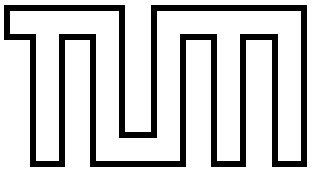


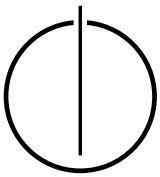
SCHOOL OF COMPUTATION, INFORMATION
AND TECHNOLOGY - INFORMATICS

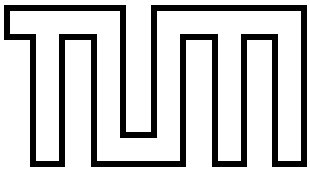
TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Vectorization of Three-Body Potentials in AutoPas

Jakob Andreas Englhauser





SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

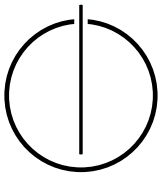
TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Vectorization of Three-Body Potentials in AutoPas

Vektorisierung von Dreikörperpotentialen in AutoPas

Author:	Jakob Andreas Englhauser
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Samuel James Newcome, M.Sc.
Date:	May 15, 2024



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

Munich, May 15, 2024 Jakob Andreas Englhauser

Abstract

Molecular dynamics simulations rely on potential functions to model intermolecular interactions. While it is often enough to only consider pair-wise potentials, such as the Lennard-Jones potential, in some situations, it may be necessary to additionally use three-body potentials to produce accurate results. One example of a three-body potential is the Axilrod-Teller potential, which should be implemented in the molecular dynamics library AutoPas. Since the calculation of the potentials is a major part of the runtime of molecular dynamics simulation, an efficient implementation is vital for the performance of the simulation. For this, SIMD instructions also play an important role. Therefore, ways of manually vectorizing the Axilrod-Teller potential with SIMD intrinsics were explored in this work.

Zusammenfassung

Molekulardynamiksimulationen basieren auf der Modellierung intermolekularer Wechselwirkungen durch Potentiale. Oft genügt es zwar nur paarweise Potentiale, wie das Lennard-Jones Potential, in Betracht zu ziehen, in manchen Fällen benötigt man jedoch zusätzlich Dreikörperpotentiale, um akkurate Ergebnisse zu erzielen. Ein Beispiel für ein Dreikörperpotential ist das Axilrod-Teller Potential, welches in der Molekulardynamikbibliothek AutoPas implementiert werden soll. Da die Berechnung der Potentiale einen großen Teil der Laufzeit von Molekulardynamiksimulationen ausmacht, ist eine effiziente Implementierung essenziell für die Performanz der Simulation. Dafür spielen auch SIMD Instruktionen eine bedeutende Rolle. Deshalb wurden im Rahmen dieser Arbeit Möglichkeiten zur manuellen Vektorisierung des Axilrod-Teller Potentials mit SIMD intrinsics erkundet.

Contents

Part I.

Introduction and Background

1. Introduction

Molecular dynamics simulations allow studying intermolecular interactions where experimental analysis is not feasible due to, for example, cost or small timescales. The knowledge gained from these simulations can prove valuable in various fields of science such as medicine, where they can aid in the discovery of new drugs [DM11]. Many molecular dynamics simulations rely primarily on modeling of pairwise interactions based on the Lennard-Jones potential [LJ24]. However, in some situations, such as the simulation of noble gases, the Lennard-Jones potential alone is not sufficient to provide fully accurate results and additional three-body potentials like the Axilrod-Teller potential have to be used [Mar01].

Molecular dynamics simulations may face challenges where fine resolution is required since the high number of particles can increase the needed computation time to an unacceptable amount. Therefore, high levels of optimizations are desired for simulation programs to achieve ideal performance. One major aspect of this is exploiting as much parallelism offered by modern computers as possible. This includes clusters where the workload is spread across multiple compute nodes, multi-core processors, and also utilization of so-called single instruction, multiple data (SIMD) instructions [Fly11]. These instructions are often used automatically by the compiler without the programmer's explicit intent. However, if the code is more complex, manual vectorization may be required to achieve optimal performance.

Another aspect of optimizations is choosing the right set of algorithms and data structures for the given task, which can prove difficult even for experienced programmers. AutoPas is an auto-tuning library for simulation programs created to alleviate this problem by automatically choosing optimal algorithms and data structures based on empirical runtime measurements. [GSBN21]

Support for the aforementioned three-body potentials in AutoPas is still in development. Therefore, the goal of this work is to aid in this task by implementing a vectorized version of the Axilrod-Teller potential.

2. Theoretical Background

2.1. Molecular Dynamics

2.1.1. Potentials

Molecular dynamics simulations mainly deal with the computation of intermolecular interactions. Due to the complex quantum mechanical nature of these interactions, they are modeled by potential functions. Since these potentials are by necessity approximations, there exist several different potentials which might excel for different use cases [Bre00].

A simple and commonly used potential is the pairwise Lennard-Jones 12-6 potential:

$$U_{LJ} = \underbrace{4\epsilon\left(\frac{\sigma}{r_{ij}}\right)^{12}}_{\text{Van-der-Walls forces}} - \underbrace{4\epsilon\left(\frac{\sigma}{r_{ij}}\right)^{6}}_{\text{Pauli repulsion}} \tag{2.1}$$

$$= 4\epsilon\left(\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right) \tag{2.2}$$

where r_{ij} is the distance between the two molecules and ϵ and σ are constants depending on the molecular model to be simulated. [LJ24]

The Lennard-Jones potential can be split into a negative, attractive and a positive, repulsive part.

The attractive part is explained by Van-der-Waals forces. Atoms consist of a positively charged nucleus and a negatively charged electron shell. The electrons are constantly moving within the shell with their charge normally uniformly distributed over the atom. However, since particles with an electric charge of the same sign repel each other, once two atoms get closer to each other, the mutual repulsion of the electrons will push them towards opposing sides of one nucleus, as shown in Figure 2.1. As a result, two temporary dipoles form, creating attractive forces between the two atoms, which are modeled by the negative part of the Lennart-Jones potential.

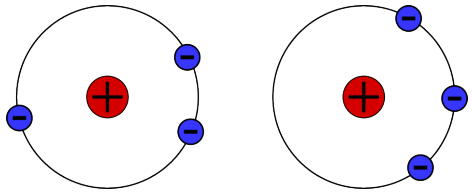


Figure 2.1.: Illustration of Van-der-Waals forces. The equal charge of the electrons pushes them away from each other, thus creating two temporary dipoles.

However, the repulsion between electrons also results in an overall repelling force between

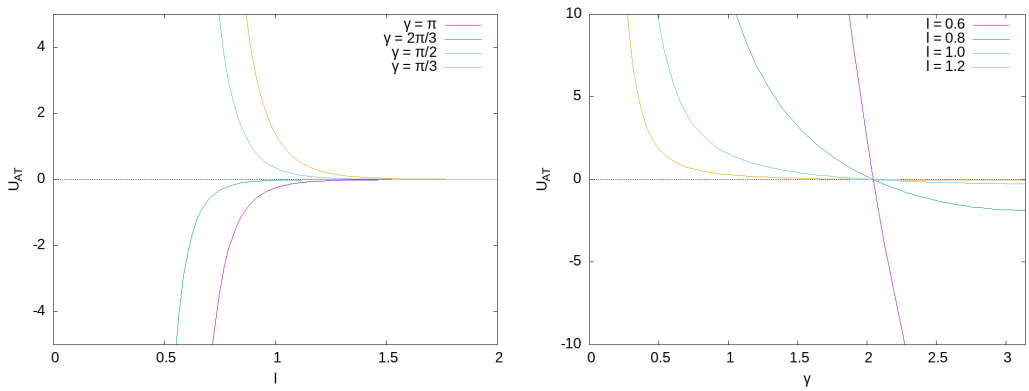
the two atoms if they get too close. This so-called Pauli repulsion is modeled by the positive part of the Lennard-Jones potential [BZBP14].

Pairwise interactions cannot fully explain all forces acting between multiple particles since the presence of more than two particles may induce additional interactions. Therefore, to create accurate simulations, it might be necessary to include three-body-potentials to model these forces [Mar01]. One such potential is the Axilrod-Teller potential defined by the formula:

$$U_{AT} = \nu \frac{1 + 3(\cos\gamma_i \cos\gamma_j \cos\gamma_k)}{(r_{ij} r_{jk} r_{ki})^3} \tag{2.3}$$

where ν is a constant depending on the ionization energy and polarizability of the particles [AT43]. Applying the cosine rule $\cos\gamma_i = \frac{r_{ij}^2 - r_{jk}^2 + r_{ki}^2}{2 r_{ij} r_{jk}}$ eliminates the dependence on the angles between the particles:

$$U_{AT} = \nu \left(\frac{1}{(r_{ij} r_{jk} r_{ki})^3} + \frac{3(r_{ij}^2 - r_{jk}^2 + r_{ki}^2)(r_{ij}^2 + r_{jk}^2 - r_{ki}^2)(-r_{ij}^2 + r_{jk}^2 + r_{ki}^2)}{8(r_{ij} r_{jk} r_{ki})^5} \right) \tag{2.4}$$



(a) With increasing distance between particles the Axilrod-Teller potential approaches zero.

(b) The sign of the Axilrod-Teller potential depends on the form of the triangle formed between the particles. For an acute triangle the potential is positive. If the particles are arranged in a line it is negative.

Figure 2.2.: Axilrod-Teller potential for particles placed at the corners of an isosceles triangle with angle between legs γ and length of legs l.

2.1.2. Forces and Movements

A particle's movement depends on its acceleration, which in turn depends on forces acting on the particle based on Newton's third law $F = m * a$ [New87]. One can obtain these forces by taking the negative gradient of the potential:

$$\mathbf{F}_{ij}(\mathbf{r}_{ij}) = -\nabla U(r_{ij}) \tag{2.5}$$

The simulation then steps through time in discrete intervals, applying Verlet integration [Ver67] to calculate velocities and positions.

2.1.3. Mixing

The parameters σ and ϵ for the Lennard-Jones potential, and the parameter ν for the Axilrod-Teller potential are constants depending on which type of molecule is simulated. When considering different types of molecules with different σ, ϵ, or ν one has to reach a compromise between the values by applying certain mixing rules [BZBP14].

2.1.4. Cutoff

Both the Lennard-Jones and the Axilrod-Teller potentials are short-range potentials [Mic79], meaning they quickly approach zero with increasing distance between particles as shown in Figure 2.2a. As a consequence, it is possible to disregard forces between particles further away than a set cutoff radius r_c entirely without qualitatively changing the result but reducing the computational intensity of the force calculations. For two particles, the interpretation of the cutoff radius is obvious. For three particles, the cutoff condition can be applied in multiple ways, such as requiring all three particles to be pairwise in cutoff or requiring only one pair to be within cutoff [Mar22]. In this work, we will only be using the former definition.

2.1.5. Newton's Third Law

Newton's third law states that for every action, there is a reaction of equal magnitude and opposing direction [New87]. In the context of three-body forces, this means that the force acting on one particle can be computed from the sum of forces acting on the other two particles [Mar01]:

$$\mathbf{F}_k = -(\mathbf{F}_i + \mathbf{F}_j)$$

Applying this principle reduces the amount of computationally intensive force calculations by one third. We will refer to this optimization as newton3 in the following.

3. Technical Background

3.1. AutoPas

Among the multitude of algorithms and data structures for molecular dynamics simulations there exists no singular optimal choice for all simulation scenarios since their performance heavily depends on the molecule layout. The complexity of large-scale particle simulations also makes it difficult to choose the best configuration of algorithms and data structures, even for a single scenario. Furthermore, as particles move around during the simulation, the optimal configuration may change at any time. A way to deal with these issues is to automatically adjust simulation parameters during the program execution through an auto-tuning library like AutoPas[1] [GSBN21].

AutoPas is a library intended to act as a black box for delivering an optimal configuration for molecular dynamics programs. It defines an extensible selection of algorithms and data structures that are chosen during runtime by empirically measuring their single-step runtime at periodic time intervals during program execution. Additionally, AutoPas offers a full molecular dynamics program called `md-flexible` to showcase AutoPas' capabilities [Fot19].

Algorithms and data structures relevant to this work will be explained here.

3.1.1. Particle Container

Recall from Subsection 2.1.4 the introduction of a cutoff radius to reduce force computations. With just the cutoff alone, every particle pair or triplet would still require calculating the distances between them, leading to a runtime complexity of $\mathcal{O}(n^2)$ for two-body functors, or $\mathcal{O}(n^3)$ for three-body functors. Therefore, it is additionally necessary to use data structures that make efficient use of the cutoff condition. A simple way to implement such a data structure is to subdivide the simulation domain into cubic cells with side-length of at least r_c and only consider interactions between particles in neighboring cells as illustrated in Figure 3.1. This approach is called the **Linked Cells** algorithm [GSBN21].

[1] `https://github.com/AutoPas/AutoPas`

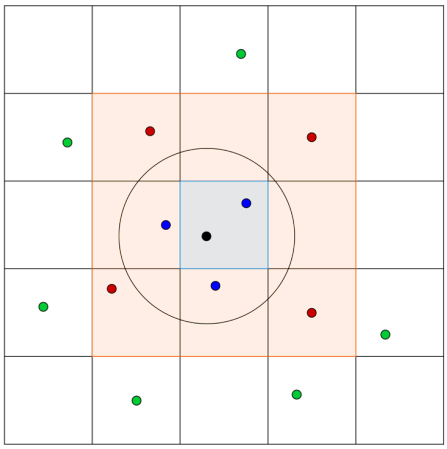


Figure 3.1.: Pairwise interactions between particles in the Linked Cells algorithm. Blue points are within the cutoff radius of the black point (full force calculation required), red points are in neighboring cells but outside the cutoff radius (only distance calculation required), green points are completely outside interaction range (no calculations required).

A drawback of this approach is the additional overhead for maintaining the cells, since particles may change cells as they move.

As of writing this thesis, the Linked Cells algorithm is the only particle container in AutoPas supporting three-body potentials.

3.1.2. Traversal

To be able to make use of the newton3 optimization explained in Subsection 2.1.5, the functor has to write forces to particles shared with other threads during parallel execution, meaning the traversal needs to employ strategies such as domain coloring to prevent race conditions [GSBN21]. Alternatively, one can opt to disregard the newton3 optimization entirely since then every thread needs to write only to particle forces in cells assigned to itself and only reads particle positions from other threads. Every cell is assigned to a thread that updates forces in that cell only, choosing any neighboring cell for the pair functor. For two-body potentials, this approach is called **lc_c01** [GSBN21]. The three-body version **lc_c01_3b** functions the same while additionally considering neighboring cell triplets.

Since the **lc_c01_3b** traversal is the only three-body traversal in AutoPas at the time of writing this work, no other traversals will be considered here.

3.1.3. Data Layout

Every particle carries information like position, velocity, and acting forces. This data can be stored in memory in different ways.

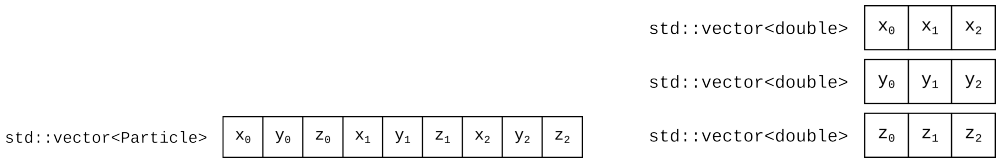
Array of Structures (AoS) Here a single particle's data is stored in an object. Multiple particle objects are stored in a `std::vector`.

Structure of Arrays (SoA) Here for every attribute of the particle a `std::vector` is created which holds the value for all particles.

Figure 3.2 illustrates both data layout versions using just spatial coordinates for three particles.

The AoS layout makes it easy to handle individual particles, for example, to move them between MPI processes or data structures. However, the SoA layout has the major advantage that data elements lie consecutively in memory, allowing for easy loading of data into vector registers, and thus more efficient vectorization of the code [GSBN21]. Therefore, this work only deals with SoA implementations.

AutoPas stores particles long-term only in AoS format. Whenever a procedure requires particles in SoA format, they have to be copied into SoA buffers and back to the AoS structure after the procedure has been completed.



(a) Array of Structures (b) Structure of Arrays

Figure 3.2.: Comparison of data layouts.

3.1.4. Functors

Functors are responsible for computing the interactions between particles, such as described in Section 2.1. Functors are implemented for AoS and SoA data layouts and can support either newton3 disabled or newton3 enabled, or both.

The triwise functor provides the following functions:

AoSFunctor The AoS functor takes a single particle triplet as an argument and calculates forces between them if they are within cutoff. The selection of triplets is handled by the traversal.

SoAFunctorSingle SoA implementation of the functor taking a single SoA container as an argument. The SoA container contains multiple particles, such as all particles in one cell, therefore the functor has to iterate over triplets itself. Since forces are written to particles in the SoA, the traversal has to ensure that no other threads are concurrently writing to the same SoA and as a consequence, it is always possible to apply the newton3 optimization.

SoAFunctorPair SoA functor taking two SoA containers as an argument. Particle triplets can be chosen by either picking two particles from the first SoA and one from the second, or one particle from the first and two from the second. Just as in the `SoAFunctorSingle` write accesses to the first SoA are always safe. However, for three-body potentials, the newton3 optimization only comes into full effect for the third particle.

SoAFunctorTriple SoA functor taking three SoA containers as an argument, taking one particle from each SoA.

InitTraversal, EndTraversal These functions are called at the start and beginning of every traversal, allowing the functor to reset and finalize buffers for global values.

SoALoader, SoAExtractor These functions allow copying data from AoS to SoA and vice versa, required for the use of the SoAFunctor, as explained in Subsection 3.1.3.

3.2. SIMD

Most modern CPU architectures provide functionality to apply the same instruction on different data simultaneously. This concept is known as single instruction, multiple data (SIMD) [Fly11].

A good compiler can analyze the code and identify repeated operations that can be vectorized on its own. However, uncertain data dependencies and complex control flow might make it impossible to guarantee that vectorization will not affect the correctness of the code, which will prevent auto-vectorization.

Instead of relying on the compiler, it is also possible to vectorize the code manually by using intrinsics functions [int], although this means that the code might no longer be portable across different CPU architectures.

Some intrinsics concepts important for this work will be explained in the following.

Aligned/unaligned memory accesses Regular load and store instructions like `_mm512_load_pd` require the memory to be aligned on a 64-byte boundary, meaning the memory address must be a multiple of 64. This can be ensured by using `aligned_alloc`, `memalign`, or similar functions. However, it might not always be possible to guarantee specific memory alignments. In that case, unaligned loads and stores such as `_mm512_loadu_pd` have to be used, which might be less efficient.

Masked instructions Many intrinsics functions have a masked alternative. These instructions take a bitmask as an extra argument and only apply the operation, or apply an alternative operation instead, if the corresponding bit is set to 1, as illustrated in Figure 3.3a.

Fused multiply-add Combinations of multiplication and addition are common especially when dealing with matrices or vectors. Therefore, AVX offers so-called fused multiply-add operations like `_mm512_fmadd_pd` which combine the two operations into one, as shown in Figure 3.3b.

Gather/Scatter Common loads and stores require data to be consecutively in memory. For situations where this is not the case, AVX512 offers gather and scatter instructions instead, which take an additional argument `vindex` specifying offsets relative to a base address for data to be loaded. Figure 3.3c visualizes this process.

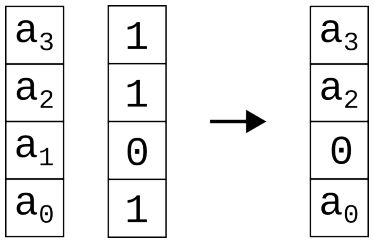
Compress Compress functions such as `_mm256_maskz_compress_pd` choose elements based on a bitmask and compress them towards the least significant bit of the register as shown in Figure 3.3d.

Alignr Figure 3.3e shows how `_mm512_alignr_epi64` and similar functions concatenate two registers and shift the result to the right (least significant bit).

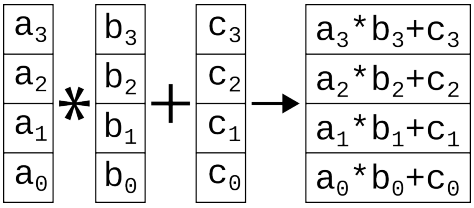
3.2.1. SIMD-everywhere

Since the availability of SIMD intrinsics depends on the target hardware, one would usually have to explicitly write multiple versions of the code to support all architectures. SIMD-everywhere[2] (SIMDe) aims to avoid this issue by providing implementations of SIMD intrinsics for hardware without SIMD support. If the hardware natively supports intrinsics, the SIMDe calls are replaced by their native counterparts during compile time, therefore no performance penalties occur. If the hardware does not support specific instructions they are replaced by a combination of available vector instructions if possible or a scalar implementation with compiler hints to ease auto-vectorization. SIMDe functions can be used by simply prefixing regular intrinsics function names with `simde` [sim].

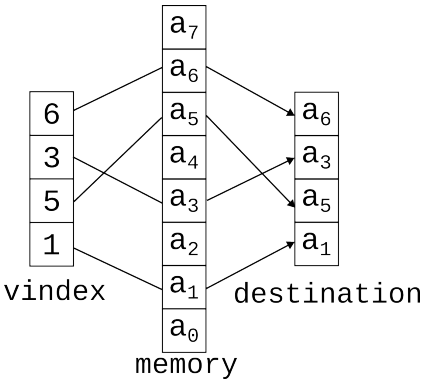
[2] `https://github.com/simd-everywhere/simde`



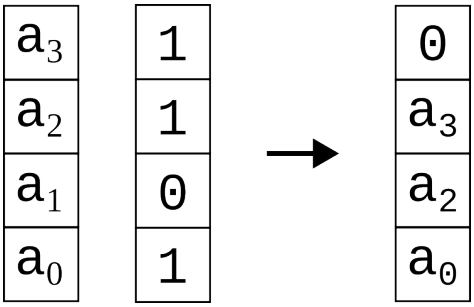
(a) Illustration of masked SIMD instructions. Values from `a` are only loaded if the corresponding mask bit is set to 1.



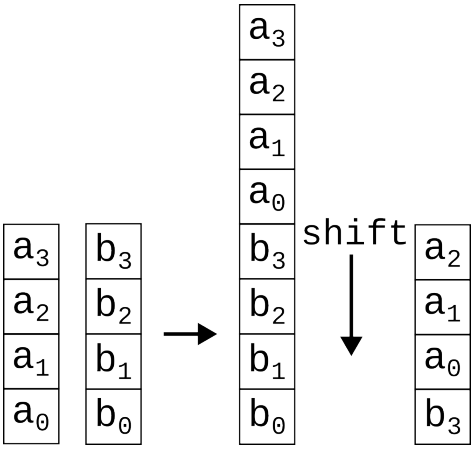
(b) Illustration of SIMD fmadd instructions. Multiplication and addition is combined into one operation.



(c) Illustration of SIMD gather instructions. `vindex` determines which elements are loaded from memory into the destination register.



(d) Illustration of SIMD compress instructions. Elements whose mask bit is set to 1 are compressed towards the lower end of the destination register.



(e) Illustration of SIMD alignr instructions. Input vectors are concatenated, then shifted towards the lower end of the vector.

Figure 3.3.: Visualization of a selection of SIMD operations.

4. Related Works

The Lennard-Jones potential is one of the most important potentials for molecular dynamics. However, work has been done to highlight the relevance of three-body interactions like the Axilrod-Teller potential [Mar01]. Several works also deal with efficient implementations of the Axilrod-Teller potential [BY13][CW00] but we are not aware of any focusing on SIMD implementations of the force kernel specifically.

Similarly, recent works regarding three-body interactions in AutoPas only deal with general algorithm approaches [Mar22], or traversals [Den24].

Furthermore, the expected low hitrate for the cutoff condition makes three-body functors an interesting area of research to achieve as much parallelism as possible.

Part II.

Implementation

5. Development of an Axilrod-Teller Functor without Intrinsics

To work towards a vectorized implementation of three-body potentials, a non-intrinsics version is required first. This will act both as an initial building block for an intrinsics version, as well as a baseline for performance evaluation.

As mentioned in Subsection 3.1.2, the functor requires implementations for one, two, and three different SoA structures. We will consider the triple functor using three different SoAs first.

The first step is to find particle triplets, whose particles are all pairwise within the cutoff range. This is accomplished through a triple nested loop as shown in Algorithm 1.

Algorithm 1: Main loop structure for the SoA triple functor

```
1  for i from 0 to soa1.size() − 1 do
2      for j from 0 to soa2.size() − 1 do
3          r²_ij ← squared distance between particles i and j
4          if r²_ij > r²_c then
5              continue
6          for k from 0 to soa3.size() − 1 do
7              r²_jk ← squared distance between particles j and k
8              if r²_jk > r²_c then
9                  continue
10             r²_ki ← squared distance between particles k and i
11             if r²_ki > r²_c then
12                 continue
13             calculateForce(i, j, k)
```

The distances between particles are calculated as the euclidean distance:

$$r_{ij}^2 = (x_j - x_i)^2 + (y_j - y_i)^2 + (z_j - z_i)^2$$

The cutoff criterium is applied to all pair combinations to match the existing AoS functor and the three-body traversal. This also allows to abort the loop early if the first two particles are not within cutoff.

For every particle triplet within the cutoff, forces are calculated and applied to particles in the first SoA. The implementation of the force formula is based on the pre-existing AoS functor. If newton3 optimization is enabled, forces applying to the other two particles are calculated using Newton's third law for the third particle and applied as well.

The single functor works conceptually the same, except that newton3 optimization is always applied as explained in Subsection 3.1.4.

For the pair functor, there are two `j`-loops depending on whether the second particle is chosen from the first or second SoA as shown in Algorithm 2.

Algorithm 2: Main loop structure for the SoA pair functor

```
1  for i from 0 to soa1.size() − 1 do
2      for j from i + 1 to soa1.size() − 1 do
3          r_ij^2 ← squared distance between particles i and j
4          if r_ij^2 > r_c^2 then
5              continue
6          ...
7      for j from 0 to soa2.size() − 1 do
8          r_ij^2 ← squared distance between particles i and j
9          if r_ij^2 > r_c^2 then
10             continue
11         ...
```

Due to the complexity of the inner loop, especially the cutoff condition, we actually expect very little auto-vectorization to be applied.

6. Vectorization of the Axilrod-Teller Functor

6.1. Vectorization via Masked Instructions

For vectorized code, the cutoff criterium can be applied via masked instructions, which are explained in Section 3.2. Vectorization happens across the innermost loop. The third SoA is processed `vecLength` elements at a time, where `vecLength` is the number of elements in a vector register. Since the number of elements in the SoA may not be a multiple of `vecLength`, there might be a remainder that has to be treated separately. For this, the largest multiple of `vecLength` that is still smaller than the size of the SoA is required, which can be efficiently computed if `vecLength` is a power of two by setting the $\log_2$ (`vecLength`) least significant bits of the SoA size to zero: `(soa3.size() & ~(vecLength - 1))`. The individual iterations and the remainder are handled by the SoA kernel, as seen in Algorithm 3.

Algorithm 3: Inner loop for the masked variant of the SoA triple functor.

```
1 k ← 0
2 while k < (soa3.size() & ~(vecLength - 1)) do
3 |   SoAKernel(i, j, k)
4 |   k ← k + vecLength
5 SoAKernelRest(i, j, k, soa3.size() - k)
```

The kernel first loads a batch of coordinates from the SoA as seen in Listing 6.1. For the rest kernel, a masked load is necessary to prevent illegal memory accesses when exceeding the size of the SoA.

```
1 const simde__m512d x3 =
2     remainderIsMasked
3         ? simde_mm512_maskz_load_pd(_masks[rest], &x3ptr[k])
4         : simde_mm512_load_pd(&x3ptr[k]);
```

Listing 6.1: Loading of coordinates into a vector register. y and z coordinates are handled analogously.

With the coordinates loaded, it is then possible to compute the squared distances between particles. The computation closely follows the non-intrinsics implementation, using intrinsics versions of scalar operations and fused multiply-add, which is shown in Section 3.2, where possible. Next, a bitmask is created by comparing the squared distances with the squared cutoff, as shown in Listing 6.2

```
1 const simde__mmask8 cutoffMask_jk =
2     simde_mm512_cmp_pd_mask(drjk2, _cutoffSquaredPd, SIMDE_CMP_LE_OS);
```

Listing 6.2: A cutoff mask is computed by comparing squared distances with the squared cutoff. The cutoff mask for particles i and k is computed accordingly

The two cutoff masks, the dummy mask, and potentially the rest mask are then combined into a single mask using `simde_mm512_kand`.

The Computation of the force again matches the non-intrinsics implementation, with the addition of fused multiply-add instructions. The mask is only needed once the force is added to the force buffer of the current particles, as shown in Listing 6.3. This process is then repeated for the second particle if newton3 optimization is enabled for the second particle.

```
1 fxiacc = simde_mm512_mask_add_pd(fxi, mask, fxi, fxiacc);
```

Listing 6.3: The computed force is added to the force buffer of the first particle. The second but not the third particle works in the same way.

If newton3 optimization for the third particle is enabled, its forces are updated directly in the SoA using `simde_mm512_mask_store_pd`.

```
1 const simde__m512d fxk_old = simde_mm512_maskz_load_pd(mask, &fx3ptr[k]);
2 const simde__m512d fxk_new = simde_mm512_sub_pd(fxk_old, fxk);
3 simde_mm512_mask_store_pd(&fx3ptr[k], mask, fxk_new);
```

Listing 6.4: Forces for the third particle are updated.

Forces for the first and second particles are accumulated in a vector register. Once a particle has been fully processed, this vector register needs to be reduced to a scalar value which can be added to the SoA by `simde_mm512_reduce_add_pd`.

Care is required for the single and pair functors since the third particle might be chosen from the same SoA as the first or the second. i and j are incremented by one per loop iteration, so when starting the third loop at $k = j + 1$, the first particle will no longer be properly aligned for aligned loads and stores, requiring potentially less efficient unaligned variants instead. This can be avoided by inverting the loop order to ensure the third loop starts at $k = 0$ as shown in Algorithm 4.

Algorithm 4: Inverting the loop order fixes alignment issues when loading values of the third particle into vector registers.

```
1 for i from soa.size() − 1 down to 0 do
2     for j from i − 1 down to 0 do
3         ...
4         for k from 0 to j − 1 do
5             ...
```

Using masked stores and loads ensures that interactions between particles outside the cutoff radius do not add to a particle's forces. However, the interactions are still calculated along the way and occupy slots in the vector register, which are thus effectively wasted. This is fine if only a few of the particle triplets in consideration are not within cutoff distance, but with decreasing hit rate the benefit of vectorization decreases as well. In the worst case, if only one element in the vector passes the cutoff check, the computation will be essentially scalar.

6.2. Vectorization via Gather/Scatter Instructions

A way to avoid the hit rate issues of the simple masked implementation is to delay force computations until the triplets that satisfy the cutoff conditions are determined. Algorithm 5 shows how this collection of triplets can be done using a `std::vector` to buffer relevant indices.

Algorithm 5: Collecting relevant indices for the third particle in the gather/scatter variant of the SoA triple functor.

```
1 indicesK ← ∅
2 for k from 0 to soa3.size() − 1 do
3     r²_jk ← squared distance between particles j and k
4     if r²_jk > r²_c then
5         continue
6     r²_ki ← squared distance between particles k and i
7     if r²_ki > r²_c then
8         continue
9     indicesK ← indicesK ∪ {k}
```

The indices in `indicesK` are then processed similarly to Algorithm 3, except that the kernel takes an offset with respect to the index buffer rather than the SoA as an argument.

Indices in the index buffer are consecutively in memory and can thus be loaded into a vector register with a simple load instruction. Particle coordinates, however, are spread over memory and instead require gather instructions explained in Section 3.2.

Listing 6.5 shows how the particles are loaded by first loading a batch of indices from the index buffer and then passing them to `simde_mm512_i64gather_pd`. Once again, masked instructions are used to avoid illegal memory accesses when processing the remainder of the index buffer.

```
1 const simde__m512i vindex =
2     remainderIsMasked
3     ? simde_mm512_maskz_load_epi64(_masks[rest], &indicesK[kStart])
4     : simde_mm512_load_epi64(&indicesK[kStart]);
5 const simde__m512d x3 =
6     remainderIsMasked
7     ? simde_mm512_mask_i64gather_pd(_zero, _masks[rest], indicesK, x3ptr, 8)
8     : simde_mm512_i64gather_pd(indicesK, x3ptr, 8);
```

Listing 6.5: Particle coordinates are gathered into a vector register.

After the coordinates have been loaded, the kernel proceeds like the simple masked variant but without the cutoff mask. If newton3 is enabled for the third particle, its forces need to be updated in the SoA directly, which faces the same issue as the loading of coordinates and therefore require gather and scatter instructions as shown in Listing 6.6.

```
1 const simde__m512d fxk_old =
2     remainderIsMasked
3     ? simde_mm512_mask_i64gather_pd(_zero, _masks[rest], indicesK, fx3ptr, 8)
```

```
4       : simde_mm512_i64gather_pd(indicesK, fx3ptr, 8);
5   const simde__m512d fxk_new = simde_mm512_add_pd(fxk_old, fxk);
6   if constexpr (remainderIsMasked) {
7     simde_mm512_mask_i64scatter_pd(fx3ptr, _masks[rest], indicesK, fxk_new, 8);
8   } else {
9     simde_mm512_i64scatter_pd(fx3ptr, indicesK, fxk_new, 8);
10  }
```

Listing 6.6: Forces for the third particle are updated using gather and scatter instructions.

6.3. Vectorization via Compress/Alignr

Another point of possible improvement is the collection of indices shown in Algorithm 5. We want to vectorize the collection process and also keep the indices in registers for faster access. This is done by first computing a cutoff mask like the masked kernel shown in Section 6.1.

Passing this mask to `simde_mm512_maskz_compress_epi64` compresses the indices towards the lower end of the vector register. Then `simde_mm512_alignr_epi64` is used to merge the new indices with the existing indices. If the vector capacity is exceeded, the kernel is called to process the current batch of indices. This procedure is shown in Algorithm 6 and visualized in Figure 6.1. The kernel is essentially the same as the one explained in Section 6.2, except it does not need to load the `vindex` from the vector buffer since the index register acts as `vindex` directly.

Algorithm 6: Compress/alignr index collection process

```
1  popCountMask ← count 1 bits in mask
2  newInteractionIndices ← simde_mm512_maskz_compress_epi64(...)
3  if numAssignedRegisters + popCountMask < vecLength then
4      interactionIndices ← simde_mm512_alignr_epi64(...)
5      numAssignedRegisters ← numAssignedRegisters + popCountMask
6  else
7      interactionIndices ← simde_mm512_alignr_epi64(...)
8      SoAKernel(i, j, interactionIndices)
9      interactionIndices ← simde_mm512_alignr_epi64(...)
10     numAssignedRegisters ← popCountMask − vecLength
```

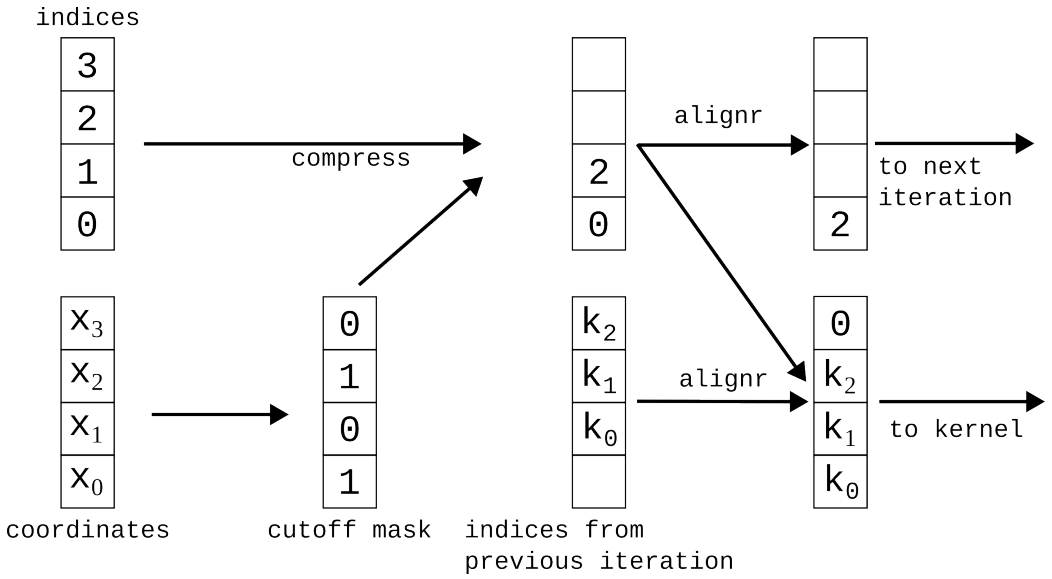


Figure 6.1.: Visualization of the compress/alignr index collection process. For simplicity vectors of length four are used for the visualization. The actual implementation uses vectors of length eight.

Part III.

Performance Evaluation

7. Performance Evaluation

7.1. Hardware Overview

Performance measurements were taken mainly on the CoolMUC-4 cluster of the Leibniz Rechenzentrum. Some additional measurements were taken on CoolMUC-2 to analyze how well the AVX512 implementation with SIMDe performs on non-AVX512 hardware. Table 7.1 gives an overview of the hardware features.

The AutoPas commit used for experiments is 42dfbe3.

	CoolMUC-4	CoolMUC-2
CPU	Intel®Xeon®Platinum 8380	Intel®Xeon®E5-2690 v3
CPU Architecture	Icelake	Haswell
Frequency	2.3 GHz	2.6 GHz
Vector Extensions	SSE, AVX, AVX2, AVX-512	AVX2

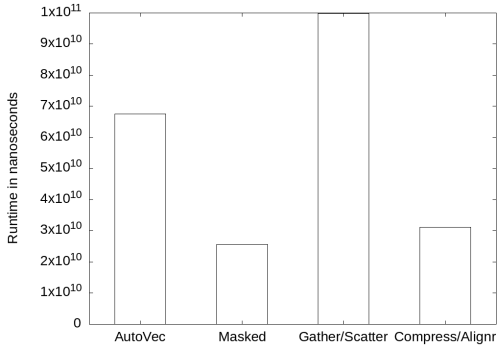
Table 7.1.: Hardware overview.

7.2. md-flexible

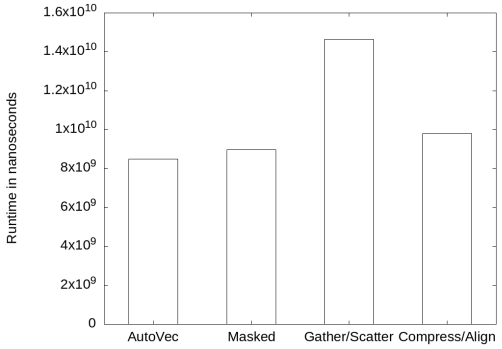
We used md-flexible to see the performance impact of the vectorized Axilrod-Teller potential within the entire simulation framework. For the simulation, particles were generated densely packed in a three-dimensional grid. This, combined with a relatively large cutoff of 5.0, ensures that there are enough particle triplets to properly see the effect of vectorization. However, such an idealized scenario will not always happen in real simulations. Therefore, another test was performed, spreading the particles further apart and using a cutoff of 2.5. The YAML[1] file used as input for the first scenario can be found in Appendix A. For the second scenario, `cutoff` was set to 2.5, `particle-spacing` to 1.7, and `box-length` to 100 in all dimensions.

In Figure 7.1a can be seen that the masked approach is actually the most promising, achieving a speedup relative to the auto-vectorized version of around 2.6. The compress/alignr implementation also performs well, achieving a speedup of approximately 2.2. Only the gather/scatter implementation performs worse than the auto-vectorized version. Comparing the gather/scatter and compress/alignr (which uses the same gather/scatter kernel) implementations also shows the significance of the index collection process, as the naive `std::vector` approach proves to be a major performance bottleneck. We suspect that the

[1] `https://yaml.org/`



(a) Runtime for a dense scenario with 22925 particles spread across a cube with sidelength 30 and a cutoff of 5.0. The masked implementation performs best, achieving a speedup of 2.6 compared to the auto-vectorized version.



(b) Runtime for a less dense scenario with 292876 particles spread across a cube with sidelength 100 and a cutoff of 2.5. Here, the auto-vectorized implementation performs best.

Figure 7.1.: Runtimes from tests using md-flexible.

index collection loop in the gather/scatter approach is not auto-vectorized, which combined with additional overhead from allocating the vector and potentially less efficient cache usage leads to poor performance.

In the sparse scenario, the auto-vectorized implementation performs slightly better than the masked and compress/alignr versions. A possible explanation for this is that due to the low number of particles per cell, the innermost loop does not have as many iterations, thus limiting the potential gain from parallelization.

7.3. Functor Benchmark

A major concern for the three-body functor is the low hit rate, meaning among the particle triplets under consideration only a few will contribute to force calculations. To properly analyze the impact of the hit rate, a benchmarking program based on AutoPasFunctorBench[2] was created. This program only runs the functor itself, without the full simulation framework. This gives us more control over the particle positions and allows us to test very specific scenarios. In particular, we placed all three cells at the same spatial coordinates, which allows us to fine-tune the hit rate by modifying the cell size relative to the cutoff.

Figure 7.2 shows speedups relative to the auto-vectorized version, using 300 particles per cell. Tests were performed for single, pair, and triple functors individually and split into whether newton3 is enabled.

Results are very similar for all considered options. As already seen in Section 7.2, the masked version generally performs best and is only slightly overtaken by the compress/alignr implementation for very low hit rates. Unsurprisingly, the masked variant also performs generally better, the higher the hit rate, which matches concerns addressed in Section 6.1. However, the same does unexpectedly not hold for the compress/alignr version, which sees

[2] `https://github.com/AutoPas/AutoPasFunctorBench`

only very little speedup at hit rates close to 100%. Finally, the gather/scatter implementation also performs marginally better for higher hit rates, but has no significant difference from the auto-vectorized version.

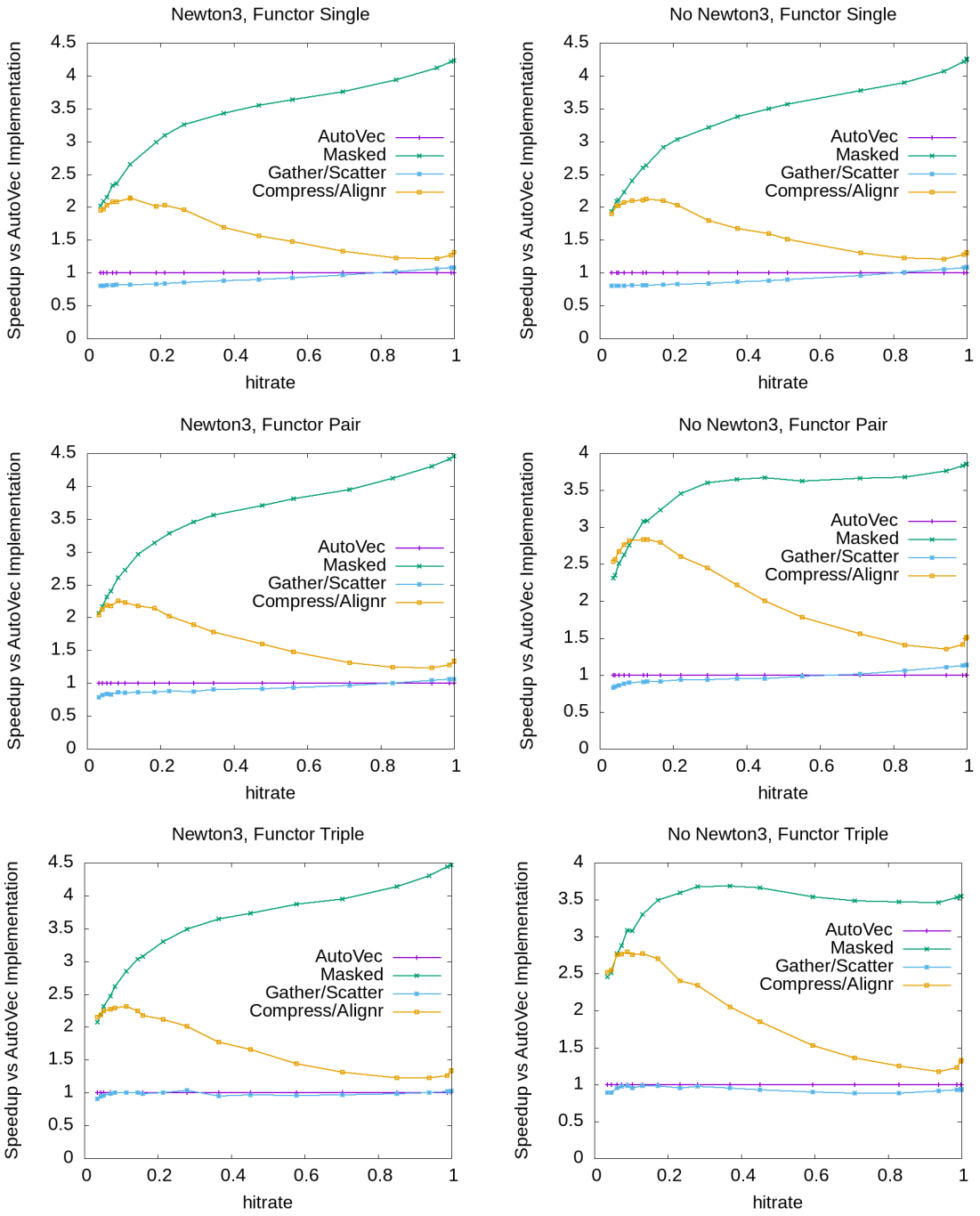


Figure 7.2.: Speedups observed from tests running the benchmarking program. Results are split into newton3 modes and single, pair, triple functors. Tests were performed with 300 particles per cell.

7.4. Portability to Non-AVX512 Hardware

Due to SIMDe, all vectorized implementations can also be directly used on hardware that doesn't support AVX512 instructions. To evaluate whether doing so would be reasonable performance-wise, the first scenario from Section 7.2 was repeated on CoolMUC-2 with `box-length` set to 20.

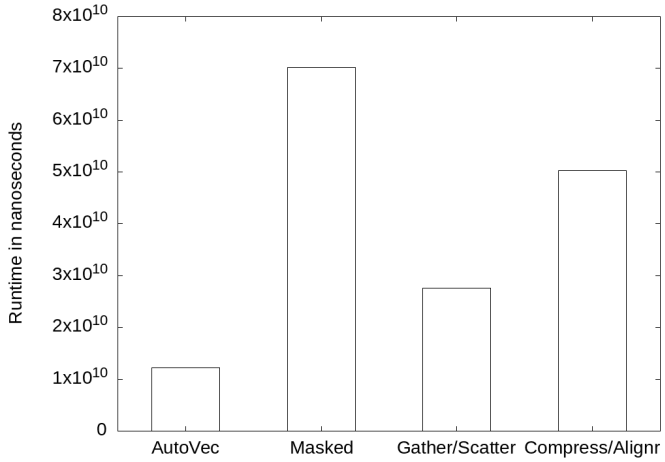


Figure 7.3.: Runtime for tests using md-flexible on a non-AVX512 CPU for a dense scenario with 6970 particles spread across a cube with sidelength 20 and a cutoff of 5.0.

Figure 7.3 shows that on non-AVX512 hardware, the intrinsics implementations perform worse than the auto-vectorized version. This means that despite SIMDe making the intrinsics code portable across different cpu architectures, it is still necessary to employ multiple implementations to achieve optimal performance on every hardware setup.

8. Conclusion

In this thesis, an SoA functor for the three-body Axilrod-Teller potential was implemented and successfully vectorized using SIMD intrinsics. Despite initial concerns about the efficiency of a simple masked approach to vectorization, it has proved to provide better performance than more elaborate schemes using gather, scatter, compress, and alignr instructions. Performance benchmarks also revealed that particle density plays a vital role for the efficiency of vectorization. How this factor comes into play for more meaningful simulation scenarios, remains an open question for potential future work.

Although the usage of SIMDe makes the vectorized implementations portable across different hardware architectures, it has not shown to be useful from a pure performance aspect, and an implementation split into AVX512 and non-AVX512 remains necessary to achieve optimal performance.

Part IV.

Appendix

A. YAML-input files

```yaml
container                       :  [LinkedCells]
functor-3b                      :  axilrod-teller
traversal-3b                    :  [lc_c01_3b]
newton3-3b                      :  [disabled]
data-layout-3b                  :  [SoA]

selector-strategy               :  Fastest-Absolute-Value
tuning-strategies               :  []
tuning-interval                 :  2000
tuning-samples                  :  3
tuning-max-evidence             :  10

cutoff                          :  5.0
cell-size                       :  [1.0]
deltaT                          :  0.0
iterations                      :  10
boundary-type                   :  [periodic, periodic, periodic]
fastParticlesThrow              :  false

Sites:
  0:
    mass                        :  1.
    nu                          :  0.073 # Value for Argon
Objects:
  CubeClosestPacked:
    0:
      box-length                :  [30, 30, 30]
      bottomLeftCorner          :  [0, 0, 0]
      particle-spacing          :  1.2
      velocity                  :  [0, 0, 0]
      particle-type-id          :  0
thermostat:
  initialTemperature            :  1.1
  targetTemperature             :  1.1
  deltaTemperature              :  0.5
  thermostatInterval            :  25
  addBrownianMotion             :  true

log-level                       :  warn
no-flops                        :  true
no-end-config                   :  true
no-progress-bar                 :  true
```

Listing A.1: YAML input file for the md-flexible test

List of Figures

List of Tables

Bibliography

[AT43] B. M. Axilrod and E. Teller. Interaction of the van der Waals Type Between Three Atoms. *The Journal of Chemical Physics*, 11(6):299–300, 06 1943.

[Bre00] D.W. Brenner. The art and science of an analytic potential. *physica status solidi (b)*, 217(1):23–40, 2000.

[BY13] W. Michael Brown and Masako Yamada. Implementing molecular dynamics on hybrid high performance computers—three-body potentials. *Computer Physics Communications*, 184(12):2785–2793, 2013.

[BZBP14] Hans-Joachim Bungartz, Stefan Zimmer, Martin Buchholz, and Dirk Pflüger. *Modeling and simulation*. Springer Undergraduate Texts in Mathematics and Technology. Springer, 2014.

[CW00] C.F. Cornwell and L.T. Wille. Parallel molecular dynamics simulations for short-ranged many-body potentials. *Computer Physics Communications*, 128(1):477–491, 2000.

[Den24] Nanxing Nick Deng. Implementation of linked-cells traversals for 3-body interactions in autopas. Bachelor's thesis, Technical University of Munich, Feb 2024.

[DM11] Jacob D. Durrant and James Andrew McCammon. Molecular dynamics simulations and drug discovery. *BMC Biology*, 9:71 – 71, 2011.

[Fly11] Michael Flynn. *Flynn's Taxonomy*, pages 689–697. Springer US, Boston, MA, 2011.

[Fot19] Nicola Fottner. Developing and benchmarking a molecular dynamics simulation using autopas. Bachelorarbeit, Technical University of Munich, Sep 2019.

[GSBN21] Fabio Alexander Gratl, Steffen Seckler, Hans-Joachim Bungartz, and Philipp Neumann. N ways to simulate short-range particle systems: Automated algorithm selection with the node-level library autopas. *Computer Physics Communications*, 273:108262, 2021.

[GST+19] Fabio Alexander Gratl, Steffen Seckler, Nikola Tchipev, Hans-Joachim Bungartz, and Philipp Neumann. Autopas: Auto-tuning for particle simulations. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Rio de Janeiro, May 2019. IEEE.

[int] Intel intrinsics guide. `https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html`. Accessed: 2024-05-12.

[LJ24] J. E. Lennard-Jones. On the determination of molecular fields. ii. from the equation of state of a gas. *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character*, 106(738):463–477, 1924.

[Mar01] Gianluca Marcelli. The role of three-body interactions on the equilibrium and non-equilibrium properties of fluids from molecular simulation. 2001.

[Mar22] David Martin. A comparison of three-body algorithms for molecular dynamics simulations. Bachelor's thesis, Technical University of Munich, Nov 2022.

[Mic79] Ronald E. Mickens. Long-range interactions. *Foundations of Physics*, 9(3):261–269, 1979.

[New87] Issac Newton. *Philosophiae naturalis principia mathematica*. 1687.

[SGH+21] Steffen Seckler, Fabio Gratl, Matthias Heinen, Jadran Vrabec, Hans-Joachim Bungartz, and Philipp Neumann. Autopas in ls1 mardyn: Massively parallel particle simulations with node-level auto-tuning. *Journal of Computational Science*, 50:101296, 2021.

[sim] Simd everywhere. `https://github.com/simd-everywhere/simde`. Accessed: 2024-05-12.

[Ver67] Loup Verlet. Computer "experiments" on classical fluids. i. thermodynamical properties of lennard-jones molecules. *Phys. Rev.*, 159(1):98–103, Jul 1967.