SCHOOL OF COMPUTATION, INFORMATION
AND TECHNOLOGY - INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Vectorization of the Lennard-Jones Potential for Multi-Site Molecules in AutoPas

Qendrim Behrami

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Vectorization of the Lennard-Jones Potential for Multi-Site Molecules in AutoPas

Vektorisierung des Lennard-Jones Potenzials für Multi-Site Moleküle in AutoPas

Author:	Qendrim Behrami
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Samuel James Newcome, M.Sc.
Submission Date:	15.07.2023

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.07.2023 Qendrim Behrami

Acknowledgments

First of all, I want to express my sincere gratitude to my advisor, Sam. Thank you for allowing me to work on this thesis and for your thorough explanations and invaluable feedback. This work would not have been possible without your continuous support, and I genuinely appreciate the time working together.

I am also grateful to the Leibniz-Rechenzentrum for providing me with the computational resources to conduct my experiments and to the Technical University of Munich for providing me with the necessary knowledge to create this thesis. My special thanks go to the chair of Scientific Computing and Professor Dr. Hans-Joachim Bungartz for allowing me to work in this research area.

Lastly, I want to thank my close family and friends for their support. Thank you for your continuous encouragement and for giving me the perseverance to continue working on this project.

Abstract

Molecular dynamics simulations are a valuable computational tool used to study the behavior of complex molecular systems. Due to the large number of particles involved in real-world simulations, significant computational effort is required. Therefore, applying efficient parallelization techniques, such as vectorization, is crucial to perform large-scale simulations within a reasonable time frame. This thesis aims to demonstrate the impact of vectorization by presenting a vectorized implementation for calculating Lennard-Jones forces between particles with multiple interaction sites. Various vectorization techniques are explored, and extensive analysis is conducted to highlight the potential benefits of vectorization in molecular dynamics simulations.

Contents

1. Introduction

Molecular Dynamics (MD) simulations deal with the physical interactions of particles with each other. They simulate particle movement and are often employed when traditional algorithms are not powerful enough. One popular application area for MD simulations is the study of biological macromolecules, such as proteins [1] or polymers, to gain deep insights into the behavior of such molecules. Another standard research area is engineering, where MD simulations are utilized to study material properties. These simulations can unveil insights about the characteristics of the system, for example, to study the precise flow of a fluid through nanochannels [2] or to analyze the coalescence of argon droplets [3].

Despite the remarkable progress in computational sciences, MD simulations remain an area of continued interest and research. Continuous improvements to the computational capabilities of modern hardware, such as the steady increase in the number of cores on a processor and improvements to vector instructions, have been essential in increasing the performance of MD simulations and cleared the way for simulations with up to 20 trillion molecules [4].

However, while these vector instructions can tremendously benefit the computation time of MD simulations, they can also be challenging to implement. Although modern compilers can sometimes automatically vectorize programs, they usually fall short for more complex programs. For example, control structures such as conditional statements and loops are simple to implement in scalar code but often require alternative methods in vectorized programs. Thus, these shortcomings motivate the implementation of manually vectorized programs via special vector intrinsics.

This thesis explores the implementation of a standard force calculation in MD simulations, the Lennard-Jones potential, using vector intrinsics. While extensive research has already been done in this area [5, 6], it often focuses on molecules with only one interaction site. In contrast, this thesis addresses multi-site molecules, a more advanced molecule model that can increase the accuracy of the simulation but requires substantially more computational effort [7, p. 290]. The goal of this thesis is to showcase various vectorization concepts and strategies to efficiently vectorize the Lennard-Jones potential for these multi-site molecules and demonstrate how vectorization can greatly benefit MD simulations.

2. Theoretical Background

This chapter contains the necessary foundation for the following chapters. The first half introduces the theoretical fundamentals of MD, the Lennard-Jones potential, and multi-site molecules. The second half provides an overview of the algorithmic foundation, including particle containers, traversal methods, and vectorization.

2.1. Fundamentals of Molecular Dynamics

MD simulations aim to determine the movement of particles for a given time frame for an initial configuration of positions $\vec{x_i}$, velocities $\vec{v_i}$, and masses m_i. Time is divided into smaller time steps, and the simulation iteratively calculates the positions, velocities, and forces for subsequent time steps to propagate the system forward in time. In each iteration, these values are updated by solving Newton's equations of motion:

$$\frac{d\vec{x_i}}{dt} = \vec{v_i} \tag{2.1}$$

$$\frac{d\vec{v_i}}{dt} = \vec{a_i} \tag{2.2}$$

The acceleration of each particle depends on all of the forces acting on it and can be calculated with Newton's second law:

$$\vec{F_i} = m_i \cdot \vec{a_i} \tag{2.3}$$

As a result, it is necessary to determine the force acting on a particle in each time step. This is typically done by summing every force contribution of all possible particle pairs, but more general methods are possible (for example, by adding a constant external force like gravity):

$$\vec{F_i} = \sum_{i \neq j} \vec{F_{ij}} \tag{2.4}$$

It is common in molecular dynamics to define force potentials and calculate the force by negating the potential's gradient. Some basic potentials include the gravitational, Coulomb, and van der Waals potential [7, pp. 28–29].
Once the total force acting on a particle is computed, the resulting acceleration can be calculated according to 2.3. Then, the new velocity and position can be determined by appropriate numerical integration techniques, such as Verlet integration [8].

2.2. The Lennard-Jones Potential

The force potential used in this thesis is the **Lennard-Jones potential** [9]:

$$U_{ij} = 4\epsilon_{ij}\left(\left(\frac{\sigma_{ij}}{r_{ij}}\right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}}\right)^{6}\right) \tag{2.5}$$

The Lennard-Jones potential (Fig. 2.1) is used to model electrostatically neutral interactions and is either repulsive or attractive, depending on the distance r_{ij} between two particles. The parameter ϵ indicates the depth of the potential well. It controls the amplitude of the potential energy, while σ determines the zero-crossing of the potential and can be considered as the diameter of one particle.
Suppose both particles are from the same chemical type. In that case, these parameters are independent of the individual molecules and constant, but if the interaction is between chemically different types, special mixing rules must be applied. One such mixing rule is the Lorentz-Berthelot rule, which calculates the average of the individual parameters:

$$\epsilon_{ij} = \sqrt{\epsilon_i \epsilon_j} \qquad \sigma_{ij} = \frac{\sigma_i + \sigma_j}{2} \tag{2.6}$$

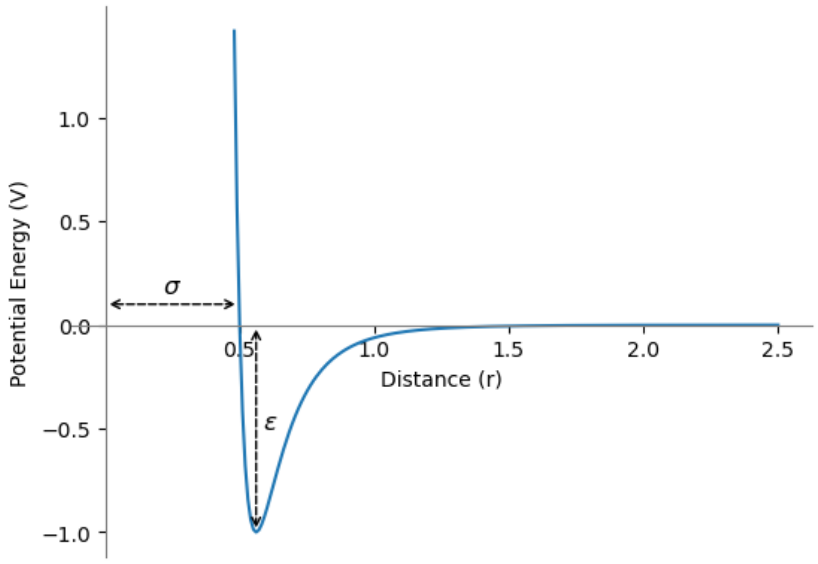


Figure 2.1.: Lennard Jones Potential ($\sigma = 0.5$, $\epsilon = 1.0$)

2.3. Multi-Site Molecules

Many existing MD simulators employ so-called single-site molecules [3, 10], which model particles as singular points in space. In contrast, there are also **multi-site** (or **multi-site)** molecules, which are composed of multiple interconnected points in space. These points are called "sites," and their connections are usually rigid. For molecules with multiple sites, the total force acting on the particle is calculated by iterating over each site and determining the force contribution with each site that is not part of the original particle, as illustrated in Fig. 2.2.

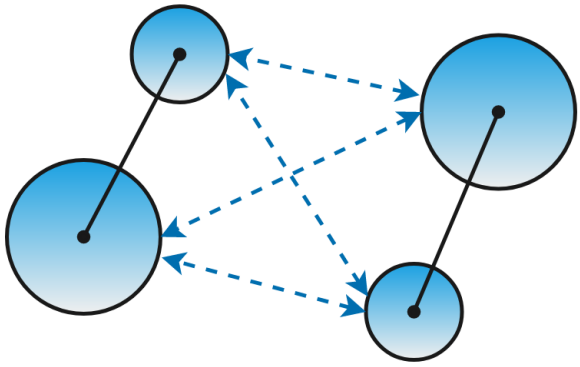


Figure 2.2.: Force interaction of multi-site molecules
Black lines represent the static connection of the particles, while blue arrows indicate the force interactions.

Multi-site molecules help increase the accuracy of the simulation [7, p. 290]. However, this increase comes with more computational cost since the number of force calculations scales quadratically with the number of sites per molecule. In addition, the force acting on a multi-site particle can induce a rotational motion, which makes torque calculation necessary:

$$\vec{\tau_i} = \sum_{k \in S_i} \vec{d_k} \times \vec{F_k} \tag{2.7}$$

The torque of a particle i is determined by summing over cross-products of every site, where $\vec{d_k}$ is the vector pointing from the center-of-mass to the k-th site and $\vec{F_k}$ is the force acting on the site. Then the angular velocity ω_i can be computed as follows:

$$\frac{d\vec{\omega_i}}{dt} = I_i^{-1}\vec{\tau_i} \tag{2.8}$$

where I_i is the moment of inertia of the current particle, which depends on various factors such as the shape and mass distribution of the object.

2.4. Newtons's third law

According to **Newton's third law (N3L)**, if two particles of a particle pair exert forces on each other, these forces have the same magnitude but opposite directions. This means that the force of site A upon site B is identical to the negated force of site B upon site A. As a result, it is sufficient to calculate the force for each pair only once, thus effectively halving the number of necessary computations.

2.5. AutoPas

There is a large variety of algorithms and data structures in molecular dynamics. However, as will be shown in the results chapter, no combination of algorithms and containers performs best in every scenario. As a result, the node-level auto-tuned simulation library **AutoPas** [11] has been developed, which aims to identify and select the best algorithm at run-time. AutoPas can achieve this with the *Full Search* strategy, which consists of tuning and non-tuning phases. In each tuning phase, the library tests every available combination of algorithms and containers for a fixed number of iterations. Then it selects the algorithm with the fastest mean time for the following non-tuning phase. Since the particles' positions, distribution, and homogeneity may alter over time, the optimal algorithm can change, so there are generally multiple tuning phases. The library offers a variety of data structures and parallelization schemes and has been successfully integrated into established MD simulators like ls1 MarDyn. [12]

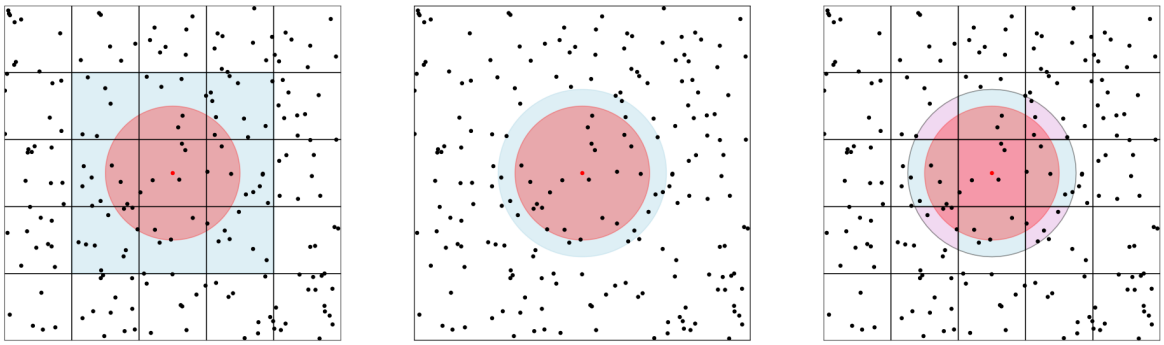
2.6. Particle Containers

2.6.1. The cutoff condition

The following containers build upon the **cutoff condition**, which is based on the observation that the Lennard-Jones force between two particles is effectively zero for large enough distances (Fig. 2.1). A standard method to utilize this observation is introducing a cutoff r_c. Then, the force calculation only occurs if the distance between the two particles is smaller than the cutoff; otherwise, the computation is omitted. Consequently, this method can significantly reduce the number of force computations for a negligible loss of accuracy.

2.6.2. Direct Sum

The direct sum method is the naive approach and stores particles in a standard container (for example, an array). The required distances for the force calculation are calculated for every possible particle pair, leading to $\mathcal{O}(n^2)$ distance evaluations for n particles. As a result, the direct sum method is only suitable for a small number of particles. Nevertheless, implementing it is simple and does not introduce additional memory overhead.



(a) Linked Cells (b) Verlet Lists (c) Pairwise Verlet Lists

Figure 2.3.: Overview of particle containers implemented in AutoPas
The red circle represents the cutoff sphere; blue areas denote the regions outside of the cutoff sphere for which distance calculations are performed. For Verlet and Pairwise Verlet Lists, the colored areas are part of the respective neighbor lists. The alternating blue and pink strips represent different neighbor lists. (Source: [13])

2.6.3. Linked Cells

The **Linked Cells (LC)** method performs a spatial division of the domain into cells, where each particle solely interacts with particles within the same cell and in adjacent cells (Fig. 2.3(a)). This can heavily diminish the number of distance checks, depending on the distribution of the particles, the cutoff, and the cell size. The container carries a relatively small memory and computational overhead since it must maintain the cell structure and check whether particles transfer into another cell after their respective position update.
However, the Linked Cells come with a considerable disadvantage. Suppose a uniform distribution of particles and a cell size that equals the cutoff radius. Since a molecule has to interact with all other molecules in its cell and the neighboring cell, the algorithm must evaluate the distances for all particles within a cube of volume $27r_c^3$. However, the cutoff sphere encircling this particle possesses a volume equivalent to $\frac{4}{3}\pi r_c^3$, which implies that only about $\frac{4}{81}\pi \approx 15.5\%$ of the distance evaluations result in a force calculation. This probability is called **hit rate** and shrinks further for larger cell sizes. In contrast, smaller cell sizes require more cell checks to cover the whole cutoff sphere, thus introducing more memory overhead. For these reasons, it is prevalent in MD to keep the cell size identical to the cutoff, although smaller cell sizes can affect performance positively. [14]

2.6.4. Verlet Lists

The **Verlet Lists (VL)** method is an alternative to Linked Cells and builds upon the observation that molecules typically move slowly between iterations. Consequently, the set of particles within a given molecule's cutoff sphere barely changes for subsequent iterations. Therefore, the

container creates a neighbor list for every particle (Fig. 2.3(b)), which contains every other particle within a sphere of radius $r_c + \Delta s$, where Δs is called the **Verlet skin**. In the following time steps, molecules solely interact with other molecules in their respective neighbor list. The container also rebuilds the neighbor list after N iterations to account for particles entering or exiting the extended cutoff sphere. Therefore, it is necessary to choose N and Δs such that no particle can move through the skin without the neighbor list being rebuilt.
The advantage of Verlet Lists is a typically higher hit rate than Linked Cells. This rate depends on the skin size and the given scenario. For example, a skin size of $\Delta s = 0.1 r_c$ results in a hit rate of:

$$\frac{\frac{4}{3}\pi r_c^3}{\frac{4}{3}\pi(r_c + \Delta s)^3} = \frac{r_c^3}{(1.1 r_c)^3} = \frac{1}{1.1^3} \approx 0.75 \tag{2.9}$$

This shows that Verlet Lists can considerably increase the hit rate compared to Linked Cells, resulting in fewer redundant distance checks. However, they usually carry a relatively high memory overhead since each particle stores a list of adjacent molecules. In addition, since the particles in a neighbor list are unordered, accessing these molecules may lead to more frequent cache misses and can thus slow down the memory operations.

2.6.5. Pairwise Verlet Lists

Gonnet [15] proposed **Pairwise Verlet Lists (PVL)** (Fig 2.3(c)), which combine both Linked Cells and Verlet Lists. Instead of storing all particles in one large container, the particle data is stored cell-wise as in the Linked Cells container. The container then builds a list of interacting cell pairs on top of the cell structure and creates Verlet Lists for each cell pair. Each Verlet List stores every particle pair that fulfills the cutoff condition.
The advantage of this container comes from the excellent data locality of Linked Cells and the small number of redundant distance checks of Verlet Lists. Furthermore, the cell structure enables the usage of cell-based traversal schemes, possibly leading to more efficient parallelization. Its disadvantage lies in the high memory overhead for maintaining a cell structure and neighbor lists, potentially leading to issues for sparse particle distributions.

2.7. Traversals

MD simulations utilize shared-memory parallelization to leverage the full capabilities of modern multi-core systems. Since this usually involves multiple threads updating force values simultaneously, great care is necessary to ensure thread safety and prevent race conditions. AutoPas implements shared-memory parallelization via the widely used OpenMP API and offers various methods for safely updating the forces [11], called **Cell Traversals**. The traversals implemented in AutoPas build upon one of these three base steps:

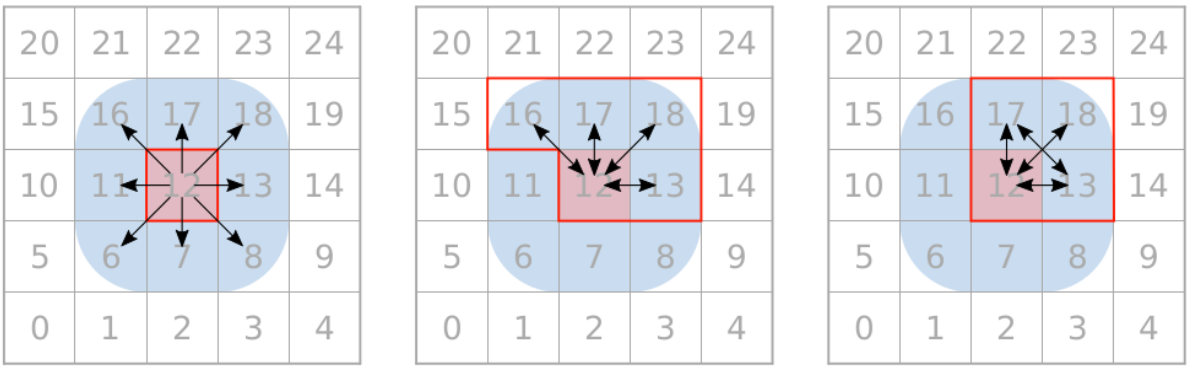
- **C01** The C01 step (Fig. 2.4(a)) lets every thread operate on a single cell simultaneously. The force calculation does not utilize N3L (2.4) for particles in different cells and only writes to particles within its assigned cell. This traversal does not require sophisticated synchronization mechanisms and is thus highly parallelizable and enables excellent load balancing.
  AutoPas offers a potential improvement to this traversal, which is called **combined C01**[1]. This modified method combines every cell of the interaction area into one big buffer. The force calculation occurs between all cells in this buffer, and when the base cell progresses, the buffer drops old cells and adds new cells depending on the cutoff sphere of the new base cell.

- **C18** The C18 step (Fig. 2.4(b)) aims to improve performance by including the N3L optimization. While the force calculation of the base cell depends on every surrounding cell within the cutoff radius, only the cells with a larger index are considered to prevent duplications. Since this step writes to multiple cells simultaneously, more cells must be synchronized to avoid race conditions, thus reducing the degree of parallelism. Nevertheless, including N3L can halve the number of force calculations and thus possibly outweigh the diminished parallelism.

- **C08** The C08 step (Fig. 2.4(c)) is similar to C18 but replaces one diagonal calculation (12 to 16 in the figure) with a forward diagonal calculation for an adjacent cell (13 to 17 in the figure). This results in a smaller parallelization area and reduces the amount of synchronization, thus increasing the degree of parallelism. In addition, the C08 traversal can reuse the cache more efficiently because of its smaller interaction area.

AutoPas implements these traversals for both Linked Cells and Pairwise Verlet Lists. The combinations of containers and traversals are described by the abbreviation of the container, followed by the traversal, and an optional suffix if the N3L optimization is used. For example, *LC-C01* refers to the C01-traversal for Linked Cells, while *PVL-C08-N3l* indicates a Pairwise Verlet List with C08-traversal that utilizes Newton's third law. Lastly, there is also the standard *VL*-method, which uses regular Verlet Lists and traverses the domain like the Direct Sum container and does not use N3L.

[1] Official AutoPas Documentation: C01 traversal



(a) c01 base step (b) c18 base step (c) c08 base step

Figure 2.4.: Overview of the three base traversal methods in AutoPas
The red cell marks the currently processed cell.
The blue area symbolizes the cutoff sphere (Here the cell size equals the cutoff).
The red box identifies the cells which must be shielded against race conditions.
(Source: [11])

2.8. Vectorization

Vectorization is another technique that enables parallel computing. In contrast to the previously mentioned parallelization via multiple threads, vectorization achieves higher parallelization by applying the same instruction on multiple data (SIMD) elements while operating on a single processing unit. Many modern processors implement vectorization via special vector registers containing up to 512 bits. Vectorized programs then pack multiple elements (typically 32- or 64-bit) in one such register and apply special vector instructions to every element simultaneously, thus reducing the overall amount of instructions (Fig. 2.5). The potential gain depends on the size of each data element and the available vector registers; for example, a program utilizing 256-bit registers with 64-bit floating-point numbers can apply one instruction to 4 elements, thus potentially quadrupling the program's performance. However, the actual speed-up of a vectorized program also depends on other factors, such as memory bandwidth and cache utilization, and is not necessarily identical to this factor.

2.8.1. Advanced Vector Extensions

Vector registers and instructions were not a part of the original x86 architecture and have instead been added over the years by supplementary extensions. For example, the **Advanced Vector Extensions(AVX)** introduced support for 256-bit vector registers, which AVX2 expanded through the introduction of advanced instructions such as *gather* operations. AVX512 further broadened these extensions by adding 512-bit vector registers and other instructions like *scatter* operations. However, due to the current limited availability of AVX512, this thesis only considers the instructions of the AVX2 set.

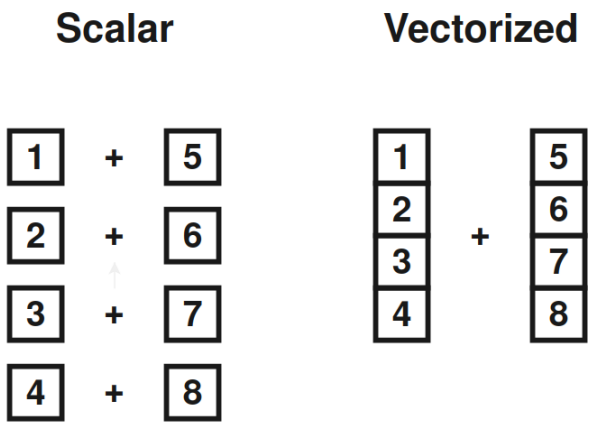


Figure 2.5.: Adding two arrays with a scalar and vectorized method
The scalar method requires four separate load and add instructions, as each value is loaded once and then individually added. In contrast, the vectorized method directly loads four values simultaneously and performs a single addition, resulting in only one load and add instruction.

2.8.2. Creation of vectorized code

There are two common ways for a developer to utilize AVX instructions [16]:

- **Autovectorization (AutoVec)**: The compiler may identify and automatically vectorize code segments. As a result, the developer can write standard scalar code and let the auto-vectorization do the rest. This makes the code easy to read and enables the code to compile on different processors without modification as long as the hardware supports vector instructions. However, the compiler can fail to identify or adequately vectorize code since it cannot guarantee that the vectorized code is correct. It can be challenging to make these guarantees since the compiler is usually not aware of every dependency between control structures, memory locations, and variables. To remedy this, OpenMP enables the programmer to annotate the program via *#pragma* directives to inform the compiler about additional properties of the program. However, there is still no guarantee that the compiler generates optimal vectorized code.
- **Intrinsics**: The C++ programming language supports the usage of special vector intrinsics, which can be included via special header files. They expose particular data types and functions to the developer to vectorize code and allow fine control while still being portable. However, this approach requires familiarity with the available intrinsics, and the resulting code is usually more challenging to read and maintain. Still, the potential performance gain can justify the additional effort.

2.8.3. Vectorization for multi-site molecules

The following snippet demonstrates a prototype scalar force calculation loop for multi-site particles in adjacent cells:

```
for (int i = 0; i < cellA.getNumberOfParticles(); ++i){
    auto molA = cellA.getMolecule(i);
    for (int j = 0; j < cellB.getNumberOfParticles(); ++j){
        auto molB = cellB.getMolecule(j);
        double distance = calculateDistance(molA,molB);
        if (distance < cutoffRadius){
            double force = calculateMultisiteForce(molA,molB);
            molA.force += force;
            molB.force -= force; // N3L
        }
    }
}
```

Listing 2.1: Scalar force calculation loop

This small segment already illustrates some of the commonly faced challenges when trying to vectorize a program [16]:

Nested loops

The first question is how to vectorize the multiple loops of the algorithm. One method is to only vectorize the inner loop (iterating over molecules in cell B). The outer loop is scalar, while for the inner loop, a chunk of particles is loaded, where the size of the chunk depends on the size of available vector registers and the size of the data.
This method loads one molecule from cell A and copies its values to each entry of a vector register, resulting in the register $\boxed{a_0 \mid a_0 \mid a_0 \mid a_0}$. Then, it loads four consecutive molecules from cell B into a register $\boxed{b_0 \mid b_1 \mid b_2 \mid b_3}$ and calculates the distances and forces via vector instructions, resulting in a register containing the force values $\boxed{f_{00} \mid f_{01} \mid f_{02} \mid f_{03}}$. These force values are then stored in a buffer for the first molecule, and if N3L is used, they are also stored in the respective memory locations of cell B with the sign reversed. This process is repeated until the force is computed for every particle in cell B. Then, the buffer of molecule A is horizontally added, resulting in the total force acting on particle A. This force is then updated in the appropriate memory location, and the outer loop continues with the next particle in cell A.
An alternative approach is to vectorize both loops by loading a chunk of molecules from each cell and repeatedly calculating the force for every combination of particles by permuting vector registers accordingly. While this method requires fewer load and store operations, it can be quite complex for multi-site molecules because there is no guarantee that every entry in a vector register belongs to the same molecule. This means that additional steps are necessary when permuting the particle chunks, which limits the gain in efficiency. For this reason, this thesis will only focus on the first method.

Divergence handling

The force in the scalar program is only computed if the distance between particles is smaller than the cutoff. However, the if-statement is impossible to vectorize, thus making alternative methods necessary:

- **Regular Masks** The regular masking method takes the two chunks $\boxed{a_0}\boxed{a_0}\boxed{a_0}\boxed{a_0}$ and $\boxed{b_0}\boxed{b_1}\boxed{b_2}\boxed{b_3}$ and calculates the distance. The distance register is then pair-wise compared against another register holding multiple copies of the cutoff. For example, if only the first two particles fulfill the cutoff condition, the resulting mask will look like $\boxed{1}\boxed{1}\boxed{0}\boxed{0}$. The computation is omitted if every entry equals zero; otherwise, the force must be computed for all four particles. However, before storing the force values, a logical and-operation between the force register and mask is necessary to prevent undesired force calculations.

- **Gather/Scatter Masks** This method is similar to the previous method and determines the distance between two chunks. However, instead of storing zeros and ones, it only stores the index of every other particle within the cutoff sphere in an array. The algorithm then loads the appropriate molecules according to the index register during the inner loop. For example the index register $\boxed{0}\boxed{2}\boxed{3}\boxed{5}$ results in a particle register $\boxed{b_0}\boxed{b_2}\boxed{b_3}\boxed{b_5}$. Since the relevant particles are not guaranteed to be stored contiguously, regular load operations are insufficient. They must be replaced by special **gather** instructions, which are part of AVX2, and load values from memory according to an additional index vector. Similarly, if Newton's third law is utilized, **scatter** instructions are required to store the force values according to the indices. For this reason, the term gather/scatter mask is used for this masking method.

The gather/scatter method can potentially prevent many unnecessary calculations compared to the regular masking approach but strictly requires gather/scatter intrinsics. While gather intrinsics are widely supported nowadays, scatter operations are only available in AVX512. It is possible to implement a custom scatter operation via scalar operations, which makes the gather-scatter approach viable but not optimal on processors without AVX512.

Distance calculation

Since molecules consist of multiple sites, there are three different variants to determine the distance between multi-site molecules:

- **Site To Site** The distance is calculated between the sites, resulting in one distance computation for each possible site pair. While this method is straightforward to vectorize, it needs to perform many calculations, which can possibly outweigh the gains of efficient vectorization.

```
for (const auto& siteA : sitesCellA){
    for (const auto& siteB : sitesCellB){
        double distance = calculateDistance(siteA,siteB);
        if (distance < cutoff){
            double force = calculateMultisiteForce(siteA,siteB);
            ...
        }
    }
}
```

Listing 2.2: Site to Site calculation

- **Center-of-mass to Center-of-mass (CTC)** In this variant, the cutoff condition is solely based on the center of mass of each molecule and thus independent of the relative site position. Before entering the force calculation loop, a molecular mask is created, which indicates whether a given molecule in cell B is within the cutoff sphere of the current molecule in cell A. The molecular mask is then extended to a site mask, which copies the result of the cutoff check for every site. This method introduces additional memory overhead for the molecular mask but requires fewer distance checks than the previous method.

```
for (const auto& molA : moleculesCellA){
    for (const auto& molB : moleculesCellB){
        double distance = calculateDistance(molA,molB);
        molMask[molB] = distance < cutoff;
    }
    siteMask = extendMolMask(molMask);
    for (const auto& siteA : molA){
        for (const auto& siteB : molB){
            if(siteMask[siteB]){
                double force = calculateMultisiteForce(siteA,siteB);
                ...
            }
        }
    }
}
```

Listing 2.3: Center to Center calculation

- **Center-of-mass to Site (CTS)** This method is an in-between solution to both previous methods. This time, the cutoff condition is evaluated between the center of mass of particle A and every site in cell B and its result is directly stored in the site mask. As a result, it is possible that this method only considers a subset of sites for one particle if its center of mass is close to the cutoff radius, while the previous method always includes or excludes the whole particle. This phenomenon is called a **partial interaction**.

```
for (const auto& molA : moleculesCellA){
    for (const auto& siteB : sitesCellB){
        double distance = calculateDistance(molA,siteB);
        siteMask[siteB] = distance < cutoff;
    }
    for (const auto& siteA : molA){
        for (const auto& siteB : sitesCellB){
            if(siteMask[siteB]){
                double force = calculateMultisiteForce(siteA,siteB);
                ...
            }
        }
    }
}
```

Listing 2.4: Center to Site calculation

2.8.4. Fundamentals of AVX intrinsics

AVX intrinsics provide data types and functions to enable vectorized processing. The data types correspond to vector registers; they are prefixed by `__m`, followed by their size and a suffix to indicate the type of its elements. For example, the data type `__m256d` refers to a vector register of 256 bits which contains double-precision floating-point numbers.
Vector operations follow a similar pattern. They use the prefix `_mm`, followed by the size, the name of the operation, and a suffix for the data type of the elements. For example, the following snippet illustrates a vectorized function to sum two vector containers:

```
// Assuming a.size() == b.size() and a.size() % 4 == 0
std::vector<double> addVectors(std::vector<double> a, std::vector<double> b) {
  std::vector<double> c(a.size());

  for (int i = 0; i < c.size(); i += 4) {
    // Load 4 unaligned (packed) doubles from a and b into vector registers
    __m256d a4 = _mm256_loadu_pd(&a[i]);
    __m256d b4 = _mm256_loadu_pd(&b[i]);
    // Add the 4 doubles
    __m256d c4 = _mm256_add_pd(a4, b4);
    // Store the values of c4 at c[i] to c[i+3]
    _mm256_storeu_pd(&c[i], c4);
  }
  return c;
}
```

Listing 2.5: Sum of vectors using AVX intrinsics

For a comprehensive overview of all available AVX intrinsics and a detailed explanation of their specific behavior, the Intel® Intrinsics Guide [17] offers comprehensive documentation.

2.8.5. Data Layouts

The data layout specifies how particle properties like position, velocity, or type are stored in memory. Figure 2.6 illustrates the two commonly used layouts:

- **Array of Structures (AoS)**: Molecules are modeled by structures/classes encapsulating the necessary attributes. These molecule objects are then stored contiguously in an array. Since standard vector load/store operations can only access contiguous memory, the AoS layout is generally unsuitable for vectorization.
- **Structure of Arrays (SoA)**: A structure manages multiple arrays for each particle property, i.e., one array for storing x-positions. The properties of the same particle are spread out in memory, but the same properties of multiple particles are stored contiguously. This enables the usage of vector load/store operations and makes SoAs thus well-suited for vectorization.

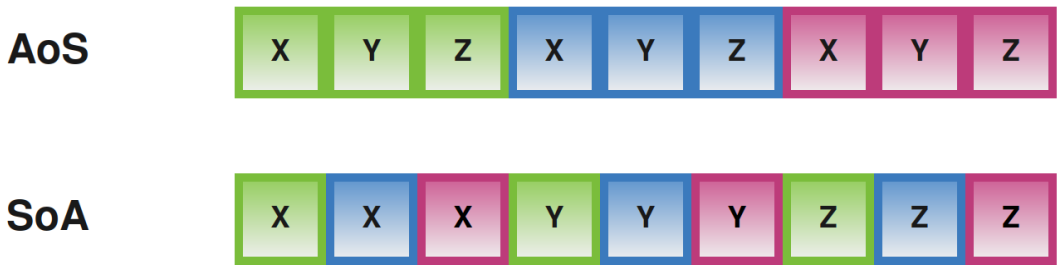


Figure 2.6.: Simplified physical memory layout of AoS and SoA
Letters represent particle attributes (i.e. positions), and colors distinguish particles.

3. Implementation

This chapter presents the implementation of the vectorized Lennard-Jones functor for multi-site molecules. AutoPas already provides functors for multi-site molecules, but since they utilize auto-vectorization, new functors have been developed.

3.1. Overview of relevant classes

The following classes have already been a part of AutoPas (unless otherwise specified) and are essential for understanding the following sections.

3.1.1. MoleculeLJ

This class manages molecules with single sites and must not be used with multi-site force functors. It stores particles in the AoS data layout (see 2.8.5) and manages attributes like position, velocity, and force. In addition, it contains a type ID, which is used to look up molecule-specific data. Lastly, it has an attribute for ownership. Since the space of a simulation is limited, AutoPas needs to handle particles moving outside of the boundaries. Standard methods for dealing with boundary particles are reflective boundaries and periodic boundaries, which insert particles at the opposing boundary. AutoPas utilizes *dummy* and *halo* particles to implement boundary conditions and uses the ownership attribute to distinguish them from standard particles. The functors must not calculate the force for dummy particles, so an additional ownership check is necessary.

3.1.2. MultisiteMoleculeLJ

This class is based on MoleculeLJ and is designed explicitly for multi-site molecules. Since multi-site particles experience rotational motion (see 2.3), this class additionally stores a quaternion for angular direction and arrays for angular velocity and torque. Multi-site particles possess a position attribute like single-site molecules to identify their center-of-mass and **relative site positions**, which specify the position of the molecules' sites relative to their center-of-mass. Both positions can be combined into **exact site positions**, which specify the absolute location of a site in the domain.

3.1.3. ParticlePropertiesLibrary

This library stores the physical properties of all molecules. It is primarily used to determine the mixing parameters of the Lennard-Jones force calculation (see 2.2) and obtain the relative site positions of a particle via its type ID.

3.1.4. LJMultisiteFunctor

This class contains multiple functors for calculating the Lennard-Jones potential. It provides one functor for a scalar AoS force calculation, which requires two particles as parameters. Since only two particles are involved, this function does not offer many opportunities for vectorization; therefore, only the SoA function supports auto-vectorization. There are three SoA functors:

- `void SoAFunctorSingle(SoAView<SoAArraysType> soa, bool newton3)`

  This function is used for the Linked Cells particle container, takes one SoA of particles, and calculates the forces for all particles within the same cell. Since the N3L optimization is always applied for particles within the same cell, the second parameter is not used for this force calculation.

- ```
  void SoAFunctorPair(SoAView<SoAArraysType> soa1,
          SoAView<SoAArraysType> soa2, bool newton3)
  ```

  This function is also required for the Linked Cells container but uses two different SoAs instead. As such, this functor is used to calculate the Lennard-Jones force for adjacent cells. The *newton3* parameter indicates whether the N3L optimization should be applied and depends on the traversal method and the user configuration.

- ```
  void SoAFunctorVerlet(autopas::SoAView<SoAArraysType> soa,
          const size_t indexFirst, const std::vector<size_t,
          autopas::AlignedAllocator<size_t>> &neighborList, bool newton3)
  ```

  The Verlet List containers use this functor. It takes one SoA containing all particles, the index of the currently selected particle, and a list of indices of potential neighbor particles. Again there is a parameter to control the N3L optimization.

In addition to the Lennard-Jones force calculation, these functors also calculate and update the torque of each particle (see section 2.3). In addition, they can calculate global physical values, such as the potential energy and the virial sum. The user can control this behavior, which is determined at compile-time via class template parameters.

3.1.5. LJMultisiteFunctorAVX

This is a new class which contains all of the following functors. It is based on the **LJMultisiteFunctor** class and shares most of its attributes and functions but uses AVX intrinsics instead of auto-vectorization. In addition, there are two boolean class attributes, called `useRegularSiteMasks` and `useCTC`. The first variable indicates whether the functors should use regular or gather/scatter site mask (see section 2.8.3). At the same time, the latter determines if the cutoff condition is using the CTC or CTS method (see section 2.8.3). These boolean values are currently compile-time constants and apply to every SoA functor.

3.1.6. AVXUtils

This new class contains a few additional custom utility functions to reduce code duplication and increase readability. The first such function is:

```
inline double horizontalSum(const __m256d &data);
```

It takes a vector register of four 64-bit double-precision floating-point numbers (doubles) and returns the sum of all entries. The remaining functions are memory operations that can conditionally load/store different data types like this:

```
inline __m256d load_pd(bool useMask, const double *const __restrict data, const
    __m256i &mask) {
  if (useMask) {
    return _mm256_maskload_pd(data, mask);
  } else {
    return _mm256_loadu_pd(data);
  }
}
```

The function takes a pointer to a memory location containing doubles. If the *useMask* flag is set, the function loads values according to a mask, or it returns all four doubles. The methods use so-called unaligned memory operations. Aligned AVX memory operations require the data to be 256-bit (or 32-Byte) aligned, which essentially means that the memory address is divisible by 32. Since the main force calculation contains random access memory patterns, additional operations would be required to guarantee this alignment, and the additional overhead to ensure correct alignment would likely outweigh the gain. This is why only unaligned memory operations are used, despite a possible loss in performance.

3.2. Gathering particle attributes

While there are three different functors, they all obtain the necessary particle properties in the same two steps:

1) Creating particle property pointers Since the functors operate on the SoA layout, each particle attribute is stored in its own array. The functor begins by creating pointers to these arrays for each attribute, like this:

```
const auto *const xptr = soa.template begin<Particle::AttributeNames::posX>();
const auto *const yptr = soa.template begin<Particle::AttributeNames::posY>();
const auto *const zptr = soa.template begin<Particle::AttributeNames::posZ>();
```

2) Creating auxiliary vectors The pointers above only point to the center-of-masses, but the functor requires additional information about the exact site positions. Since the particle library only contains the relative site positions, the functor must create one *exactSitePosition* vector for each dimension and determine the exact positions by itself. This process involves quaternion operations to obtain the rotated site positions, which are then added to the center

of mass, resulting in the exact site positions. Furthermore, the functor requires buffers for the force acting on each site to calculate the torque according to equation 2.7. This part of the functor is not vectorized and can occupy a considerable portion of the functors' run-time.

3.3. The site mask

The three functors support both site mask types: regular masks and gather/scatter masks. In addition, they also support the CTC and CTS distance evaluation. As a result, there are four different combinations for creating site masks.
The Linked Cells' functors loop over each particle in the first cell. Each particle in this loop is referred to as the primary particle, and the site mask is built once in every iteration. This loop does not exist for the Verlet functor since the primary particle is already part of the function's signature (see section 3.1.4). There are two functions for building the site mask, one for the CTC variant and one for the CTS method.

3.3.1. The Center-To-Center site mask

This is the default mask and is illustrated in Listing 3.1. The signature consists of the following arguments:

- **buildMask:** This parameter indicates the type of the site mask and determines if it stores either zeros and ones or the site indices.
- **indices:** This vector contains the indices of all molecules potentially within the current particle's cutoff sphere. For example, when the Verlet functor calls this function, it contains all entries of the current neighbor list.
- **xptr,yptr,zptr:** These parameters point to the center-of-mass of each particle for each dimension.
- **typeptr:** This parameter contains the types of all particles and is necessary for determining the number of sites of each molecule.
- **ownedStatePtr:** It contains the ownership state of every particle. This is important because dummy particles must not appear in the site mask, even if they are within the cutoff range.
- **centerOfMass:** This is the position of the primary particle in world space.
- **offset:** This parameter is unused in the CTC variant and defaults to zero.

```
1 std::vector<size_t> buildSiteMaskCTC(bool buildMask, const std::vector<size_t>
      &indices, const double *const xptr, const double *const yptr, const double
      *const zptr, const size_t *const typeptr, const autopas::OwnershipState *const
      ownedStatePtr, const std::array<double, 3> &centerOfMass, size_t offset) {
2   std::vector<size_t> siteMask;
```

```
3   size_t siteIndex = 0;
4
5   for (const auto molIndex : indices) {
6     const size_t siteCount = useMixing ? _PPLibrary->getNumSites(typeptr[molIndex]);
7
8     // load secondary particle position
9     const double xPosB = xptr[molIndex];
10    const double yPosB = yptr[molIndex];
11    const double zPosB = zptr[molIndex];
12
13    // determine distance between center of masses
14    const double distanceSquaredCoM = calculateDistance(centerOfMass, xposB, yposB,
          zposB);
15
16    // evaluate cutoff and ownership condition
17    const bool cutoffCondition = distanceSquaredCoM <= _cutoffSquaredAoS;
18    const bool dummyCondition = ownedStatePtr[molIndex] != OwnershipState::dummy;
19    const bool condition = cutoffCondition and dummyCondition;
20
21    // store the result for each site of the secondary particle
22    for (size_t siteB = 0; siteB < siteCount; ++siteB) {
23      if (buildMask) {
24        const size_t mask = condition ? -1 : 0;
25        siteMask.emplace_back(mask);
26      } else if (condition) {
27        siteMask.emplace_back(siteIndex++);
28      } else {
29        siteIndex += siteCount;
30        break;
31      }
32    }
33  }
34  return siteMask;
35 }
```

Listing 3.1: CTC site mask creation

The function first creates a container to store the result of the cutoff and ownership check. Then it loops over each molecule index and calculates the Euclidean distance to the primary particle. The process continues by checking whether the distance is smaller than the cutoff and whether the particle is actually owned. Finally, the function loops over each site of the secondary particle and updates the site mask container according to the two previous checks. After repeating this procedure for every secondary particle, the method returns the site mask, which now identifies all owned particles within the cutoff radius of the primary particle.
This function is based on Listing 2.3 but does not use a molecular mask. While creating a molecular mask is straightforward to implement and vectorize, it can introduce a considerable memory overhead, which this implementation avoids at the cost of solely using scalar instructions.

3.3.2. The Center-To-Site site mask

This alternative method is used when the `useCTC` flag is set to false and is illustrated in Listing 3.2. While the signature is identical (apart from the name) to the previous method, the arguments have slightly different meanings:

- **buildMask:** This parameter has the same meaning as in the CTC functor.
- **indices:** This container manages the indices of all **sites** that are potentially within the cutoff sphere.
- **xptr, yptr, zptr:** These parameters point to the exact position of each **site** in world space for each dimension.
- **typeptr:** This parameter is not used in this function.
- **ownedStatePtr:** It contains the ownership state of every **site**. It is necessary to identify owned sites correctly.
- **centerOfMass:** This is the position of the primary particle in world space.
- **offset:** This parameter is only required for the Linked cell functor, which operates on particles within the same cell. Since all particles share the same SoA, the function requires the offset to identify the index of the first site after the primary particle. Unless explicitly specified, this parameter defaults to zero.

The function operates quite differently compared to the previous method. First of all, the method loops directly over all sites instead of all particles. The loop considers as many entries as can fit into a vector register to utilize vectorization fully. If there are not enough elements left, the method still uses vector registers but creates a *remainderMask* to identify the valid entries of the register. In the remainder case, the function selects the appropriate mask from a static array, depending on the number of current elements.
The function then uses AVX intrinsics to load the exact site positions and calculate the primary particle's distance to up to 4 neighboring molecules by computing the Euclidean distance between the center of mass of the primary particle and the respective exact site positions. Then the function determines the cutoff and ownership condition and stores the result in the site mask. Since gather/scatter site masks only contain values for particles within the cutoff radius, the method must store the appropriate values in a scalar fashion. In contrast, regular masks can use vectorized store operations without restrictions.
While this method is vectorized, the number of distance computations depends on the total site count, whereas this number only depends on the particle count for the CTC approach. Therefore, the CTS method typically requires more computational effort and is thus not guaranteed to run faster despite vectorization, which will be subject to further investigation in the analysis chapter.

```
1  std::vector<size_t> buildSiteMaskCTS(bool buildMask, const std::vector<size_t>
       &indices, const double *const xptr, const double *const yptr, const double
       *const zptr, const size_t *const typeptr, const autopas::OwnershipState *const
       ownedStatePtr, const std::array<double, 3> &centerOfMass, size_t offset) {
2      std::vector<size_t> siteMask;
3
4      for (size_t index = 0; index < indices.size(); index += vecLength) {
5        // handle the remainder case
6        const size_t remainder = indices.size() - index;
7        const bool remainderCase = remainder < vecLength;
8        const __m256i remainderMask = remainderCase ? _masks[remainder - 1] :
             _mm256_set1_epi64x(-1);
9
10       const size_t siteIndex = indices[index];
11
12       // Load site positions
13       const __m256d xposB = autopas::utils::avx::load_pd(remainderCase, &xptr[offset
             + siteIndex], remainderMask);
14       const __m256d yposB = autopas::utils::avx::load_pd(remainderCase, &yptr[offset
             + siteIndex], remainderMask);
15       const __m256d zposB = autopas::utils::avx::load_pd(remainderCase, &zptr[offset
             + siteIndex], remainderMask);
16
17       // calculate displacement
18       const __m256d distanceSquaredCoM = calculateDistance(centerOfMass, xposB,
             yposB, zposB);
19
20       // Evaluate the cutoff and ownership condition
21       const __m256d cutoffCondition = _mm256_cmp_pd(distanceSquaredCoM,
             _cutoffSquared, _CMP_LE_OQ);
22       const __m256i ownedState = autopas::utils::avx::load_epi64(remainderCase,
             ownedStatePtr+siteIndex, remainderMask);
23       const __m256d dummyMask = _mm256_cmp_pd(ownedState, _zero, _CMP_NEQ_UQ);
24       const __m256i condition = _mm256_and_pd(cutoffCondition, dummyMask);
25
26       // Store result
27       if (buildMask) {
28         autopas::utils::avx::store_epi64(remainderCase,&siteMask[index],remainderMask,condition);
29       } else {
30         for (size_t i = 0; i < std::min(vecLength, remainder); i++) {
31           if (condition[i] != 0) siteMask.push_back(siteIndex+i);
32         }
33       }
34     }
35     return siteMask;
36   }
```

Listing 3.2: CTS site mask calculation

3.4. The force kernel

After setting the site mask up, the functors prepare to invoke the force kernel. They first create accumulators for the force and torque of the primary particle and then loop through every site of the primary particle. In each iteration, the functor determines the exact position of the current site and then calls the force kernel.

3.4.1. Signature

```
template <bool newton3, bool calculateTorque, bool regularMasks>
void SoAKernel(const std::vector<size_t> &siteMask, const std::vector<size_t>
    &siteTypesB, const std::array<std::vector<double>,3> &exactSitePositions,
    std::array<std::vector<double>,3> &siteForces, const std::vector<size_t>
    &siteOwnership, const autopas::OwnershipState ownedStateA, const size_t
    siteTypeA, const std::array<double, 3> exactSitePositionA, const
    std::array<double, 3> rotatedSitePositionA, std::array<double, 3>
    &forceAccumulator, std::array<double, 3> &torqueAccumulator, double
    &potentialEnergyAccumulator, std::array<double, 3> &virialAccumulator, size_t
    offset = 0);
```

Listing 3.3: Force Kernel signature

The signature expects three template parameters, determining whether N3L is used and whether the torque should be computed. The third parameter indicates whether the site mask operates on zeros and ones or site indices.
The function then requires the site mask and various information about each site in the SoA, namely the site type, position (for each dimension), and ownership. Similarly, it expects the same data for the primary particle and requires the rotated site position for calculating the torque. In addition, there are *siteForce* containers to buffer each calculated site force, which is necessary for the N3L optimization and torque computation. Lastly, the function requires the accumulators for the force, torque, and global values and an optional offset.

3.4.2. General structure

The functor first sets up additional registers for holding the mixing data and the broadcasted site position of the primary particle. Then it enters a vectorized loop, which is illustrated in Listing 3.4.
The loop iterates over each entry of the site mask and concurrently processes as many elements as can fit into a vector register. The remainder case is handled identically as in the CTS site mask function, and then the current entries of the site mask are loaded with the help of a custom utility function. In the next step, the functor must determine the appropriate mixing parameters of the Lennard-Jones potential and load the exact site positions. Next, the kernel computes the distance between the primary particle and all secondary molecules. With the displacements, it is finally possible to determine the force. The function then continues by optionally performing the N3L optimization, the torque calculation, and the computation of

the global values. Finally, the accumulators of the primary particle and the global values are updated accordingly.

```
for (size_t siteIndex = 0; siteIndex < siteMask.size(); siteIndex += vecLength) {
      const size_t remainder = siteMask.size() - siteIndex;
      const bool remainderCase = remainder < vecLength;
      const __m256i remainderMask = remainderCase ? _masks[remainder - 1] :
          _mm256_set1_epi64x(-1);

      __m256i sites = autopas::utils::avx::load_epi64(remainderCase,
          &siteMask[siteIndex], remainderMask);

      load_mixing_data();
      load_exact_site_positions();

      calculate_displacement();
      calculate_force();

      if (calculateTorque) calculate_torque();
      if (newton3) apply_newton3();
      if (calculateGlobals) calculate_globals();

      update_accumulators();
}
```

Listing 3.4: Force Kernel structure

3.4.3. Obtaining the mixing data

Before calculating the Lennard-Jones potential, the functor requires the appropriate mixing values, which the *ParticlePropertiesLibrary* provides. This class internally manages a container consisting of multiple structs holding the σ- and ϵ-values values and a so-called shifting parameter, which is only relevant for calculating the global values.
First, the functor creates a pointer to the start of the mixing data of the primary particle. The method then continues by loading the types of the current sites. Since the gather/scatter mask only continues values within the cutoff range, it must use gather intrinsics here, while the regular masks require standard load instructions. Independent of the particular site mask type, the load operation can result in a non-contiguous type register, for example, $\boxed{0}\boxed{1}\boxed{0}\boxed{2}$. In the next step, each entry must be multiplied by the size of the mixing structs, three, to obtain valid offsets to the mixing data, i.e., $\boxed{0}\boxed{3}\boxed{0}\boxed{6}$. Finally, the mixing registers are populated via *gather*-intrinsics, which require a pointer to a data location, an index vector, and the size of one element. If the primary particle was of type 1, the mixing registers in the example now contain $\boxed{\sigma_{10}}\boxed{\sigma_{11}}\boxed{\sigma_{10}}\boxed{\sigma_{12}}$ and $\boxed{\epsilon_{10}}\boxed{\epsilon_{11}}\boxed{\epsilon_{10}}\boxed{\epsilon_{12}}$.
It should be noted here that the auto-vectorized functors use another strategy to determine these parameters. Therefore, any performance difference between the two multi-site functor classes is partially caused by this discrepancy.

```
const double *const mixingPtr = _PPLibrary->getMixingDataPtr(siteTypeA, 0);
__m256i siteTypeIndices = regularMasks?
    autopas::utils::avx::load_epi64(remainderCase, &siteTypesB[siteVectorIndex +
        offset], remainderMask):
    autopas::utils::avx::gather_epi64(remainderCase, &siteTypesB[offset], sites,
        remainderMask);
__m256i siteTypeIndices = _mm256_mul_epu32(siteTypeIndices, _mm256_set1_epi64x(3));

__m256d epsilon24 = _mm256_i64gather_pd(mixingPtr, siteTypeIndices, 8);
__m256d sigmaSquared = _mm256_i64gather_pd(mixingPtr + 1, siteTypeIndices, 8);
__m256d shift6 = _mm256_i64gather_pd(mixingPtr + 2, siteTypeIndices, 8);
```

Listing 3.5: Gathering mixing data

3.4.4. Calculation of force and torque

The last step before the force calculation is to load the appropriate exact site positions into vector registers. There are three registers, one for each dimension, which are also populated by either regular load instructions or gather functions, depending on the site mask type.
Listing 3.6 illustrates the force calculation process. First, the kernel computes the squared Euclidean distance between the exact site positions via arithmetic intrinsics. The distance must be inverted since it only occurs in the denominator of the Lennard-Jones potential (see 2.5). Then the functor continues to compute the potential with several vector instructions. Here, various intermediate results are temporarily stored to avoid redundant calculations. The next step is to apply a logical *and*-operations to the result with the cutoff mask to ensure that only valid entries occur. This step is skipped for gather/scatter masks since they already only contain elements within the cutoff range. Finally, a second and-operation is required for correctly handling the remainder case. The function proceeds to compute the torque (if the template parameter is set). This computation utilizes special *Fused multiply-add* operations that combine a multiplication with an addition, which can speed up calculation and also increase the accuracy of the results.

3.4.5. Application of N3L

If N3l is activated, the functor proceeds to load the current force values from the *siteForce*-buffers, subtracts the calculated force, and conditionally stores the updated value back into the buffer. Again, standard load operations are used for the regular mask, while gather/scatter masks require both gather and scatter instructions. Since scatter intrinsics are unavailable for AVX2, the current implementation only uses a sequential write and is therefore not optimal.

```
const __m256d displacementX = _mm256_sub_pd(exactSitePositionsAX,
    exactSitePositionsBX);
const __m256d displacementY = _mm256_sub_pd(exactSitePositionsAY,
    exactSitePositionsBY);
const __m256d displacementZ = _mm256_sub_pd(exactSitePositionsAZ,
    exactSitePositionsBZ);

const __m256d distanceSquaredX = _mm256_mul_pd(displacementX, displacementX);
const __m256d distanceSquaredY = _mm256_mul_pd(displacementY, displacementY);
const __m256d distanceSquaredZ = _mm256_mul_pd(displacementZ, displacementZ);

const __m256d distanceSquared = _mm256_add_pd(distanceSquaredX,
    _mm256_add_pd(distanceSquaredY, distanceSquaredZ));

const __m256d invDistSquared = _mm256_div_pd(_one, distanceSquared);
const __m256d lj2 = _mm256_mul_pd(sigmaSquared, invDistSquared);
const __m256d lj6 = _mm256_mul_pd(_mm256_mul_pd(lj2, lj2), lj2);
const __m256d lj12 = _mm256_mul_pd(lj6, lj6);
const __m256d lj12m6 = _mm256_sub_pd(lj12, lj6);
const __m256d scalar = _mm256_mul_pd(epsilon24, _mm256_mul_pd(_mm256_add_pd(lj12,
    lj12m6), invDistSquared));
__m256d scalarMultiple = regularMasks ? _mm256_and_pd(sites, scalar) : scalar;
scalarMultiple = _mm256_and_pd(_mm256_castsi256_pd(remainderMask), scalarMultiple);

const __m256d forceX = _mm256_mul_pd(scalarMultiple, displacementX);
const __m256d forceY = _mm256_mul_pd(scalarMultiple, displacementY);
const __m256d forceZ = _mm256_mul_pd(scalarMultiple, displacementZ);

const __m256d torqueAX = _mm256_fmsub_pd(rotatedSitePositionsAY, forceZ,
    _mm256_mul_pd(rotatedSitePositionsAZ, forceY));
const __m256d torqueAY = _mm256_fmsub_pd(rotatedSitePositionsAZ, forceX,
    _mm256_mul_pd(rotatedSitePositionsAX, forceZ));
const __m256d torqueAZ = _mm256_fmsub_pd(rotatedSitePositionsAX, forceY,
    _mm256_mul_pd(rotatedSitePositionsAY, forceX));
```

Listing 3.6: Main force calculation

3.4.6. Computation of global values

The calculation of additional physical properties only occurs if it was enabled during compile-time and is illustrated in Listing 3.7.
First, the function calculates the virial values by multiplying the force and displacement for each dimension. Then it calculates the potential energy using a fused operation and applies the current site mask if regular masking was used, in addition to the remainder mask. The function proceeds to determine the energy factor, which depends on the ownership of the current particles. For this purpose, the blending intrinsic sets each entry in the energy factor to either zero or one, depending on the ownership of the corresponding particle. Finally, the

masked potential energy and virial values are multiplied by this energy factor and added to the accumulators, again by fused multiply-add operations.

```
const __m256d virialX = _mm256_mul_pd(displacementX, forceX);
const __m256d virialY = _mm256_mul_pd(displacementY, forceY);
const __m256d virialZ = _mm256_mul_pd(displacementZ, forceZ);

const __m256d potentialEnergy6 = _mm256_fmadd_pd(epsilon24, lj12m6, shift6);
__m256d potentialEnergy6Masked = masked ? _mm256_and_pd(sites, potentialEnergy6) :
    potentialEnergy6;
potentialEnergy6Masked = _mm256_and_pd(remainderMask, potentialEnergy6Masked);

__m256i ownedStateA4 = _mm256_set1_epi64x(ownedStateA);
__m256d ownedMaskA = _mm256_cmp_pd(ownedStateA4, _ownedStateOwned, _CMP_EQ_UQ);
__m256d energyFactor = _mm256_blendv_pd(_zero, _one, ownedMaskA);

if constexpr (newton3) {
  const __m256i ownedStateB = regularMasks ?
      autopas::utils::avx::load_epi64(remainderCase, &siteOwnership[offset +
      siteVectorIndex], remainderMask) :
      autopas::utils::avx::gather_epi64(remainderCase, &siteOwnership[offset],
      sites, remainderMask);
  __m256d ownedMaskB = _mm256_cmp_pd(ownedStateB, _ownedStateOwned, _CMP_EQ_UQ);
  energyFactor = _mm256_add_pd(energyFactor, _mm256_blendv_pd(_zero, _one,
      ownedMaskB));
}

potentialEnergySum = _mm256_fmadd_pd(energyFactor, potentialEnergy6Masked,
    potentialEnergySum);
virialSumX = _mm256_fmadd_pd(energyFactor, virialX, virialSumX);
virialSumY = _mm256_fmadd_pd(energyFactor, virialY, virialSumY);
virialSumZ = _mm256_fmadd_pd(energyFactor, virialZ, virialSumZ);
```

Listing 3.7: Global values calculation

3.5. Reduction

The reduction phase is the final step of the functor and happens after the force kernel has been invoked for every site and every primary particle. The functor loops over the *siteForce*-buffers, which were populated by the force kernel, and calculates the torque for each site. Similarly to the primary particle, the loop accumulates the forces and torques for each molecule and stores them in the SoA at the end of every iteration. In the end, the SoA contains all updated force and torque values after the loop completes.

4. Comparison and Analysis

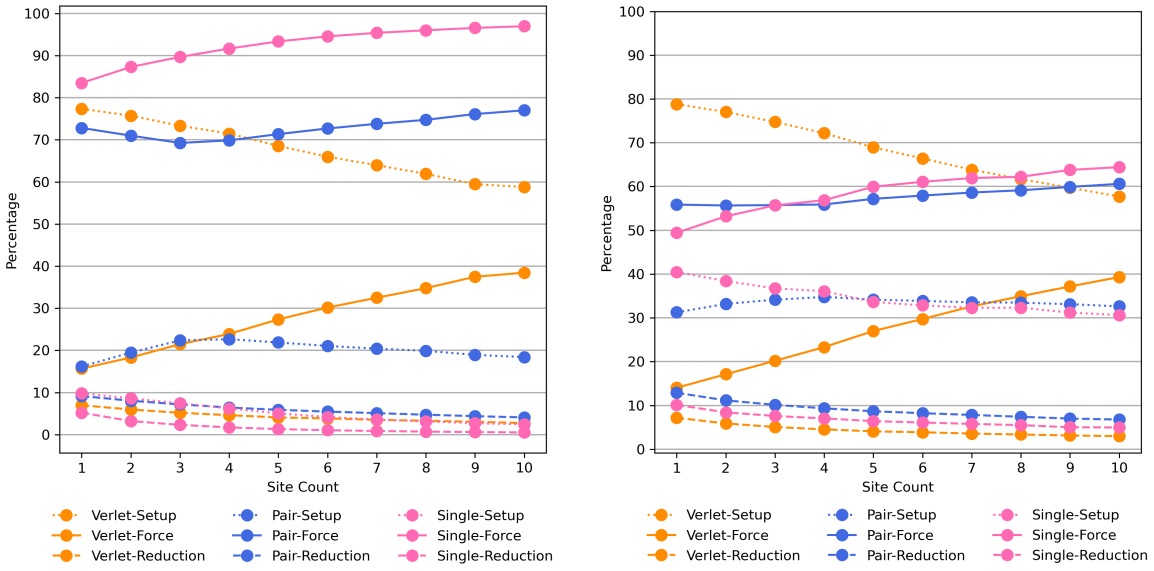
This chapter compares each previously discussed functor variant and discusses the potential performance gain of vectorization. As previous studies([12], [13]) have shown before, the best algorithmic choice for a given simulation depends on various factors, such as particle density and homogeneity. Therefore, the comparisons contain various combinations of particle containers (section 2.6) and traversals (section 2.7) for two different particle distributions. Unless otherwise specified, the **Uniform** scenario consists of a box of 10^5 uniformly distributed particles, and the **Gaussian** distribution contains 2.5×10^4 particles around the center of the domain with a standard deviation of 5. Both scenarios contain particles in a $50 \times 50 \times 50$ domain and use periodic boundary conditions with a cutoff of $r_c = 3$ and a Verlet skin of $\Delta s = 0.667$.
The run times were measured via AutoPas' tuning mechanism, where each available combination of container and traversal was tested for ten iterations and up to 10 sites, and the average wall time was taken. Furthermore, these measurements were conducted on two different machines: a personal workstation and the CoolMUC-2 cluster of the Leibniz-Rechenzentrum(LRZ) in Munich. The workstation contains an AMD Ryzen™ 7 3700X processor[1] that features a base clock speed of 3.6 GHz with 8 cores and 16 threads. The cluster[2] uses processors based on Intel's Haswell architecture with a base clock speed of 2.6 GHz, 14 cores, and 28 threads. Most of the cluster data is included in the appendix.
In the following sections, these Lennard-Jones force functors are going the be compared against each other:

- **AoS:** This functor acts on Arrays-of-Structures. It takes two particles as parameters and is not vectorized via OpenMP or intrinsics.
- **AutoVec-CTC-M:** This basic functor uses Structure-of-Arrays and the auto-vectorization capabilities of OpenMP.
- **AVX-CTC-G/S:** This is the default AVX functor. It uses gather/scatter site masks and evaluates the cutoff on a Center-To-Center basis.
- **AVX-CTC-M:** This AVX functor utilizes regular site masks and evaluates the distance on a Center-To-Center basis.
- **AVX-CTS-G/S:** This AVX functor uses gather/scatter site masks and calculates the distance on a Center-To-Site basis.
- **AVX-CTS-M:** This AVX functor operates on regular site masks and determines the distance on a Center-To-Site basis.

[1] AMD Ryzen™ 7 3700X specification
[2] CoolMUC-2 cluster specification

4.1. Profile of the AVX functors

To better understand the following results, the durations of the different code segments were measured with C++ timers. Figure 4.1 shows the relative portion of the total time for all three functors, where the site masks use the CTC method to evaluate the cutoff and gather/scatter to handle the masking. The three segments are the setup phase, which creates auxiliary vectors for the exact site positions, the force phase, which includes the site mask build process and the actual force calculations and the reduction phase, which maps the site forces back to their respective particle.
First of all, the reduction phase occupies a nearly insignificant amount of time in all cases, and the other two segments dominate the run-time. For the Linked Cells functors, namely Single and Pair, the force calculation dominates the run-time in the uniform case and still requires the most amount of time in the Gaussian case. For the Verlet functor, however, setting up the auxiliary vectors requires nearly 80% of the total time for single sites. While this percentage decreases for more sites, it remains the major bottleneck for the AVX functor. Consequently, improving the computation of the exact site positions could significantly affect the performance of the Verlet functor.



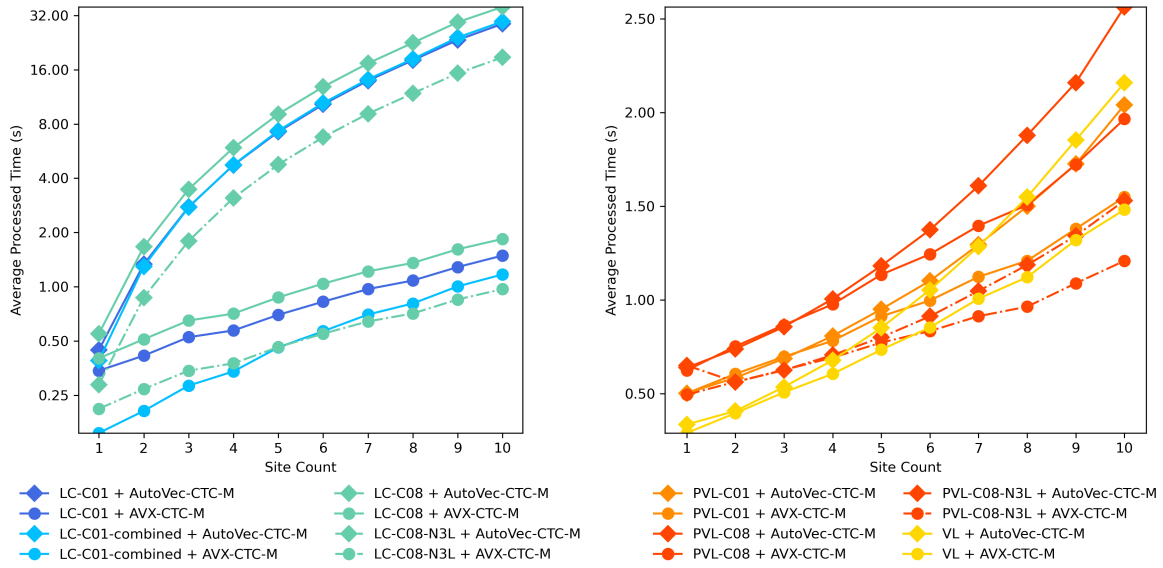
(a) Uniform distribution

(b) Gaussian distribution

Figure 4.1.: Percentages of code sections for the default AVX functor

4.2. Comparison of AVX and AutoVec

Figure 4.2 illustrates the measured run times for the auto-vectorized and the AVX functor. When Linked Cells are used, the AVX functor significantly outperforms the auto-vectorization, up to a factor of 30 for traversals without N3L. There are various possible explanations for this behavior. First, the manually vectorized implementation may contain segments the compiler failed to vectorize adequately for the auto-vectorized version and thus perform more optimally. Second, the AVX functor does not manage designated containers for mixing data like AutoVec, but instead loads the appropriate values directly (see section 3.4.3). Third, the AVX functor, in contrast to AutoVec, avoids building a molecular mask and creates the site mask directly, thus preventing these memory allocations.
In the case of Verlet Lists, the functors show considerably more similar run times. For the uniform distribution, the run times between the AVX and AutoVec functors are nearly identical for small site counts, and even for a high number of sites, the difference is relatively insignificant. The Gaussian distribution shows a similar behavior, although the differences between the two functors and the same traversal become larger for more interaction sites. However, the choice of traversal seems to have a more considerable impact here than choice of the functor.
Finally, it can be observed that the best-performing combination is different for the uniform and the Gaussian case: For the uniform distribution, the **LC-C08-N3L** combination performs best, while the standard **LC-C01** traversal shows the shortest processed time for Gaussian-distributed particles. This discrepancy demonstrates that the most optimal algorithmic choice depends on the simulation parameters and can justify the usage of auto-tuning libraries.



(a) Uniform distribution - Linked Cells

(b) Uniform distribution - Verlet Lists

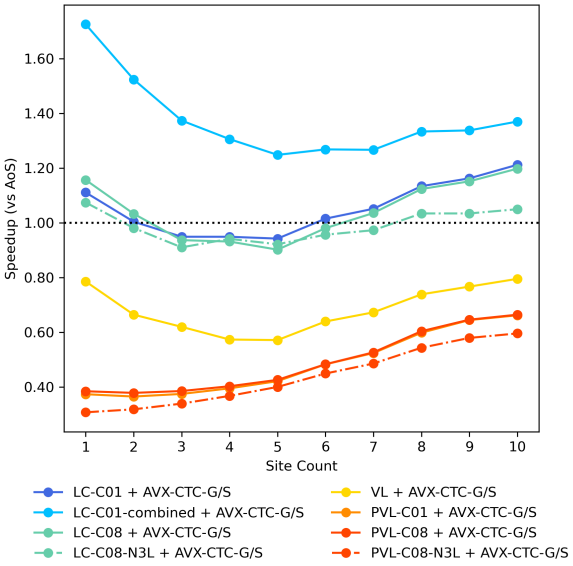
(c) Gaussian distribution - Linked Cells

(d) Gaussian distribution - Verlet Lists

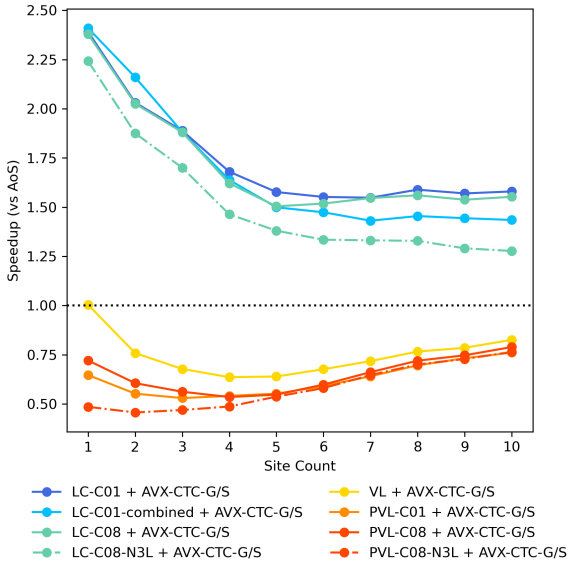
Figure 4.2.: Comparison of AVX-CTC-M and AutoVec-CTC-M on the workstation with 16 threads
For Linked Cells, the wall time differs significantly, which made a logarithmic scale necessary.
For Verlet Lists, the difference between AVX and AutoVec is considerably smaller, and a linear scale is sufficient.

4.3. Comparison of AVX and AoS

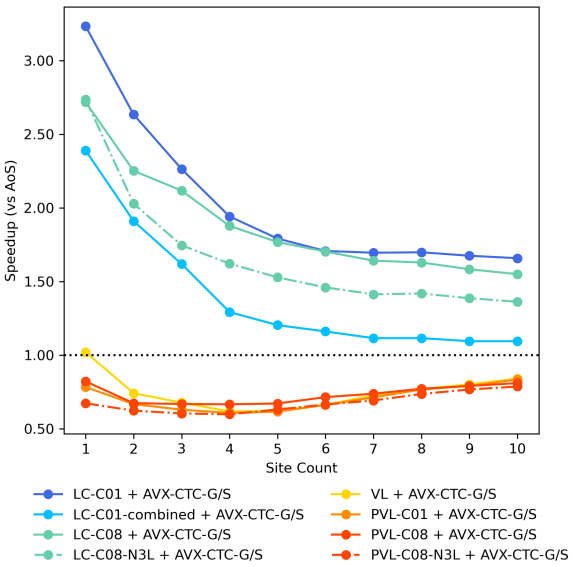
Figure 4.3 illustrates the standard AVX functor's relative speed-up compared to the scalar AoS functor for multiple different particle distributions and domain sizes. The graphs of the traversals generally follow a similar trend for both distributions but seem vertically shifted. In the uniform case with a $50 \times 50 \times 50$ domain, only the combined C01-traversal can consistently outperform the AoS functor. However, the respective graph compares the combined traversal of the AVX functor to the standard C01-traversal of the AoS functor due to the latter's inability to use this traversal. As a result, the speed-up for the combined traversal is partially caused by the comparison of different traversals. The remaining Linked Cell traversals perform similarly to the AoS version, while the traversals for Verlet Lists run faster with the AoS functor. This behavior could potentially be caused by the additional overhead of the SoA functors to create containers for the exact site positions, potentially outweighing the gains of vectorization (see section 4.1).
In the second case (Figure 4.3b), the simulation contains only half of the previous particles, but the domain is also halved in each dimension. This change effectively quadruples the particle density while keeping the same level of particle homogeneity. The increased density clearly favors the AVX functor for the Linked Cells containers: There is a considerable speed-up for small site counts, which diminishes when the number of interaction sites increases. Nonetheless, the relative speed-up remains nearly constant at about 1.5 if particles consist of at least 4 sites. Since each cell in this scenario contains 4 times as many particles as in the previous scenario, the number of force calculations per functor call increases significantly. These additional calculations likely outweigh the overhead of vectorization, and cause the increased relative speed-up of the AVX version. In contrast, the AVX Verlet List functor remains inferior to the AoS version in this scenario, likely for similar reasons as in the previous configuration.
The Gaussian scenarios share the same domain size and particle count but differ in the standard deviation of the particle distribution. This means that, due to the properties of the Gaussian distribution, about 95% of particles are located within 10 units of the mean in the first scenario. In contrast, this holds for only about 68% of particles in the second scenario. Therefore, the particles in the second scenario are more spread out and less condensed in the center. For Linked Cells, the graphs suggest that the more concentrated scenario favors the AVX functor, while the AoS functor is more suitable for the sparser simulation. This is likely due to similar reasons as in the uniform case, where the high concentration of particles in the center leads to many force calculations, outweighing the additional vectorization overhead. At the same time, sparsely populated cells are more receptive to the scalar functor since the absence of auxiliary containers seems to be more significant here than the gains of vectorization. Nevertheless, increasing the site count also positively affects the performance of the AVX functor in the second scenario because of the increased number of required force computations.
Lastly, while the vectorized Verlet functor remains inferior in both cases, it is more receptive to increased force calculations per cell. It could potentially match the AoS functor for sufficiently large site counts or densities.



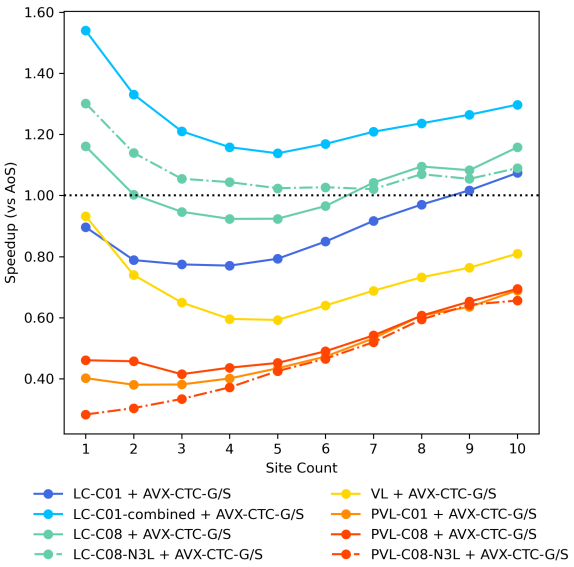
(a) Uniform - 100k particles - 50x50x50 domain



(b) Uniform - 50k particles - 25x25x25 domain



(c) Gaussian($\sigma = 5$) - 25k particles 50x50x50 domain

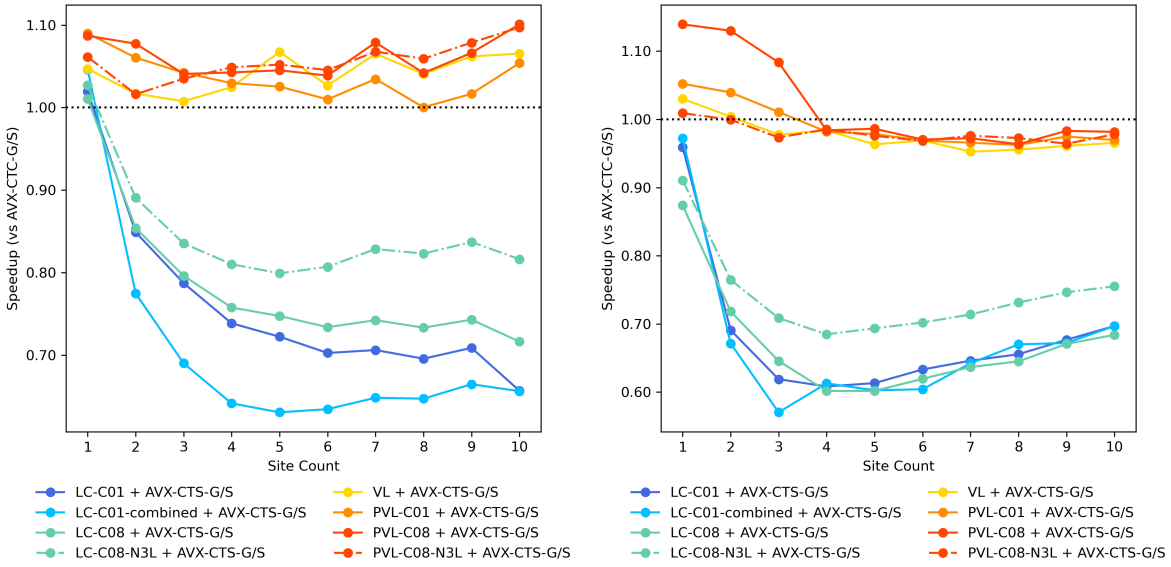


(d) Gaussian($\sigma = 10$) - 25k particles 50x50x50 domain

Figure 4.3.: Speed-up of AVX-CTC-G/S functor against the AoS functor for different scenarios

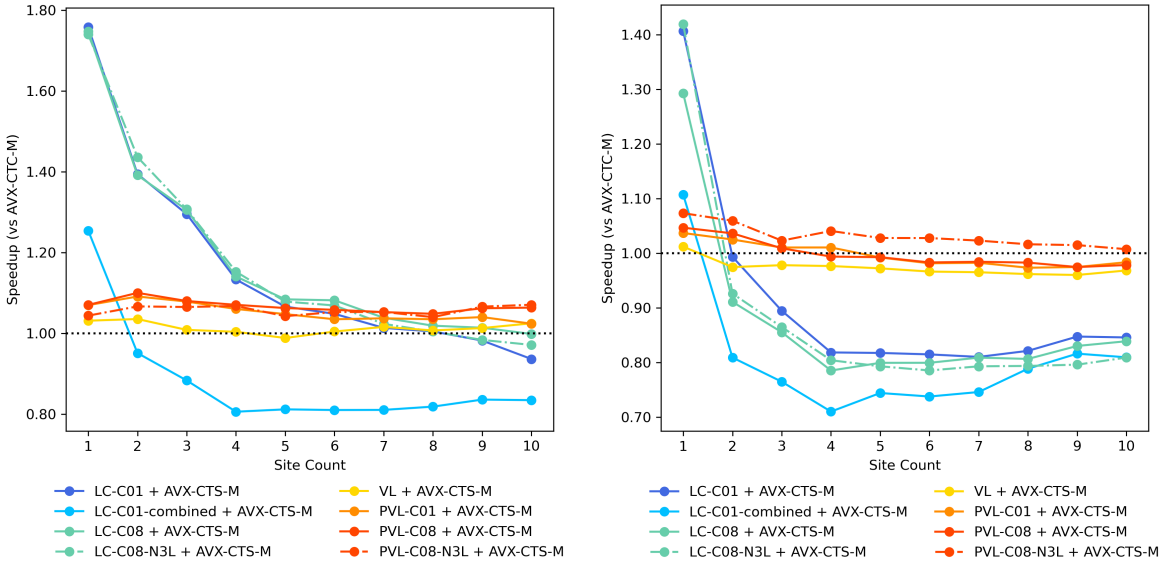
4.4. Comparison of Center-To-Center and Center-To-Site site masks

Figure 4.4 shows the speedups of the CTS mask, which is the average wall time of CTS divided by the average time of the CTC method. Furthermore, both site mask types were compared against their respective CTC counterpart.
For Linked Cells and gather/scatter masks, the CTS mask performs considerably worse than the CTC method for both distributions, down to a factor of 0.6 in the Gaussian scenario. While both methods demonstrate similar run-time for single sites, the relative speed-up decreases for more interaction sites per particle. This behavior is caused by the increased number of distance computations of CTS compared to CTC. However, the speed-up remains relatively constant when at least 4 sites are used and even increases for the Gaussian distribution, possibly due to the vectorization of the CTS variant compared to the scalar CTC method. In addition, the number of required force calculations scales quadratically with the number of sites compared to the linear scaling for building the site mask. As a result, the impact of the site mask build strategy likely diminishes for larger site counts, and the force calculations dominate the total run time. Finally, It is also worth noting that the speedup of CTS is higher for the N3L optimization since fewer site masks are being built, reducing the impact of the site mask build process.
On the other hand, CTS performs substantially faster for single sites if regular masks are employed, but this effect quickly diminishes for more interaction sites, similar to the gather/scatter masks. However, both distance strategies demonstrate near similar run-time in the uniform case, while the Gaussian scenario favors the CTC variant. Since regular masks usually involve more force calculations, the site mask build process is likely again overshadowed by the force kernel, and the difference between strategies remains near-constant.
If the Verlet List containers are used, both methods demonstrate similar performance in every scenario. This is again likely due to the small number of sites that each functor has to consider, which limits the impact of the site mask build process. In the uniform case, the CTS variant generally performs better, which is probably caused by the CTS vectorization. On the other hand, the CTC method is slightly better in the Gaussian case. It could be possible that the overhead of vectorization outweighs its gain due to the relatively low site count of Verlet Lists compared to Linked Cells.
Finally, it is essential to highlight that the exact positions of all sites are identical to their respective center of masses, and, as a result, both methods produce the same site mask. CTS may perform better in scenarios where these positions are not identical since it can drop sites outside of the cutoff while their center of mass is within the cutoff, thus decreasing the number of force calculations.



(a) Uniform distribution - Gather/Scatter masks

(b) Gaussian distribution - Gather/Scatter masks



(c) Uniform distribution - Regular masks

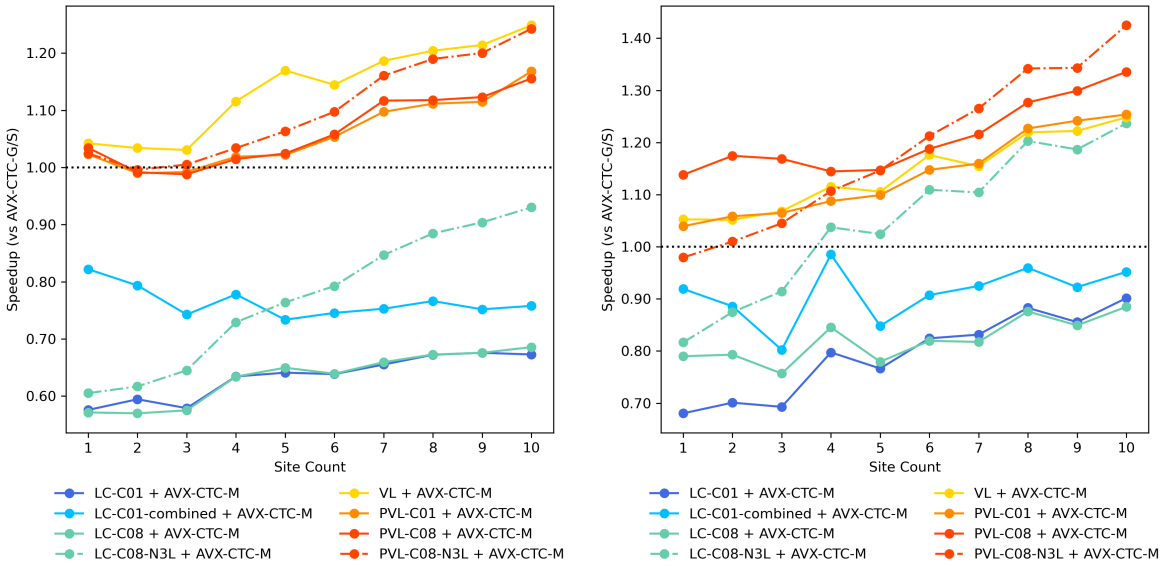
(d) Gaussian distribution - Regular masks

Figure 4.4.: Comparison of CTC- and CTS-based site masks

4.5. Comparison of regular and gather/scatter site masks

Figure 4.5 illustrates the speedup of the regular masking method compared to the gather/scatter method. The speedups were measured on a CTC basis for the uniform and Gaussian distribution and multiple combinations of particle containers and traversals.
If Linked Cells are used, there is a negative speedup, which means the regular masks perform worse than the gather/scatter approach. Since Linked Cells typically possess a low hit rate (see 2.6.3) and the gather/scatter method only calculates forces for particles within the cutoff, the latter can drop many unnecessary force computations and can thus outperform regular masks. If N3L is not utilized, the speedup stays relatively constant. To understand this behavior, it is helpful to consider the number of force calculations for each method. Since both strategies differ in the number of particles but not in the site count per particle, the ratio of force calculations for the regular and the gather/scatter method is independent of the site count, and, as a result, the total speedup is nearly constant.
In contrast, the N3L traversal shows a different behavior, where the speedup successively increases and even surpasses the gather/scatter method for Gaussian-distributed particles on the workstation. This can be explained by the scatter-instructions which are necessary to perform the N3L optimization. As previously mentioned (section 2.8.3), the scatter-intrinsics are unavailable on both the cluster and the workstation and were replaced by scalar instructions. In contrast, the regular mask uses efficient vectorized store operations. Since the number of these operations depends on the site count, the speed-up of the regular mask increases. Also, it is interesting to see that this effect cannot be not observed on the CoolMUC-2 cluster. This can be explained by the older Haswell-based hardware of the cluster, which does not implement gather-intrinsics as efficiently as successive architectures [18]. This discrepancy suggests that the optimal method may also depend on the hardware. In addition, it demonstrates that the absence of efficient scattering significantly impacts the performance of the gather/scatter site mask.
On the contrary, Verlet Lists demonstrate a similar performance for low site counts and show an increased speed-up for more sites. Since Verlet Lists typically have a higher hit rate than Linked Cells, the index site mask cannot filter out as many particles. Both masking methods must therefore consider a similar number of force calculations, which reduces the performance gain of the gather/scatter method. However, the gather/scatter method requires random memory access, whereas regular masks contiguously read and write to memory. Since standard memory operations do not carry as much overhead as their gather/scatter counterparts for managing the index vector, they typically show faster memory access and can thus perform better than the gather/scatter approach. The graphs suggest that the usage of contiguous memory operations can outweigh the cost of potentially more force calculations, which is demonstrated by a speed-up of up to 40% in the Gaussian distribution on the workstation.
Finally, it can also be observed how specific site counts affect the speed-up of regular masks. For example, while the speed-up of the C01-combined traversal stays relatively constant for the uniform distribution, there are minor peaks at 4 and 8 sites. Similarly, while the graph for the Gaussian distribution fluctuates more, it also demonstrates peaks at 4 sites and 8 sites. This suggests that there is a positive effect on the regular masks when the site count is a multiple

of the vector length, which is likely due to an optimization of regular masks, where a force calculation is omitted if the currently loaded mask contains only zeros. If all particles share the same number of interaction sites, and this number is a multiple of the vector length, then each mask in the force kernel will contain either only zeros or ones. Therefore the regular mask can skip a lot of unnecessary calculations and thus performs similarly to the gather/scatter method. However, this effect only occurs under these particular circumstances and depends on the simulation configuration. Finally, this effect suggests that the optimal site mask type also depends on the given scenario since the regular site masks could perform similarly to the gather/scatter masks at specific site counts (which nearly occurs for the Gaussian distribution on the workstation).



(a) Uniform distribution - Workstation

(b) Gaussian distribution - Workstation

(c) Uniform distribution - Cluster

(d) Gaussian distribution - Cluster

Figure 4.5.: Comparison of gather/scatter and regular site masks

5. Future Work

5.1. AVX512

While the current implementation can achieve a considerable speed-up compared to auto-vectorization, this speed-up is limited by the capabilities of AVX2. Adding support for AVX512 can further increase the intrinsic code's performance gain. First, the program can utilize the doubled vector register size to halve the number of required instructions potentially. Furthermore, AVX512 added support for native scatter instructions, which could significantly improve the performance of traversals that utilize Newton's third law.

5.2. Exact site position calculation

The profiling of the AVX functor (4.1) demonstrated that there is a considerable amount of time spent calculating the exact site positions, especially for Verlet functors. Therefore, an improved mechanism to retrieve the exact site positions could significantly improve the run-time of the functors. One way to achieve this is by creating a global container for the exact site positions in the functor class, which is updated once per time step. Then the simulation does not need to build a new container in each functor call but only once per iteration, thus decreasing the number of exact site position calculations. However, this approach can be challenging to implement for multiple threads. If only one thread builds the exact site vector, the remaining threads are effectively idling, thus potentially reducing the gain of this method. Another idea is to generate the exact site positions as part of the particle data in the SoA.

5.3. Dynamic site mask strategy

The results chapter demonstrated that the optimal vectorization technique depends on the given scenario. For example, the regular masks performed better for Verlet Lists, while the indexed masks were a better fit for Linked Cells (Figure 4.5). Currently, the combination of site mask strategies is determined at compile-time and is shared by every functor. However, this selection could also occur during run-time, and each functor could also use a different strategy. Therefore, implementing a mechanism that dynamically determines and selects the most appropriate site mask strategies for each functor could be advantageous.

6. Conclusion

This thesis has explored the vectorization of the Lennard-Jones potential with AVX intrinsics. The primary objective of this research was to investigate the benefits and limitations of utilizing these intrinsics to speed up the calculation of the Lennard-Jones potential for multicentered molecules.

The background chapter briefly introduced the foundations of molecular dynamics, the Lennard-Jones potential, and multicentered particles. Afterward, the chapter discussed the algorithmic foundation, including particle container, traversals, and vectorization, and thus laid the groundwork for subsequent chapters.

The subsequent chapter provided the implementation of the Lennard-Jones force calculation via AVX intrinsics. Various different strategies for the site mask construction were presented, namely center-to-center and center-to-site masks, as well as regular and indexed masks. The section continued with the force calculation kernel, which calculates the actual force using a sequence of arithmetic and memory AVX intrinsics. Furthermore, the kernel also described the torque calculation, the optimization with Newton's third law, and the computation of additional physical properties.

This new implementation was first compared against an auto-vectorized and a scalar version in the comparison chapter. The implementation for Linked Cells was able to significantly outperform the auto-vectorization, up to a factor of 30 in the uniform case. Furthermore, it was demonstrated that the AVX version could beat the scalar version for large enough densities, while simulations with lower densities or more particle spread generally favored the AoS functor. However, the Verlet List implementation could not deliver a similar speed-up and was consistently outperformed by the scalar functor. A separate section that profiled the several stages of the implementation aimed to explain this behavior and highlighted the need for further optimization for currently non-vectorized portions of the functor.

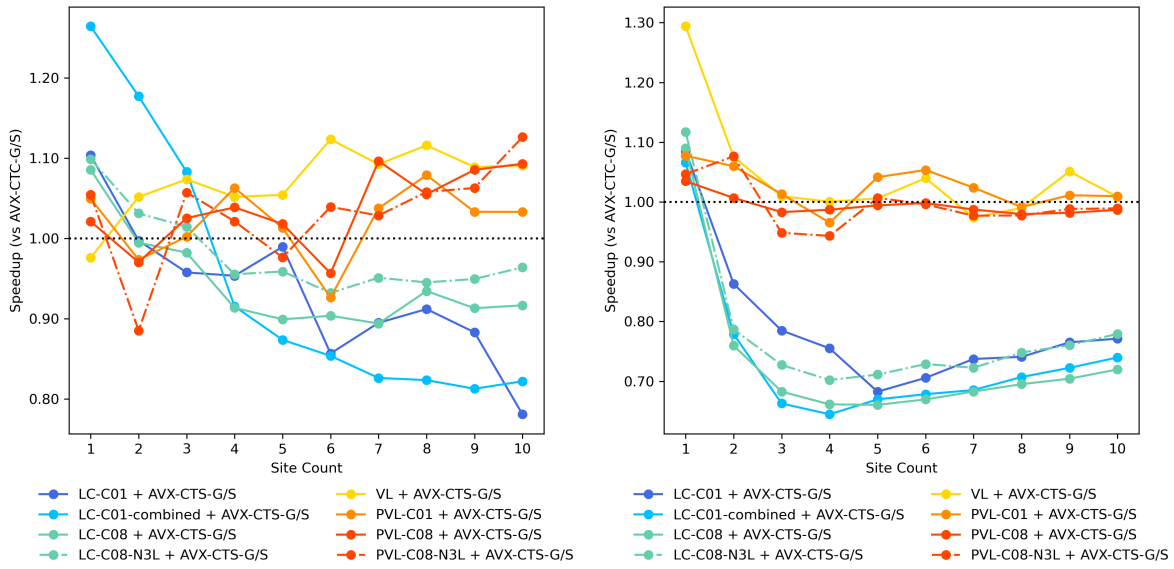
Furthermore, this thesis has provided a detailed comparison of different site mask strategies. In the case of distance evaluation, the Center-To-Center method was a better fit for Linked Cells due to their comparatively low number of distance calculations, while the Verlet functor slightly preferred the Center-To-Site approach. A similar behavior could be observed for regular and indexed masks: While the first strategy performed better for Verlet Lists, the functor with indexed masks ran considerably faster for Linked Cells. This section has also highlighted that the optimal vectorization strategy depends on the particle container and traversal, the specific hardware, and the simulation parameters.

In conclusion, this thesis has demonstrated the potential improvements of vectorization in molecular dynamics. It has shown that manual vectorization can potentially open avenues for faster and more efficient simulations, ultimately enabling researchers to explore larger systems and longer timescales in their investigations.

A. Appendix

This chapter contains additional results of the SuperMUC2-cluster.

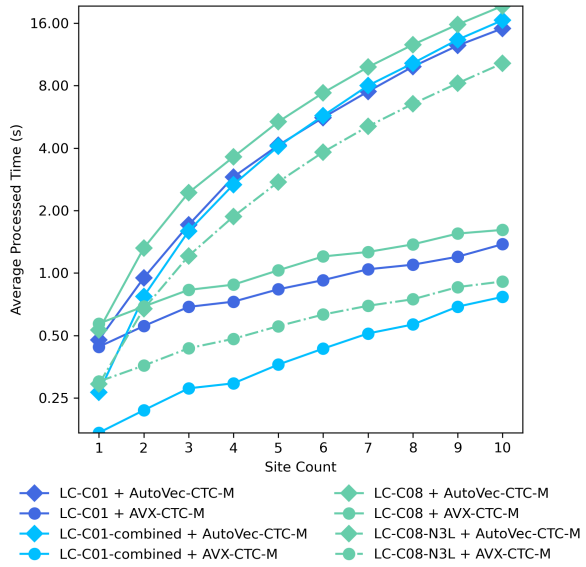
A.1. Linux Cluster results



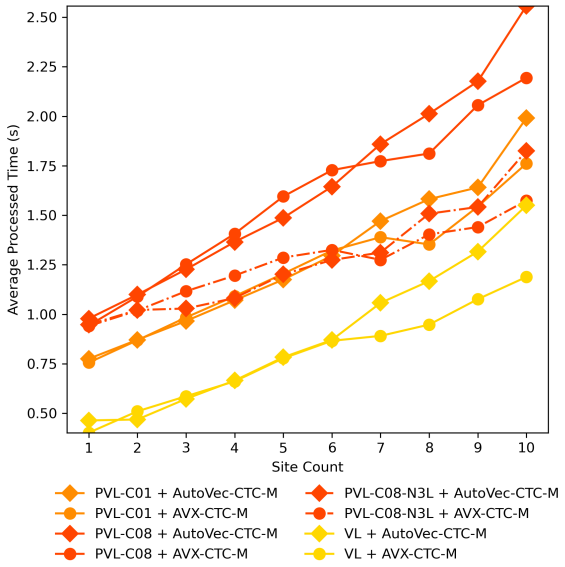
(a) Uniform distribution

(b) Gaussian distribution

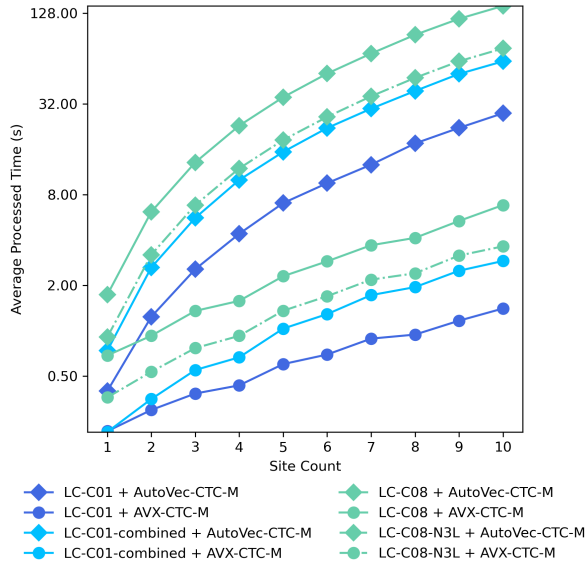
Figure A.1.: Comparison of CTC- and CTS-based site masks on the cluster with 32 threads



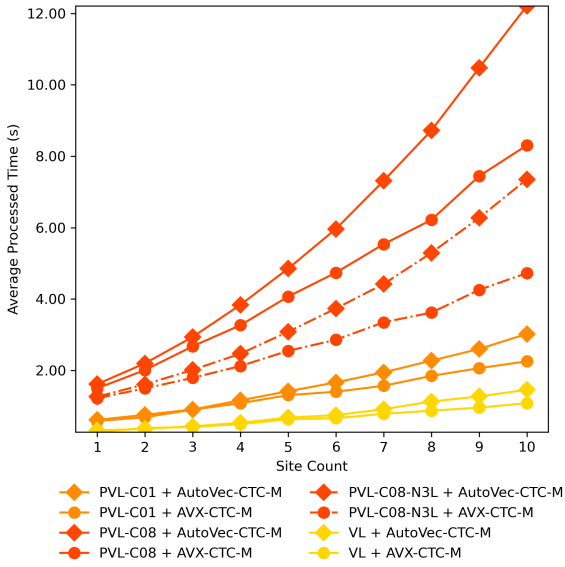
(a) Uniform distribution - Linked Cells



(b) Uniform distribution - Verlet Lists

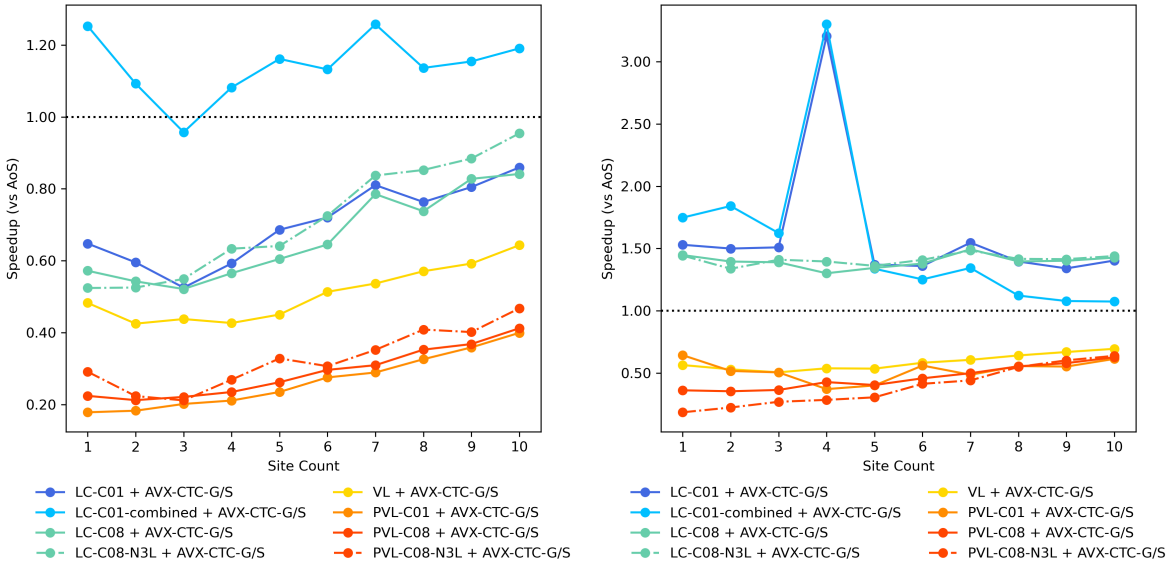


(c) Gaussian distribution - Linked Cells



(d) Gaussian distribution - Verlet Lists

Figure A.2.: Average processed times of AVX-CTC-M and AutoVec-CTC-M on the cluster with 32 threads



(a) Uniform - 100k particles - 50x50x50 domain

(b) Uniform - 50k particles - 25x25x25 domain

(c) Gaussian($\sigma = 5$) - 25k particles 50x50x50 domain

(d) Gaussian($\sigma = 10$) - 25k particles 50x50x50 domain

Figure A.3.: Speed-up of AVX-CTC-G/S compared to AoS on the Linux Cluster with 32 threads

List of Figures

Bibliography

[1] M. Karplus and J. A. McCammon. “Molecular dynamics simulations of biomolecules”. In: *Nature Structural Biology* 9 (9 2002), pp. 646–652.

[2] G. Nagayama and P. Cheng. “Effects of interface wettability on microscale flow by molecular dynamics simulation”. In: *International Journal of Heat and Mass Transfer* 47.3 (2004), pp. 501–513. ISSN: 0017-9310.

[3] P. Neumann, N. Tchipev, S. Seckler, M. Heinen, J. Vrabec, and H.-J. Bungartz. “PetaFLOP Molecular Dynamics for Engineering Applications”. In: *High Performance Computing in Science and Engineering ’ 18*. Ed. by W. E. Nagel, D. H. Kröner, and M. M. Resch. Cham: Springer International Publishing, 2019, pp. 397–407.

[4] N. Tchipev, S. Seckler, M. Heinen, J. Vrabec, F. Gratl, M. Horsch, M. Bernreuther, C. W. Glass, C. Niethammer, N. Hammer, B. Krischok, M. Resch, D. Kranzlmüller, H. Hasse, H.-J. Bungartz, and P. Neumann. “TweTriS: Twenty trillion-atom simulation”. In: *The International Journal of High Performance Computing Applications* 33.5 (2019), pp. 838–854.

[5] J. Pennycook, C. Hughes, M. Smelyanskiy, and S. Jarvis. “Exploring SIMD for Molecular Dynamics, Using Intel® Xeon® Processors and Intel® Xeon Phi Coprocessors”. In: May 2013, pp. 1085–1097. ISBN: 978-1-4673-6066-1.

[6] H. Watanabe and K. M. Nakagawa. “SIMD Vectorization for the Lennard-Jones Potential with AVX2 and AVX-512 instructions”. In: *CoRR* abs/1806.05713 (2018). arXiv: 1806.05713. URL: http://arxiv.org/abs/1806.05713.

[7] G. Z. Michael Griebel Stephan Knapek. *Numerical simulation in molecular dynamics*. 2007.

[8] L. Verlet. “Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules”. In: *Phys. Rev.* 159 (1 July 1967), pp. 98–103.

[9] J. E. Lennard-Jones. “On the determination of molecular fields. ii. from the equation of state of a gas”. In: *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 106 (1924), pp. 463–477.

[10] W. Eckhardt and A. Heinecke. “An Efficient Vectorization of Linked-Cell Particle Simulations”. In: *Proceedings of the 9th Conference on Computing Frontiers*. CF ’12. Cagliari, Italy: Association for Computing Machinery, 2012, pp. 241–244. ISBN: 9781450312158.

[11] F. A. Gratl, S. Seckler, H.-J. Bungartz, and P. Neumann. “N ways to simulate short-range particle systems: Automated algorithm selection with the node-level library AutoPas”. In: *Computer Physics Communications* 273 (2022), p. 108262. ISSN: 0010-4655.

[12] S. Seckler, F. Gratl, M. Heinen, J. Vrabec, H.-J. Bungartz, and P. Neumann. "AutoPas in ls1 mardyn: Massively parallel particle simulations with node-level auto-tuning". In: *Journal of Computational Science* 50 (2021), p. 101296. ISSN: 1877-7503.

[13] S. J. Newcome, F. A. Gratl, P. Neumann, and H.-J. Bungartz. "Towards auto-tuning Multi-Site Molecular Dynamics simulations with AutoPas". In: *Journal of Computational and Applied Mathematics* 433 (2023), p. 115278. ISSN: 0377-0427.

[14] W. Mattson and B. M. Rice. "Near-neighbor calculations using a modified cell-linked list method". In: *Computer Physics Communications* 119.2 (1999), pp. 135–148. ISSN: 0010-4655.

[15] P. Gonnet. "Pairwise verlet lists: Combining cell lists and verlet lists to improve memory locality and parallelism". In: *Journal of Computational Chemistry* 33.1 (2012), pp. 76–81.

[16] N. P. Tchipev. "Algorithmic and Implementational Optimizations of Molecular Dynamics Simulations for Process Engineering". Ph.D. dissertation. Technische Universität München, 2020.

[17] *Intel Intrinsics Guide*. `https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#techs=AVX_ALL`. Accessed: July 7, 2023.

[18] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Figure 15-4: Throughput Comparison of Gather Instructions. Intel Corporation. 2023.