced, violating the underlying physics.

7.3 Backend Flexibility in Futhark

One of the large advantages of Futhark is its ability to target multiple hardware backends from the same codebase. In this project, these included a sequential C, a parallel multicore, and a parallel CUDA backend for Nvidia GPUs. This backend flexibility provides the capability to develop and test particle systems with a small number of particles on a local workstation, and later run them with a far greater number of particles on an HPC cluster. The alternative is to write and maintain three separate codebases (e.g., C, OpenMP, and CUDA), which carries an increased developer burden and a risk of inconsistencies across targets. Futhark eliminates that burden and risk entirely.

The cost of the flexibility, however, is visible in the scaling results as presented in Section 6.4. Here, the Futhark multicore backend showed a sequential fraction of 75% vs. 7% in the OpenMP implementation. This is an expected tradeoff as the compiler makes the decisions about work distribution and memory management. Whether the tradeoff is acceptable depends on the situation, as a physicist writing a particle simulator may prioritize the correctness and physics itself rather than a few percent faster runtime.

8 Conclusion

This thesis investigated how Futhark compares to traditional languages for scientific simulation in both performance and programmer experience.

Chapter 2 established the mathematical model for the molecular simulation used in this project, describing the forces governing it, the Lennard-Jones potential, and Newton's laws of motion. It further described how time was discretized and advanced with the Velocity Verlet algorithm, and how the computational complexity was reduced from $\mathcal{O}(N^2)$ to $\mathcal{O}(N)$ using a cell decomposition of the domain. Chapter 3 described some hardware-aware optimizations, such as the structure of array memory layout, along with arithmetic simplifications. Chapter 4 showed how the mathematical model was implemented in C, OpenMP, CUDA, NumPy, JAX, and Futhark, which vary in abstraction level.

In Chapters 5 and 6, the performance modeling and results were presented, which showed that the low-level implementations outperformed Futhark compiler-generated versions at every level, sequential, multicore, and GPU, and they all maintained the expected $\mathcal{O}(N)$ complexity. The Roofline analysis in Section 6.3 showed that all the implementations, both on the AMD EPYC 7352 and the Nvidia A100, are memory-bound. The scaling analysis in Section 6.4 using Gustafson's law showed that OpenMP scaled near linearly up to sixteen cores but failed to scale at a higher core count, while Futhark-multicore carried a higher sequential fraction, and it was able to continue scaling up to 32 cores.

Beyond the empirical data obtained, the qualitative experience of working with the languages across their abstraction levels reveals a clear trade-off: The low-level languages C, OpenMP, and CUDA give the programmer more control but come with the entire burden of correctness and optimization, including memory and thread management, both of which require hardware understanding and are hard to reason about. While NumPy eliminates this

burden, its runtime proved too slow to be considered. JAX, while still reducing the programmer burden, recovers much of the runtime using JIT compilation, but introduces non-obvious constraints on array shapes as mentioned in Section 7.1, requiring changes to the general algorithm. Futhark, however, allows the programmer to disregard thinking about hardware, including memory and thread management, and instead focus mostly on the physical system to be implemented.

In Section 7.2, the debugging experience across abstraction levels was discussed. The low-level implementations would crash or produce obviously wrong results, while the high-level implementation in Futhark would silently produce subtly wrong results, making it hard to isolate the bug. Due to the pure functional model of Futhark, bugs, including race conditions and aliased writes, were eliminated, which were a significant source of errors encountered in OpenMP and CUDA in this project.

As discussed in Section 7.3, Futhark's backend flexibility allows the programmer to develop a single codebase that can target a sequential and multicore CPU as well as an Nvidia GPU, which removes the need to maintain three separate implementations. The burden of maintaining three implementations, ensuring no inconsistencies, while optimizing all three, was a burden experienced directly in this project. While Futhark is not the fastest implementation, it remains competitive with the low-level implementations, as shown in Figure 6.2, with the added benefit of a single code base versus three.

9 Bibliography

Advanced Micro Devices, Inc. (2026). *STREAM Benchmark*. Accessed: 2026-05-19. URL: https://www.amd.com/en/developer/zen-software-studio/applications/spack/stream-benchmark.html.

Allen, M. P. and D. J. Tildesley (1998). *Computer Simulation of Liquids*. Oxford Science Publications.

Binder, Kurt, Jürgen Horbach, Walter Kob, Wolfgang Paul, and Fathollah Varnik (Jan. 2004). „Molecular dynamics simulations“. In: *Journal of Physics: Condensed Matter* 16.5, S429.

Chandra, Rohit (2001). *Parallel programming in OpenMP*. Morgan kaufmann.

Elsman, Martin, Troels Henriksen, and Cosmin E Oancea (2018). „Parallel programming in Futhark“. In: *DIKU, November*.

Griffiths, D. J. and D. F. Schroeter (2018). *Introduction to Quantum Mechanics*. Cambridge University Press.

Hloucha, M. and U. K. Deiters (1998). „Fast Coding of the Minimum Image Convention“. In: *Molecular Simulation* 20.4, pp. 239–244. eprint: https://doi.org/10.1080/08927029808024180.

Huang, Kerson (1987). *Statistical Mechanics*. Wiley.

JAX (2026). *JAX Documentation*. URL: https://docs.jax.dev/en/latest/ (visited on May 19, 2026).

Lenhard, Johannes, Simon Stephan, and Hans Hasse (2024). „On the History of the Lennard-Jones Potential“. In: *Annalen der Physik* 536.6, p. 2400115.

Magenheimer, Daniel, Liz Peters, Karl Pettis, and Dan Zuras (Oct. 1987). „Integer Multiplication and Division on the HP Precision Architecture“. In: vol. 37, pp. 90–99.

McKee, Sally A (2004). „Reflections on the memory wall“. In: *Proceedings of the 1st conference on Computing frontiers*, p. 162.

NumPy (2026). *NumPy*. URL: https://numpy.org/ (visited on May 19, 2026).

NVIDIA (2024). *NVIDIA A100 Tensor Core GPU*. Accessed: 2026-05-19.

NVIDIA (2026a). *CUDA Toolkit*. Accessed: 2026-05-19.

NVIDIA (2026b). *Thrust: Parallel Algorithms Library*. Accessed: 2026-05-19.

Omelyan, I. P., I. M. Mryglod, and R. Folk (May 2002). „Optimized Verlet-like algorithms for molecular dynamics simulations“. In: *Phys. Rev. E* 65 (5), p. 056706.

Shi, Yuan (1996). „Reevaluating Amdahl's law and Gustafson's law“. In: *Computer Sciences Department, Temple University (MS: 38-24)*, p. 25.

Tee, Liong Seng, Sukehiro Gotoh, and Warren E Stewart (1966). „Molecular parameters for normal fluids. Lennard-Jones 12-6 Potential“. In: *Industrial & Engineering Chemistry Fundamentals* 5.3, pp. 356–363.

Williams, Samuel W, David Patterson, Leonid Oliker, John Shalf, and Katherine Yelick (2011). „The roofline model“. In: *Performance tuning of scientific applications* 20102662, pp. 195–215.

Winsberg, Eric (2022). „Computer Simulations in Science“. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta and Uri Nodelman. Winter 2022. Metaphysics Research Lab, Stanford University.

Input File Example

A

See next page.

```
1000
10.772173450159418
5001
0.0001
0.5386086725079708 0.5386086725079708 0.5386086725079708 -2.6892787678646375   -0.7284562290769571  -1.9866933152726298
0.5386086725079708 0.5386086725079708 1.6158260175239125 -1.2345264298959686   -1.4108121568719074   0.26553267746196585
0.5386086725079708 0.5386086725079708 2.6930433625398544  0.5848915822921762    1.4747929880144617   0.5278063169459057
0.5386086725079708 0.5386086725079708 3.7702607075557957  1.275414497217242    -0.05934992539925022  0.4343554292437296
0.5386086725079708 0.5386086725079708 4.847478052571738   0.005005557470931622 -2.4439263426910904   0.11112714230156942
0.5386086725079708 0.5386086725079708 5.92469539758768    0.3285690049800769    1.030929181549707    0.3085738713959297
0.5386086725079708 0.5386086725079708 7.001912742603621   1.5574701905498245    1.4725359108436369  -0.4340617675922834
0.5386086725079708 0.5386086725079708 8.079130087619562   0.713625045648512    -0.26348537930652444 -1.6270290275110013
0.5386086725079708 0.5386086725079708 9.156347432635505  -0.5231916544738336    0.2886509344445969  -0.13780766213455858
0.5386086725079708 0.5386086725079708 10.233564777651447 -1.741112631347618     0.04689289584390862 -1.3954458079010985
1.6158260175239125 0.5386086725079708 0.5386086725079708 -0.40508657705293993   2.142120572720949    1.4479534535799095
1.6158260175239125 0.5386086725079708 1.6158260175239125 -0.5377805085693292    0.43239488565655687  1.516661168779627
1.6158260175239125 0.5386086725079708 2.6930433625398544 -0.01934738022026012   0.08484569414247294 -0.7303197962320809
1.6158260175239125 0.5386086725079708 3.7702607075557957  0.1731037683956248    0.7016217363346084   0.07456901753506298
1.6158260175239125 0.5386086725079708 4.847478052571738   1.1033976999050807   -0.08866335874507948  0.23243582948914465
```

Thread Local Accumulation

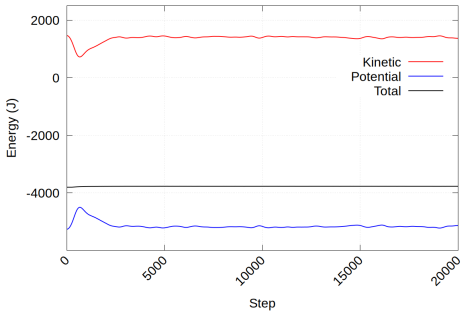
B

```
void compute_LJ(...) {
    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        int nthreads = omp_get_num_threads();

        // Allocate shared private arrays once
        static double *fx_all = NULL;
        static double *fy_all = NULL;
        static double *fz_all = NULL;

        #pragma omp single
        {
            fx_all = calloc(N * nthreads, sizeof(double));
            fy_all = calloc(N * nthreads, sizeof(double));
            fz_all = calloc(N * nthreads, sizeof(double));
        }

        double *fx_loc = fx_all + tid * N;
        double *fy_loc = fy_all + tid * N;
        double *fz_loc = fz_all + tid * N;

        #pragma omp for schedule(static)
        for (int k = 0; k < npairs; k++) {
            // ...
            if (r2 < rcut2) {
                // ...
                fx_loc[i] += fxij;
                fy_loc[i] += fyij;
                fz_loc[i] += fzij;

                fx_loc[j] -= fxij;
```

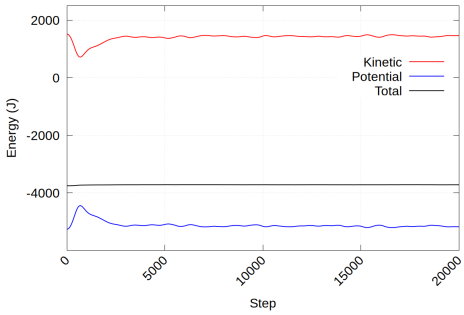
```
                fy_loc[j] -= fyij;
                fz_loc[j] -= fzij;
            }
        }
        #pragma omp barrier

        // Parallel reduction across threads
        #pragma omp for schedule(static)
        for (int i = 0; i < N; i++) {
            fx[i] = 0.0;
            fy[i] = 0.0;
            fz[i] = 0.0;
            for (int t = 0; t < nthreads; t++) {
                fx[i] += fx_all[t*N + i];
                fy[i] += fy_all[t*N + i];
                fz[i] += fz_all[t*N + i];
            }
        }
    }
}
```

C

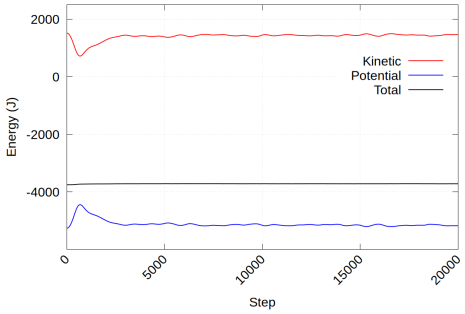
Energy Over Time



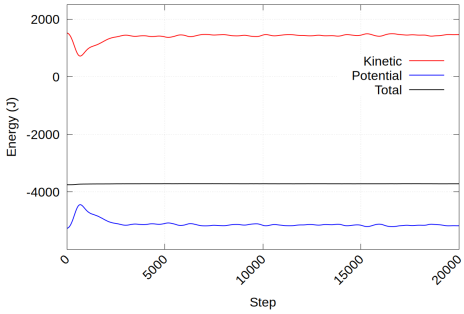
(a) C



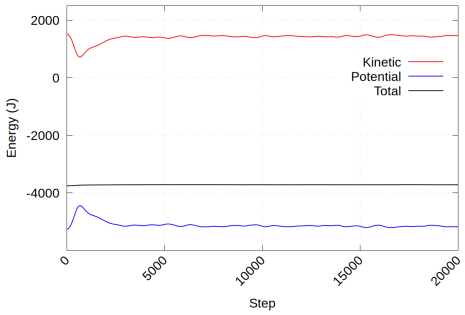
(b) OpenMP



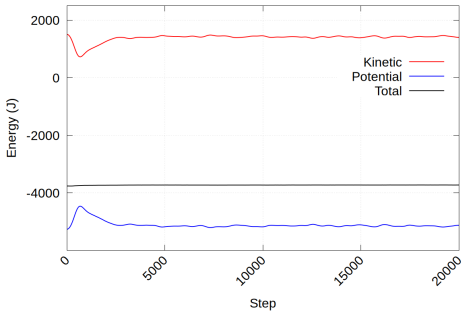
(c) CUDA



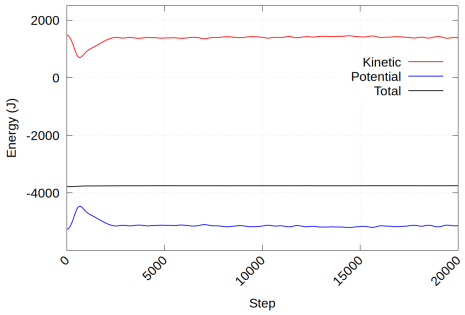
(d) NumPy



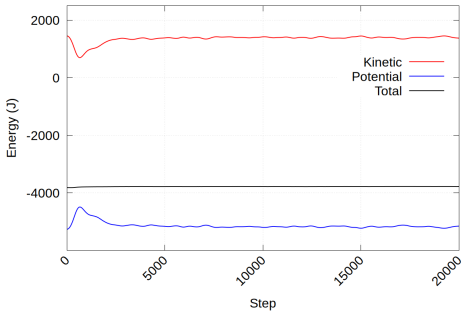
(e) JAX



(f) Futhark (C)



(g) Futhark (multicore)



(h) Futhark (CUDA)

Figure C.1.: Energy over time for all implementations.

End-to-End Runtimes

D

See next page.

N	C	OpenMP	CUDA	Numpy	JAX	Futhark-C	Futhark-multicore	Futhark-CUDA
10000	178.86	26.51	7.83	2044.25	34.78	488.15	95.41	13.88
25000	483.67	81.89	14.23	4949.18	48.35	1249.25	224.22	25.51
50000	959.26	195.35	22.59	10065.07	97.79	2359.02	389.03	51.30
75000	1352.13	275.48	27.88	17005.98	139.21	3340.00	552.15	69.50
100000	1913.45	383.82	38.67	20545.25	187.40	4690.05	744.12	97.67
1000000	18065.29	2914.20	336.25	Null	1811.95	45087.42	6542.64	836.24

Table D.1.: End-to-end runtimes measured in seconds.