SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Implementation and Vectorization of the Mie Potential in AutoPas

Kay Cole

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor’s Thesis in Informatics

Implementation and Vectorization of the Mie Potential in AutoPas

Implementierung und Vektorisierung des Mie Potentials in AutoPas

Author:	Kay Cole
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Samuel James Newcome, M.Sc.
Submission Date:	15.01.2024

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

Munich, 15.01.2024 Kay Cole

Acknowledgments

First of all, I would like to thank both my advisor Sam and Prof. Dr. Hans-Joachim Bungartz for giving me the opportunity to conduct my Bachelor Thesis at the Chair for Scientific Computing. Thanks a lot for the great supervision, the fast answers to any arising questions and for offering me the opportunity to work on this interesting topic in the first place.

I am grateful for the Leibniz-Rechenzentrum in Munich and the Helmut-Schmidt Unversität in Hamburg for access to their computational resources.

A big thank you also goes to my friends Simon, Felix and Jonas for proof-reading parts of this thesis, for their helpful comments and generally all the support they gave me.

Last of all, I would like to thank my family. Thanks a lot for the constant encouragement and support you gave me throughout the course of this thesis.

Abstract

Molecular Dynamics simulations are a crucial tool for understanding the dynamic behaviour of molecular systems. They are often based on force calculations between particle pairs described by the Lennard-Jones potential. While the Lennard-Jones potential suffices to correctly model interactions in a broad range of applications, its fixed exponents for attractive and repulsive forces fail in specialized applications such as the accurate description of dense fluids. Here, the Mie potential as a more flexible, generalized form of the Lennard-Jones potential offers a solution. When simulating the interactions for a large number of particles, a lot of computational effort is needed. Therefore, parallelization techniques such as vectorization are essential for enhancing performance in molecular dynamics. The goal of this thesis is the implementation of force calculations based on the Mie potential in the particle simulation library AutoPas. To reduce computational cost, vectorization techniques are applied. Different implementations are introduced and analyzed. For these implementations, the influence of different exponents in the Mie potential on the performance is compared.

Contents

1. Introduction

In recent decades, Molecular Dynamics (MD) simulations have become a crucial tool for gaining new insights into the dynamic behaviour of molecular systems at an atomic and molecular level[1]. The ability of MD simulations to provide detailed insights into structural dynamics and kinetics of diverse molecular systems makes them significant for many fields ranging from material sciences[2] to the simulations of biomolecules[1]. Particularly in the field of drug development, MD simulations are well integrated. This helps making the development of novel drugs more efficient and less costly by reducing the number of real-life experiments needed during the development process[3].

MD simulations help understanding, interpreting and even replacing experiments by modelling physical interactions of many individual particles at the atomic or molecular scale. For calculating these interactions, the Lennard-Jones potential[4] is commonly used, which models van der Waals interactions between two particles.

However, while there are many successful applications of the Lennard-Jones potential, it fails to accurately model more complex interactions. Especially the repulsive force of the Lennard-Jones potential may not accurately depict the forces at hand, with examples in the modelling of noble gases[5], dense fluids[6] or coarse-grain models of larger molecules[7]. A possible solution is the implementation of a force calculation based on a more general force potential, with the Mie potential[8] being the method of choice. The Mie potential offers more flexibility in the choice of attractive and repulsive forces, allowing for the modelling of forces that are more similar to those observed in the experiment at hand.

However, due to the high numbers of computations needed for the particle interactions during a MD simulation, the computational costs of each calculation has to be considered. Besides a computationally effective implementation, parallel programming can accelerate MD simulations even further. In this context vectorization techniques, like the usage of Single Instruction Multiple Data (SIMD) instructions, are an important tool to speed up MD simulations significantly [9].

This thesis aims to implement force calculations based on the Mie potential in MD simulations with a focus on applying vectorization techniques. To this end, different implementations of the Mie potential are explored and compared regarding their computational cost. Then these approaches are implemented in the particle simulation library AutoPas[10] and finally the performances of the different implementations are evaluated.

2. Theoretical Background

2.1. Fundamentals of Molecular Dynamics Simulations

Molecular Dynamics simulations calculate the movement of particles from a given initial position x and velocity v in a defined time period. This is done by discretising in time and solving Newton's equations of motion at each time step:

$$\mathbf{v} = \frac{d\mathbf{x}}{dt}, \quad \mathbf{a} = \frac{d\mathbf{v}}{dt} \tag{2.1}$$

The velocities and positions are then updated at every time step. In order to calculate the acceleration Newton's second law is used to describe the relation between the force F acting on a particle with a mass m and its acceleration a.

$$\mathbf{F} = m \cdot \mathbf{a} \tag{2.2}$$

To calculate the acceleration one needs to compute the total force acting on a particle at every simulation step. The total force is equal to the sum of all individual forces. The individual force is defined by the negative gradient of the corresponding potential U:

$$\mathbf{F} = -\nabla U \tag{2.3}$$

2.2. Force Potentials

Several potentials exist that can be used in different contexts, such as the gravitational or coulomb potential [11, pp. 28–29]. In Molecular Dynamics a commonly used potential is the Lennard-Jones potential.

2.2.1. Lennard-Jones Potential

The Lennard-Jones potential [4] is a short range pair potential that describes the interactions between two electronically neutral particles. It is defined as the following:

$$U(r) = 4\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right] \tag{2.4}$$

Here, r represents the distance between the interacting particles, ϵ defines the depth of the potential well and σ, also known as the particle size, the finite particle distance at which the potential is zero.

The Lennard-Jones (LJ) potential consists of two terms. The first term which scales with r^{-12} describes attractive van der Waals interactions between particles whereas the second term, scaling with r^{-6}, describes repulsive interactions stemming from the overlap of electron orbitals of the particles at short distances. The different exponentiation of the interactions results in strong repulsive interactions dominating at short distances below σ and attractive interactions dominating at larger distances. This creates a distinct shape for the resulting energy curve featuring an energy minimum with a depth of ϵ at a certain distance between particles (see Figure 2.1).

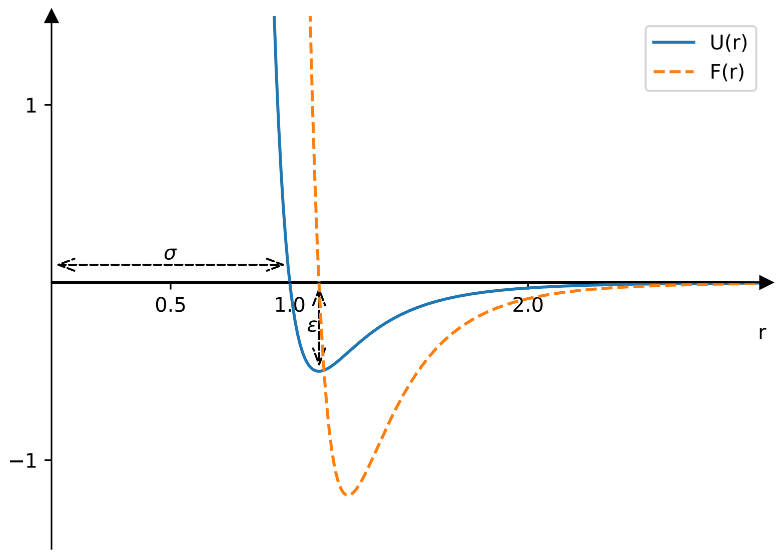

Figure 2.1.: The Lennard-Jones Potential. For large distances r the force approaches zero

σ and ϵ typically are particle specific. To describe the potential between two particles of different types special mixing rules need to be used. One such rule is the Lorentz-Berthelot combining rules[12, 13]:

$$\epsilon_{ij} = \sqrt{\epsilon_i \epsilon_j} \quad \sigma_{ij} = \frac{\sigma_i + \sigma_j}{2} \tag{2.5}$$

While there is a physical justification to use the exponent 6 in the Lennard-Jones potential to describe repulsive forces [14], the exponent 12 for attractive forces is primarily chosen to allow for efficient computation. This results in inaccuracies when applying the potential to specialized applications such as the prediction of properties of noble gases[5] or of saturated liquid viscosities of non-spherical particles[15]. Here, the Mie potential[8] provides a suitable alternative.

2.2.2. Mie Potential

The Mie potential[8] is a generalization of the Lennard-Jones potential. As such it is often also referred to as the general Lennard-Jones potential or the (n,m) Lennard-Jones potential. Instead of using fixed exponents of 12 and 6 to model attractive and repulsive interactions, it uses flexible exponents of *n* and *m*:

$$U(r) = C\varepsilon \left[\left(\frac{\sigma}{r}\right)^n - \left(\frac{\sigma}{r}\right)^m \right] \tag{2.6}$$

with

$$C = \frac{n}{n-m} (\frac{n}{m})^{\frac{m}{n-m}} \tag{2.7}$$

where $n > m > 3$ [16]. Here *n* and *m* represent positive integers. For the exponents *n* = 12 and *m* = 6, this corresponds to the Lennard-Jones potential. Its flexible exponents, however, make the Mie potential a better choice in potential for certain applications. Especially the exponent *n*, characterizing the distance dependency of attractive interactions, is often varied when applying the Mie potential[17, 18, 19].

Figure 2.2 shows the (n,6) Mie potential for different *n* values. Increasing *n* results in a shift of the potential minimum to lower distances while simultaneously increasing the steepness of the repulsion, effectively narrowing the potential.

For example, the pair potential of noble gases is only accurately described by the (n,6) Mie potential with *n* values ranging from *n* = 11 for Neon to *n* = 14 for Krypton[17]. Analogously, for modelling phase equilibria of branched alkanes, alkenes and perfluoralkanes, only the (n,6) Mie potential with *n* values between 14-16, 16 and 36-44 yielded accurate results[18, 19]. These examples highlight the advantages of using the Mie potential instead of the Lennard-Jones potential when a fixed exponent of 12 fails to describe attractive interactions accurately.

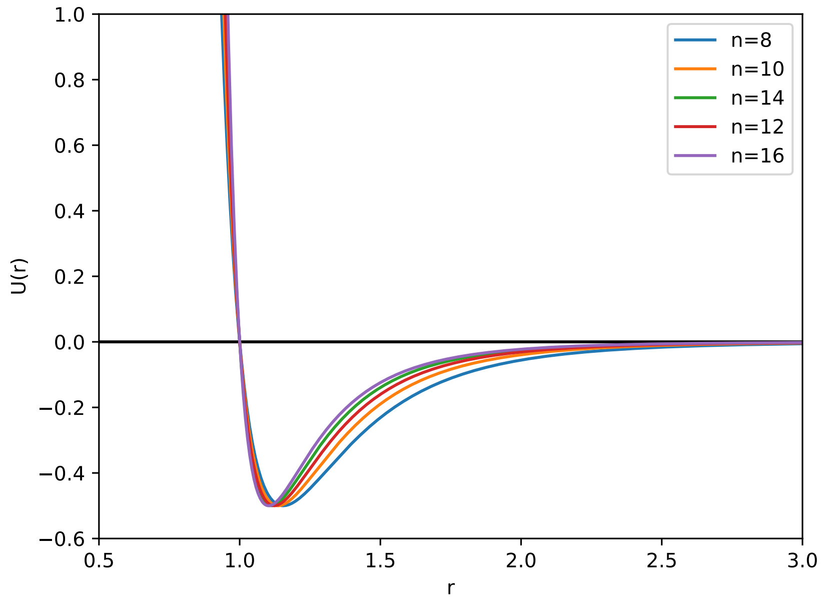

Figure 2.2.: (n,6) Mie potential with different repulsive exponents. The attractive exponent m is fixed at 6.

2.2.3. Cutoff

To calculate the correct force of a particle in each simulation step, one needs to consider every interaction between particles inside the domain. This gets computationally expensive for large numbers of particles. For a simulation based upon a pairwise potential of N particles, this approach would lead to a computational complexity of $\mathcal{O}\left(N^2\right)$.

The computational cost can be significantly reduced by the introduction of a cutoff radius r_c. The pairwise forces are then only calculated for particle pairs with distances smaller than the cutoff radius. This is based upon the observation that for short-range potentials the forces between particles converge to 0 for larger distances. While this comes with a loss of accuracy, the effects are very small when the chosen r_c is large enough [11]. However, using different exponents for the Mie potential, can alter the potential curve and thus requires choosing the cutoff radius carefully. For instance, decreasing the repulsive exponent n leads to a slower convergence of the potential to 0 (see Figure 2.2). This increases the forces for longer distances, which requires a higher cutoff to be set.

2.3. Efficient Exponentiation

An efficient computation of the pairwise force calculation is desirable to calculate a large number of particle interactions. The fixed exponents in the Lennard-Jones Potential (6 and 12) allow for optimized exponentiation. In contrast, the Mie potential's variable exponents require a different approach.

The simplest way to compute x to the power of n, is to multiply x n times with itself. This requires $n-1$ multiplications leading to a computational complexity of $\mathcal{O}(n)$.

Our goal is to minimize the number of multiplications for the calculation of x^n to maximize efficiency. This is known to be equivalent to minimizing the number of additions to compute n starting from 1. This problem is modelled by addition-chains. An addition-chain for the number n is a sequence of positive integers starting from one:

$$[1, k_2, k_3, k_4, .., k_i, .., n] \tag{2.8}$$

where each number $k > 1$ in the list is equal to the sum of any two previous numbers in the sequence and the final number is n[20]. An addition-chain for n has length l, where l is one less than the number of elements in the chain.

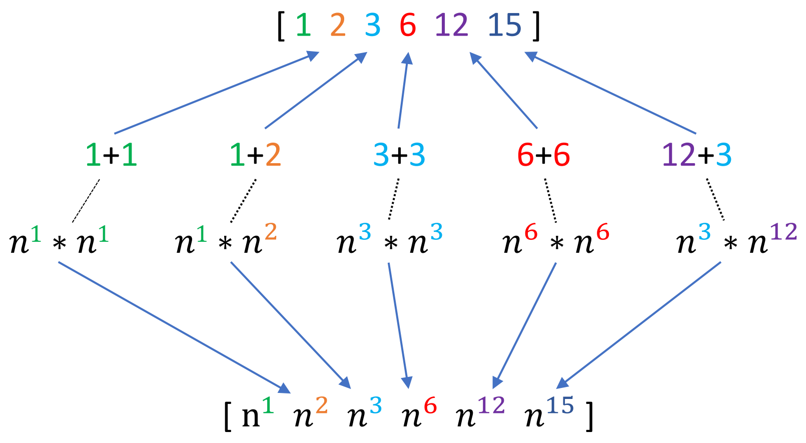

Figure 2.3.: Additive Exponentiation. A minimal addition chain for 15 is shown. Five multiplications are necessary to compute n^{15}.

The calculation of the power x^n can then be done using an addition-chain for n. The shorter the chain, the fewer additions are needed to compute n starting from 1, or the fewer multiplications are needed to compute x^n starting from x. How a chain is used to calculate a power is shown in more detail in Figure 2.3.

Then, instead of n multiplications with the naive loop approach only l multiplications are needed for the exponentiation when using an addition-chain for n with length l.

In the context of the Mie potential two powers with the same base are computed. This enables to combine calculations for both powers.

An addition-sequence is an addition chain with the constraint that it contains a set of given numbers [21]. Say n is the biggest integer in the set s, then $[1, k_2, k_3, ..., n]$ is a addition sequence of s if:

$$\forall m \in s : m \in [1, k_2, k_3, ..., n] \tag{2.9}$$

and each number $k > 1$ in the sequence is equal to the sum of any two previous numbers of the sequence.

In the context of the Mie potential using an addition-sequence for $\{m, n\}$ is of interest. The length of the addition-sequence indicates the number of multiplications needed to compute both x^m and x^n.

In the following different algorithms for computing powers will be introduced.

2.3.1. Binary Exponentiation

Binary exponentiation, also known as exponentiation by squaring, is a common method to compute integer powers efficiently [20]. To compute x^n it repeatedly

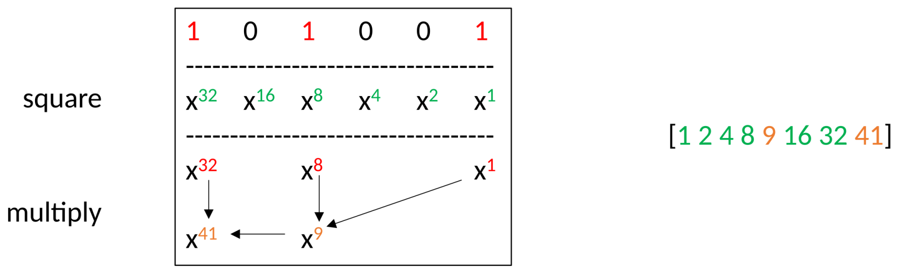

Figure 2.4.: Calculating x^{41} by squaring and multiplying. The product of the red numbers is the solution. To compute the red numbers x is repeatedly squared (green colours). On the right is the corresponding addition chain. 8 Multiplications are required to compute x^{41}

squares x for every digit of the binary expansion of n. If the digit is a 1 it also multiplies the current value of x with an accumulator. The accumulator is initialized with 1. An example for computing x^{41} is shown in Figure 2.4.

The number of multiplications needed to compute the power is the number of digits minus 1 of the binary representation of the exponent, added to the number of 1's in the exponent. As the exponent e has $\lfloor \log_2 e \rfloor$ binary digits, the computational complexity is $\mathcal{O}(\log n)$.

2.3.2. Double Addition-Chain

Now a method will be introduced that computes two exponentiations x^m and x^n simultaneously. For that an addition sequence for $\{m, n\}$ will be constructed, which will be called a **double addition-chain**. The concrete computation of the exponentiations then uses a representation of said chain. The algorithm follows this paper: [22]

Each element x_i of the double addition-chain is either the sum of the two elements before it $x_{i-2} + x_{i-1}$, a doubling of the element before it $2 \times x_{i-1}$, or the sum of 1 and the element before it $1 + x_{i-1}$. It will be seen that by doing this only three variables are needed when calculating x^m and x^n: One to store x, and two accumulators for x^m and x^n. Furthermore, when calculating the exponentiations, in each step one of the accumulators is multiplied either by x, by itself or by the other accumulator. At first the algorithm applies the inverse rules to $(m, n))$ until $(0, 1))$ is reached. The order of the applied rules is stored, and can later be used to construct the double-addition chain or to compute the exponentiations (x^m, x^n) and implement the exponentiation.

Construction of the Double Addition Chain

Starting with $(\alpha, \beta) = (m, n)$ a sequence is constructed by recursively applying the inverse operations until the pair $(0, 1)$ is reached. Each number from an intermediate pair $j \in (\alpha_i, \beta_i)$ obtained in this way will be part of the double addition-chain. The inverse operations are either a decrement, a division by two, or a subtraction of one element from the other. The inverse rules are shown in Equation 2.10.

$$(\alpha_{i+1}, \beta_{i+1}) = \begin{cases} \left(\alpha_i, \frac{\beta_i}{2}\right) & \text{if } \alpha_i \leq \frac{\beta_i}{2} \wedge \beta_i \bmod 2 = 0 \\ \left(\alpha_i, \frac{\beta_i - 1}{2}\right) & \text{if } \alpha_i \leq \frac{\beta_i}{2} \wedge \beta_i \bmod 2 = 1 \\ (\beta_i - \alpha_i, \alpha_i) & \text{if } \alpha_i > \frac{\beta_i}{2} \end{cases} \quad (2.10)$$

Let k be the recursion depth where $(\alpha_k, \beta_k) = (0,1)$. Two k-bit vectors τ, ν will be used to represent the sequence of additions that construct the chain.

$$\tau_i = \begin{cases} 0 & \text{if } \alpha_{k-i} \leq \frac{\beta_{k-i}}{2} \\ 1 & \text{if } \alpha_{k-i} > \frac{\beta_{k-i}}{2} \end{cases} \tag{2.11}$$

$$\nu_i = \beta_{k-i} \mod 2 \tag{2.12}$$

Starting from $(0,1))$ a sequence $(m_i, n_i)_i$ that ends with (m, n) will then be constructed. This is done by starting from $(0,1))$ and processing the information encoded in τ and ν:

$$(m_{i+1}, n_{i+1}) = \begin{cases} (m_i, 2n_i) & \text{if } \tau_{i+1} = 0 \wedge \nu_{i+1} = 0 \\ (m_i, 2n_i + 1) & \text{if } \tau_{i+1} = 0 \wedge \nu_{i+1} = 1 \\ (n_i, n_i + m_i) & \text{if } \tau_{i+1} = 1 \end{cases} \tag{2.13}$$

It can be seen that the information in ν is only relevant, if $\tau_{i+1} = 0$. This allows merging τ and ν by inserting ν_i into τ if $\tau_i = 0$. The specific exponentiation uses the merged bit-vector for computing x^m and x^n as will be further detailed in the implementation section [22].

2.4. Vectorization

Vectorization can further accelerate force calculations[9]. It is a parallel computing method that processes multiple data elements simultaneously within a single instruction, known as **S**ingle **I**nstruction **M**ultiple **D**ata (SIMD). The same instruction is applied to all data elements at once, thus reducing the number of instructions needed. Most modern CPUs support SIMD through special vector-registers and vector-instructions that come with instruction set extensions. The acceleration compared to non-vectorized calculations depend on the size of the register, and the size of the datatype. A vectorized multiplication is illustrated in Figure 2.5. Vectorization can be achieved by different approaches.

2.4.1. Autovectorization

One option is to rely on the compiler to automatically detect suitable code-segments and vectorize them. In this case, standard scalar code without any additional modifications is sufficient, which keeps the code easy to read and maintain while ensuring functionality independent of the used processor. The

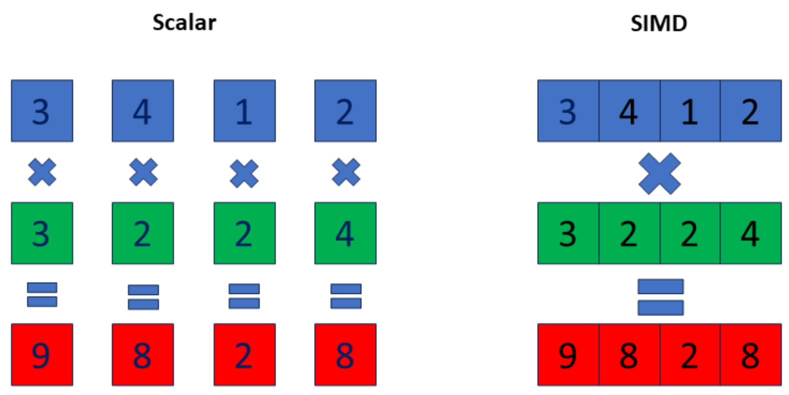

Figure 2.5.: Example of a SIMD operation. Assuming a vector-register size of 256 bit, four 64bit floating-point values can processed using a single instruction instead of four.

processor must support vectorization and the required compile-flags need to be set. There is no guarantee that autovectorization will be successful as the compiler needs to ensure the correctness of the vectorized code. In more complex scenarios it may fail due to data dependencies or other reasons. Compiler directives can provide additional information to the compiler, increasing chances of successful vectorization. For instance, OpenMP can be used to indicate that a loop can be vectorised[1].

```
#pragma omp simd
for(i=0; i<N; i++){
 // do stuff
}
```

Listing 2.1: OpenMP SIMD directive indicating that the for loop can be vectorized.

However the vectorised code may not be optimal.

2.4.2. Vector Intrinsics

AVX

AVX (**A**dvanced **V**ector **E**xtensions) intrinsics were introduced by Intel in 2011. They offer support for 256 bit and 128 bit sized registers. The datatypes and

[1]`https://www.openmp.org/spec-html/5.0/openmpsu42.html`

functions follow the following pattern:

- Datatypes: **__[size][type]**
- Functions: **_mm[size]_[instruction]_[type]**

SVE

SVE (Scalable Vector Extensions)[23] intrinsics are defined in the arm_sve.h header file. The vector length for SVE is variable and can range from 128 bit up to 2048 bits. The functions and arithmetic datatypes follow the schema:

- Datatypes: **sv[type]_t**
- Functions: **sv[instruction][_disambiguator][_type0][_type1][_predication]**

2.5. Compile-time Programming

The calculation of the forces based on the Mie potential depends strongly on the exponents of the potential. If the exponents are known at compile time, the compiler can make additional optimizations which potentially can increase efficiency. Compile-time programming allows to evaluate code segments at compile-time rather than at run-time. Different C++ features that enable compile-time programming will be briefly reviewed.

Constexpr

Variables and functions marked with the keyword **constexpr** can be evaluated at compile-time. **Constexpr** variables are constant and must be initialized with a **constant** expression. Constexpr functions must not have any side-effects. Conditional expressions can be evaluated at compile-time when using **constexpr-if** statements.

Template Metaprogramming

Templates are evaluated and instantiated at compile time. This allows for more complicated compile-time programming. It is known that template metaprogramming in C++ is turing complete [24]. The following snippet shows a basic example.

```
template <int N>
 constexpr int fac(){
    if constexpr(N==0)
```

```
        return 1;
    else
        return N* fac<N-1>();
}
int main(){
    std::cout << fac<12>() << std::endl;
}
```

Listing 2.2: A short example function using constexpr if and template metaprogramming. The faculty will be computed at compile time.

3. AutoPas

In this chapter AutoPas is introduced and existing structures that are relevant for this thesis are discussed.

A tool of choice for efficient MD simulations is the C++ library AutoPas[1]. To the user AutoPas acts a black box that chooses the optimal algorithm for the provided scenario.[25] It achieves this by testing all available combinations of algorithms and containers periodically and selecting the fastest one. It does this repetitively during the simulation as the best performing algorithm might change with time.

3.1. Particle Container

To ensure that only particle pairs with distance less than the specified cutoff radius are included in the pairwise force calculation, all particle pairs with distance within the cutoff must be determined at each simulation step. For that AutoPas implement different algorithms.

1. **Direct Sum** The most straightforward approach is to store all N particles in a single container. For each particle the distances to all other particles are computed. This is visualized in Figure 3.1 where the red particle is the particle for that the pair-wise forces are currently calculated. The arrows stand for the distance-calculations. This leads to a complexity of $O(N^2)$. Thus it is only suitable for small N where the number of computations required is small and not outweighed by the complexity of other methods.[25]

2. **Linked Cells** The Linked Cells (LC) method divides the simulation domain into a grid of cells. In most cases a cell width r_c of equal or bigger than the cutoff radius is chosen. Particles distances are then only calculated between particles within a cell and adjacent cells, thus reducing the complexity to $\mathcal{O}(N)$. The neighboring cells are shown in blue in Figure 3.1 and the particle cell is marked red. While the memory overhead for the cell structure is quite small, an additional computational overhead for moving the particles in between cells is required[25]. Particles of the same cell can be placed

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

continuously in memory, improving vectorization capabilities. However the LC method still computes a lot of unnecessary distance calculations, with a hitrate of below 16% in neighboured cells[10]:

$$\frac{Cutoff\ \ volume}{Search\ \ volume} = \frac{\frac{4}{3}\pi r_c^3}{(3r_c)^3} \approx 0.155 \tag{3.1}$$

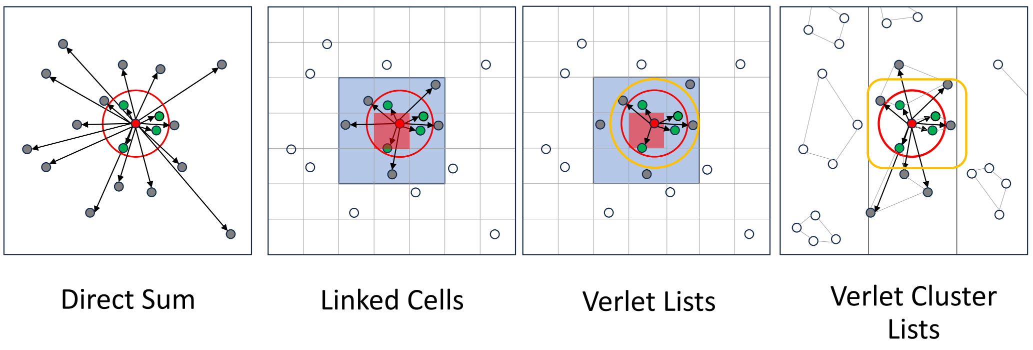

Figure 3.1.: Comparison of different particle containers. Interactions are computed for the red particle. Particles within the cutoff are green, particles that are potentially within the cutoff radius are grey. White particles are out of reach for the red particle. Adapted from [25]

3. **Verlet Lists** Each particle stores a list of references to all neighbouring particles within a certain interaction radius $r_c + \Delta s$ where Δs is called Verlet skin (yellow circle in Figure 3.1)[25]. When a particle leaves the interaction radius, the neighbour lists need to be rebuilt for each particle, which is expensive. Due to the introduction of the Verlet skin this only has to be done every n^{th} iteration. To facilitate the rebuilding of the lists AutoPas builds the Verlet List container upon the Linked Cell container. Only the distances of particles within the interaction radius has to be evaluated. With $\Delta s = s * r_c$

the hitratio increases to[10]:

$$\frac{Cutoff\ \ volume}{Search\ \ volume_{VL}} = \frac{\frac{4}{3}\pi r_c^3}{\frac{4}{3}\pi r_c(1+s)^3} \approx \frac{1}{1+s^3} \tag{3.2}$$

For $s = 0.15$ the probability evaluates to $\frac{1}{1+0.15^3} \approx 0.658$ which is a far better value than in Linked Cells. However storing the lists for all particles result in a big memory overhead. Based on the lists it is not possible to infer whether two particles are next to each other in memory. This can lead to poor caching and vectorization difficulties[25].

4. **Verlet Cluster List** is built upon Verlet Lists and was developed by Páll and Hess[26]. Neighbouring Particles are put together into cluster of size M. Instead of particles the neighbour lists store references to clusters, thus reducing the size of the lists and the total number of lists by a factor of $\frac{1}{M}$. For the distance calculations all particles of the clusters in the neighbour lists are considered. Choosing M as a multiple of the processors vector length allows for efficient vectorization. A downside is the rather complex implementation and a lower hitrate because of the increased interaction radius.

3.2. Data Layouts

There are two common options for storing particle data in memory that are displayed in Figure 3.2.

- **Structure of Arrays (SoA)** stores each particle property (e.g. x-position, velocity) in a separate array. This enables to load sequential data (of one particle property) from memory in one load operation, which is of advantage for vectorization.
- **Arrays of Structure (AoS)** stores an array of particle objects. Each object contains all of the particle's properties. This can be advantageous if one wants to load all properties in a non sequential manner. However for vectorization we want to load contiguous data of a single property from memory, which makes AoS not suitable for vectorization.

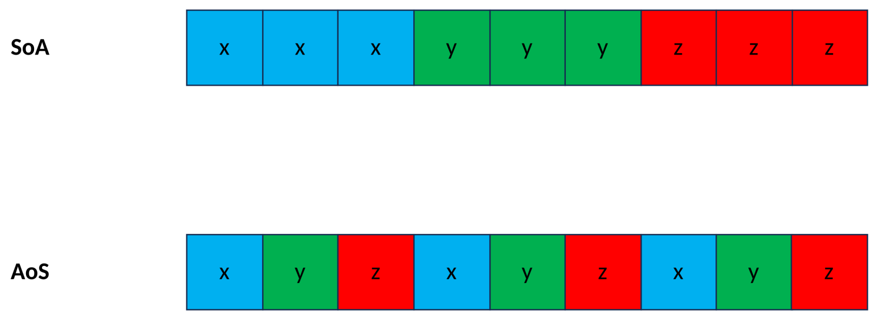

Figure 3.2.: Comparison of the SoA and AoS data layout by storing coordinates. For simplicity only the storage of three coordinates is displayed.

3.3. Overview of Relevant Classes

With MD-flexible, AutoPas includes an executable for running particle simulations. Force calculations used in MD-flexible so far are based solely on the Lennard-Jones potential. The following section introduces its most relevant classes which have to be adapted to also use the more generalizable Mie potential for force calculations.

3.3.1. Functor

For calculating pairwise interactions between multiple particles, AutoPas introduces the class Functor.h. It includes four functors:

- **AoSFunctor** The AoS-Functor exists for situations where the AoS data layout is used. It calculates the pairwise interaction between two particles. However for reasons described in Section 3.2 this functor is not vectorizable.
- **SoAFunctorSingle** This Functor calculates all the pairwise interactions of a single SoA-Structure. When using Linked Cells this corresponds to computing the forces of Particles within the same cell.

- **SoAFunctorPair** When calculating forces between two adjacent cells using Linked Cells the SoAFunctorPair-Functor need to be used. It takes two SoAs as parameters and calculates the pairwise interactions between them.
- **SoAFunctorVerlet** The Verlet Functor used by the Verlet List container takes an SoA of particles, an index for the current particle, and a (neighbour) list of particles for the neighbours of the current particle. It then computes all pairwise interactions between the current particle and its neighbours.

To use specific force potentials in simulations, a custom functor class needs to be implemented that provide implementations of these functors.

Lennard-Jones Functor

AutoPas already contains a variety of custom functors for the Lennard-Jones potential using Autovectorization, AVX-intrinsics and SVE-intrinsics. In the heart of the functors the same sequence of instructions is used to calculate the force, that is displayed in algorithm 1.

Algorithm 1: LennardJonesForce

```
1 $lj6 \leftarrow \sigma^2 * \frac{1}{r^2}$
2 $lj6 \leftarrow lj6 * lj6 * lj6$
3 $lj12 \leftarrow lj6 * lj6$
4 $lj12m6 \leftarrow lj12 - lj6$
5 $fac \leftarrow \varepsilon * 24 * (lj12 + lj12m6) * \frac{1}{r^2}$
6 $force \leftarrow dr * fac$
7 **return** $force$
```

Figure 3.3.: Sequence of instructions to calculate the forces, given σ, r and ε using the LJ potential

ParticlePropertiesLibrary

This class contains the physical properties of all molecule types. It also stores the mixed values for the force calculations between the different types. We will extend the library for the Mie potential.

4. Implementation

Three functor classes of the Mie potential with different vectorization techniques are implemented: One using AVX vector intrinsics, one using SVE vector intrinsics and one using autovectorization with OpenMP. Additional, for each functor, a variant is offered where the exponents are provided at compile time. These functors compute the exponentiations using three different algorithms.

For a more effective comparison with the Lennard-Jones functors, most of the structure of the LJ functors will be adopted. Where necessary changes will be introduced. The primary difference to the LJ functors will be the computation of the forces due to the variable exponents.

The formula for the force computation of the Mie potential is as follows:

$$F_{Mie}(r) = C\varepsilon\left[n\left(\frac{\sigma}{r}\right)^n - \left(m\frac{\sigma}{r}\right)^m\right]\frac{1}{r} = \left[C\varepsilon n\left(\frac{\sigma}{r}\right)^n - C\varepsilon m\left(\frac{\sigma}{r}\right)^m\right]\frac{1}{r} \tag{4.1}$$

The terms $C\varepsilon n$ and $C\varepsilon m$ are independent of r and remain constant throughout an simulation. They can be precomputed. For that reason the **ParticlePropertyLibrary** is extended with a separate mixing-Data storage for the Mie-potential to store the new values. Consequently, the key computational factor are the two exponentiations. These will be referred to as **mien** for $\left(\frac{\sigma}{r}\right)^n$ and **miem** for $\left(\frac{\sigma}{r}\right)^m$ in the remainder of the thesis.

4.1. Force Calculation of the Mie Potential

4.1.1. Square-root

When calculating the distance r between two particles, a square root must be taken. However, for reasons of efficiency, this should be avoided where possible.This can be done for all straight exponents, because raising a Euclidean distance to its second power is just a dot product: $\left(\frac{\sigma}{r}\right)^2 = \frac{\sigma^2}{(x_1-x_2)^2+(y_1-y_2)^2+(z_1-z_2)^2}$

4.1.2. Naive-Loop

The naive method to implement the exponentiation involves using a simple for-loop. First mien is initialized with one or with the square root of the second power of the fractal $\sqrt{\frac{\sigma^2}{r^2}}$, depending on whether the exponent is odd. Then mien is multiplied $\lfloor \frac{n}{2} \rfloor$ times with $\frac{\sigma^2}{r^2}$. (see Listing 4.1).

```
mie_n = n & 1 ? sqrt(fract) : 1; //check if n is odd; fract = sigmaSquared * invdr2;

for (size_t k = 1; k < n; k += 2) {
    mie_n = mie_n * fract;
    }
```

Listing 4.1: The naive loop approach for calculating the exponentiaton. The implementation using SVE/AVX or auto-vectorization is similar.

4.1.3. Binary Exponentiation

Now binary exponentiation is implemented to compute mien and miem. Like in the naive loop approach mien (miem) is set to one or to the square root of the second power of the fractal $\sqrt{\frac{\sigma^2}{r^2}}$. For each iteration of the loop, the exponents are shifted to process their bits according to the algorithm discussed in section 2.3.1. This is done starting from the second bit of the exponents to avoid unnecessary square root computations.

```

  // check if exponents are odd
  // fract is sigmasquared * invdr2
  mie_n = n & 1 ? sqrt(fract) : 1;
  mie_m = m & 1 ? sqrt(fract) : 1;

  //last bit is not needed
  n = n >> 1;
  m = m >> 1;

  //iterate through bits of exponent
  while(n || m) {

    //if current bit in m is odd, multiply
    if(m_exp&1)
        mie_m = fract * mie_m;

    //if current bit in n is odd, multiply
    if(n_exp&1)
```

```
        mie_n = fract * mie_n;

    //square for every bit in exponent
    fract = fract*fract;

    //shift exponents, to process next bit
    n = n >> 1;
    m = m >> 1;
}
```

Listing 4.2: Binary Exponentiation is used for computing the exponentiatons. Both exponents n and m are being processed in a single loop as the squaring part of the algorithm is equal for both exponents. The implementation using SVE/AVX or auto-vectorization is similar.

4.1.4. Double Addition-Chain

We now want to implement the algorithm introduced in section 2.3.2 that uses combined calculations.

Construction of the Double Addition Chain

The double addition chain only depends on n and m. Therefore it is constant for the duration of an simulation and only needs to be computed once at the beginning. For the bit-vector representation, an unsigned 16-bit integer is selected. Starting with the pair (m, n) we iterate until the pair $(0, 1)$ is reached. In each iteration the inverse rules of the conditional equation 2.10 are applied to n and m. We encode what rule was chosen in our addition chain according to 2.11 and 2.12. We use another integer to point to the current bit of the chain.

```
int chain = 0;
int pointer = 1;
int i = 0;

or(; m || n != 1; i++, pointer <<= 1) {
 if (m <= n / 2) {
   i++;
   if (n & 1) {
     chain |= pointer;
   }
   pointer <<= 1;
   n >>= 1 ;
 } else {
   auto diff = n − m;
```

```
  n = m;
  m = diff;
  chain |= pointer;
}
chain = reverse_chain(chain);
```

Listing 4.3: Constructing the Double Addition Chain.

Implementation of the Exponentiation

When using this method taking a square root is always required. Miem is set to 0, and mien is set to 1. It iteratetively processes the chain by applying the encoded rules.

```
// fract is sigmasquared * invdr2
auto base = sqrt(fract);
miem = 1.0, mien = base; sqrt(fract);
for(size_t k = 0; k < chain_len; k++ ,chain >>= 1){
      if(chain&1){

            //multiplying miem and mien
            auto tmp = miem;
            miem = mien;
            mien = tmp;
            mien = miem * mien;
      }
      else{
            chain>>=1;
            k++;

            //doubling operation
            mien = mien * mien;

            if(chain&1){

            //increment operation
            miem= miem*;
       }
}
```

Listing 4.4: The exponentiation of mien and miem using a double addition-chain. The implementation using SVE AVX or autovectorization is similar.

Compile Time Programming

As discussed in Section 2.5 it can be beneficial to provide the values of the exponents at compile-time. For that three additional functors are offered. In these functors the values of the exponents are hardcoded and stored as **constexpr** values.

5. Results

This chapter analyses the performance of the different functor variants and the algorithms for the exponentiations using the MD-flexible example that is integrated in AutoPas. MD-flexible is a MD simulator enabling the user to configure scenarios and running simulations with the help of input files. Performance test are run on the the CoolMUC-2[1] cluster of the Leibnitz-Rechenzentrum (LRZ) and on the Fujitsu ARM FX700 system[2] located at the Helmudt-Schmidt-Universität Hamburg. CoolMUC2 employs the "Haswell"-based Intel Xeon E5-2690 v3 as processors. Each node features a clock frequency of 2.60 GHz and contains 28 cores with 2 threads per core. The AVX instruction set is supported by the processor. Each node of the Fujitsu ARM FX700 system features a clock frequency of 2.0 GHz and has 48 cores. The SVE instruction set is supported by its used A64FX processor. AutoPas is compiled using gcc/11.2 and the *–ffast-math* flag is activated to help with autovectorization.

For the testing environment a cubic domain of size $51 \times 51 \times 51$ is chosen and the cutoff radius is set to 3. The number of particles within the domain is specified by the number of particles per dimension x. This results in x^3 particles within the domain. The particles are uniformly distributed, which is being enforced by choosing a particle spacing of $\frac{domainsize}{x}$. DeltaT, the time in between two simulation steps, is set to 0. This prevents the particles from moving and is done as our interest lays mainly in the performance of the force-calculations.

5.1. Increasing the n-Exponent

To study the influence of the exponents on the run-time, experiments are conducted for m set to 6 and selected values for n. This is especially interesting as the (n,6) Mie Potential is commonly used in research [27]. 50 Particles were inserted per dimension, which corresponds to a total of 125,000 particles within the domain. The run-time is measured for the values 12,13 and 14 of n. All combinations of functors and algorithms for the exponentiation were tested for a varying number

[1]https://doku.lrz.de/linux-cluster-10745672.html
[2]https://www.hsu-hh.de/hpc/hpc-hardware-systems/

of iterations depending on the speed of the specific functor. The **Linked Cells** container was used with the traversal lc_c08. Figure 5.1 displays the data from CoolMUC2, and Figure 5.2 from HSU.

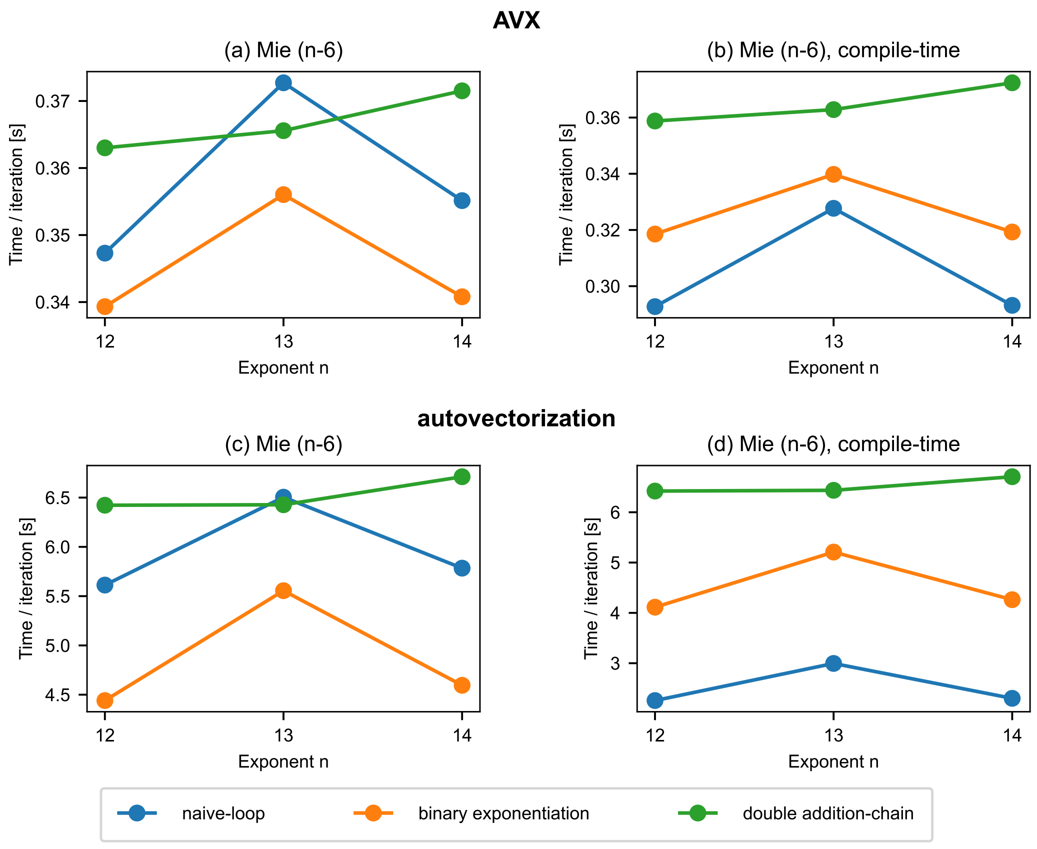

Figure 5.1.: Comparison of the Mie-AVX and Mie-Autovec Functors for increasing n. The exponents of the Mie Potential were once provided at compile-time and once at run-time. The tests were conducted on CoolMUC2 with a single thread.

At first one can note that the AVX and the SVE functors are significantly faster than the functors that use autovectorization with OpenMP. This could be due to difficulties of the compiler vectorizing the code efficiently as the algorithms for the calculation of the powers introduce additional complexity compared to the LJ functors.

When comparing the modes for the power calculation, it can be seen that across all functors the computation time of the naive-loop mode and the binary-

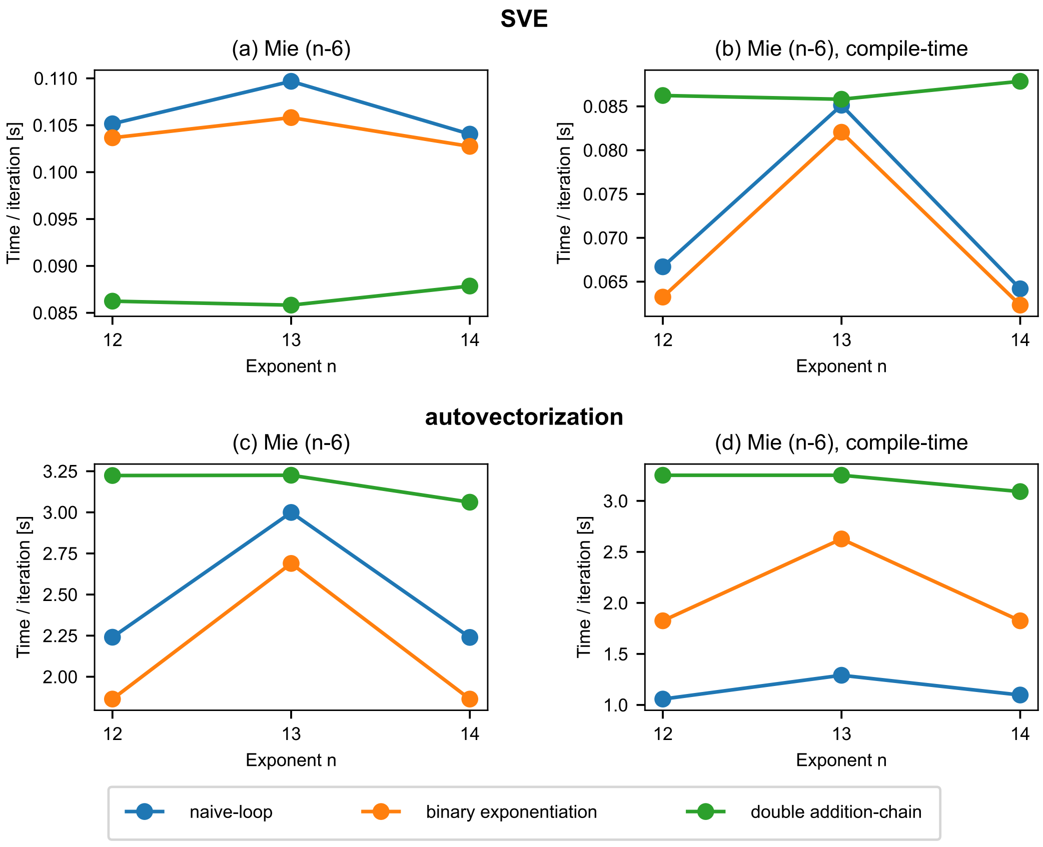

Figure 5.2.: Comparison of the Mie-SVE and Mie-Autovec Functors for increasing n. The exponents of the Mie Potential were once provided at compile-time and once at run-time. The tests were conducted on the Fujitsu ARM FX700 system with 8 threads.

exponentiation mode increase for $n = 13$ before decreasing again at $n = 14$, while the performance of the double-addition chain is more consistent across the exponents. This is explainable by the fact that it is necessary to take a square root for odd exponents, which is a computational expensive instruction as explained in 4.1.1. In contrast the double addition-chain will calculate a square root regardless of the oddness of the exponent.

One can also see, that while providing the exponents at compile time results in a performance boost for the naive-loop and the binary-exponentiation methods, it has no effect on the performance of the double-addition chain algorithm. When providing the exponents at run-time the binary exponentiation algorithm is faster than the naive-loop algorithm. However, when the exponents are known at compile-time this changes. A possible explanation could lie in the simplicity of the naive-loop approach that enables the compiler to make more optimizations compared to the binary exponentiation algorithm, which is more complicated. However, this does not hold true for the SVE functor where binary exponentiation experiences a bigger speed up when providing the exponents at compile-time.

The double addition-chain method depends strongly on the double addition-chain, which so far can only be constructed and provided at run-time. Therefore the compiler will have difficulties optimizing the algorithm to the fullest.

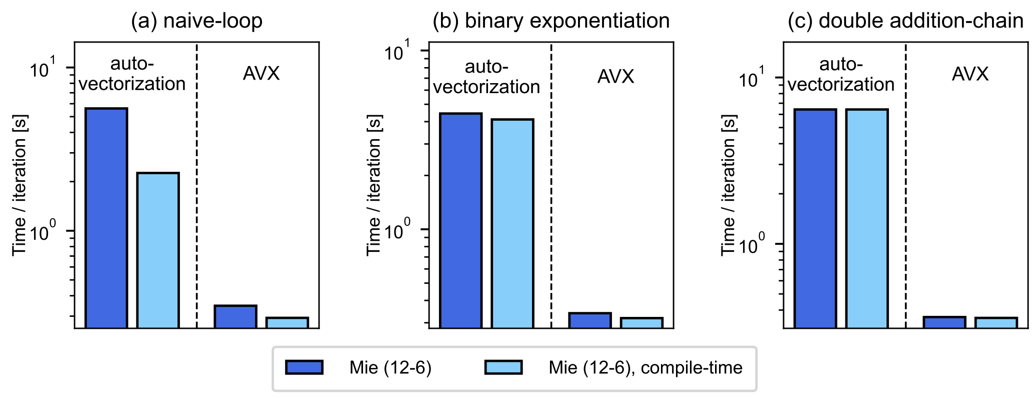

Figure 5.3.: Effect of providing the exponents at compile-time for the case n=12 on the exponentiation algorithms on SuperMUC2.

Figure 5.3 gives a better visualisation of the individual speedups of the different exponentiation modes when the exponents are provided at compile time with the data from CoolMUC2 and n fixed at 12. The corresponding figure for the ARM system can be found in the Appendix.

5.2. Comparison of LJ and Mie(12,6)

As previously discussed the Mie potential corresponds to the Lennard-Jones potential when the exponents are set to 12 and 6. This allows for a direct comparison of the Lennard-Jones functors with the implemented Mie functors. The exponents will be provided at compile-time and as a exponentiation method the naive-loop approach is chosen, as this was in all cases except of the SVE functor the best performing method in the previous experiment for n=12. Figure 5.4 compares the performance of the different functors on CoolMUC2.

All LJ functors perform better than their corresponding Mie equivalents. This was expected as the Lennard-Jones functors allow for a hardcoded computation of the force-computation due to the fixed exponents. The run-time is 20% higher in the case of autovectorization, and 4% higher in the case of AVX when using the Mie functors with compile-time exponents.

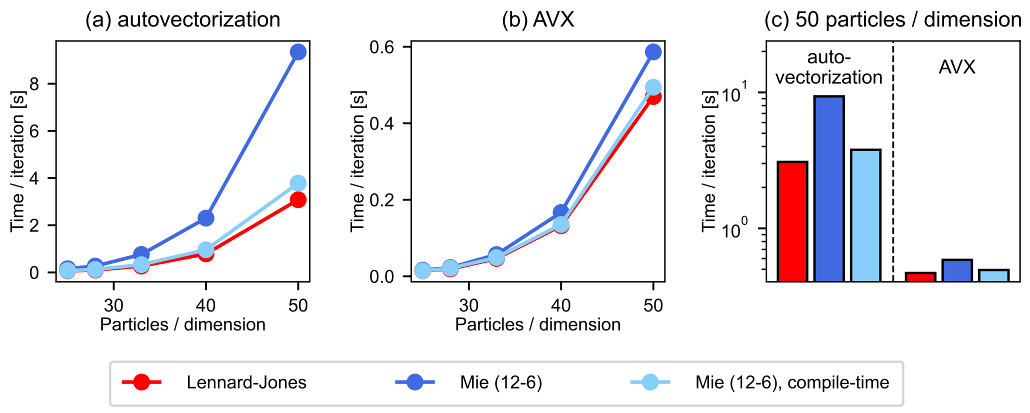

Figure 5.4.: Comparison of the AVX and the Autovectorization Functors of the LJ and Mie Potential. In the Mie-Potential the exponents are once provided at compile-time and once at run-time. The tests were conducted on CoolMUC2 with a single thread

5.3. Comparison of Different Particle Containers

Also interesting is the performance of the Mie potential when using different particle containers in AutoPas. For that the run-time of the particle containers Linked Cells, Verlet Lists and Verlet Cluster Lists are compared. Verlet Lists is tested in combination with the vl_list_iteration traversal and Verlet Cluster Lists is tested using the vcl_clusterIteration traversal. The experimental setup is the same to the prior *increasing n scenario*. The results for the ARM system are displayed in Figure 5.5.

The LinkedCell container performs far better than the Verlet Cluster Lists and the Verlet List containers. A possible explanation could be the better vectorization properties as discussed in section 3.1. Verlet Lists perform better than the Verlet Cluster Lists. However the opposite holds true for the CoolMUC2 cluster as apparent in Figure 5.6. This could have multiple reasons: Firstly, it could be due to the different architectures of the systems. Further it is also possible that the Verlet Lists scales better with the number of threads used. The tests on the ARM cluster were run with 8 threads, while only 1 thread was used on CoolMUC2. Therefore running the test again with multiple threads could give better insights. Unexpected behaviour occurs for Verlet Lists when using the AVX functor with compile-time exponents. The performance increases for the odd exponent of 13. This stands in contrast to previous observations and reasoning. Further experiments, e.g. with more iterations and/or more simulated particles, are necessary to check the reproducibility of this behaviour and identify underlying reasons.

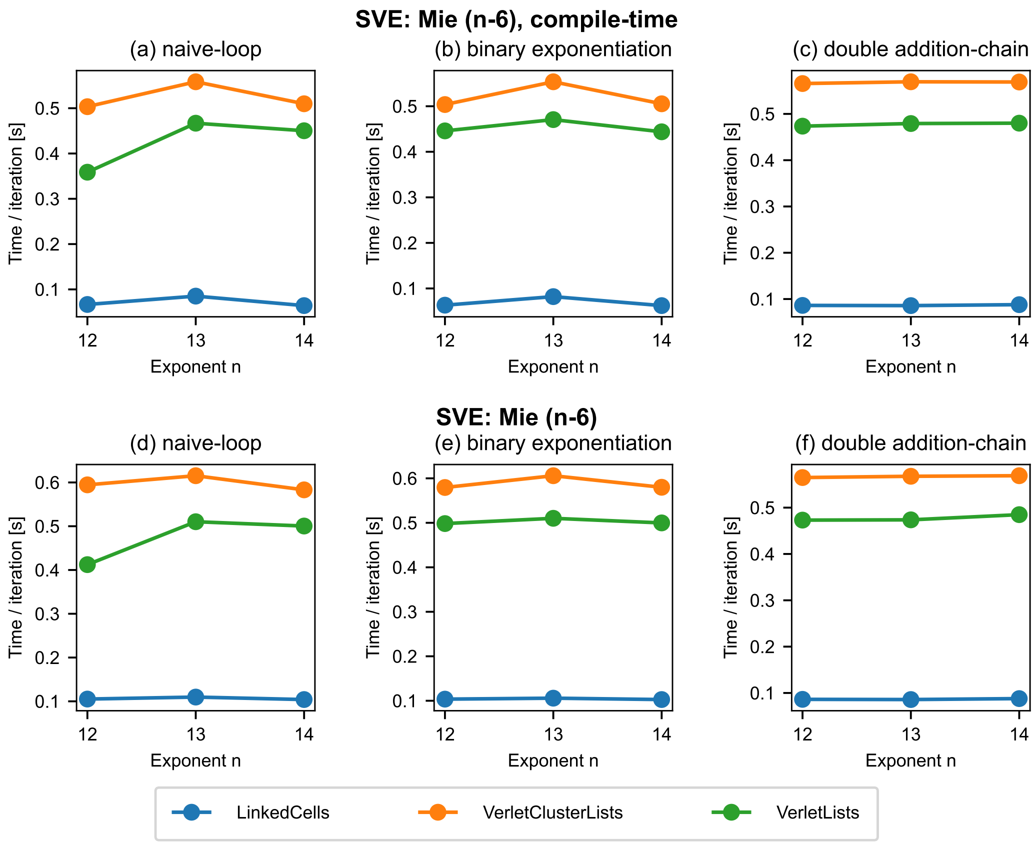

Figure 5.5.: Comparison of different Particle-Containers. The Mie-SVE Functor is used with exponents provided at compile and runtime. The tests were conducted on the Fujitsu ARM FX700 system with 8 threads

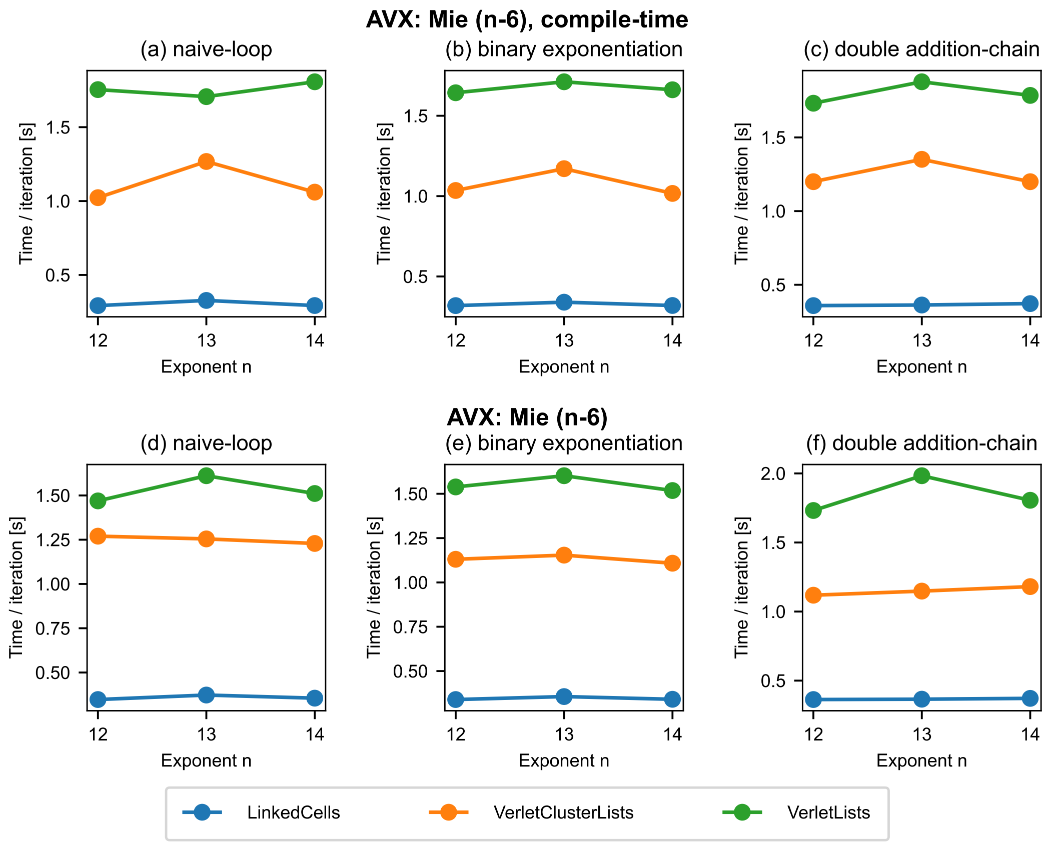

Figure 5.6.: Comparison of different Particle-Containers. The Mie-AVX Functor is used with exponents provided at compile and run-time. The tests were conducted on CoolMUC2 with a single thread.

6. Future work

This thesis presents an implementation for force calculations based on the Mie potential in AutoPas. Some parts of the code might benefit from further analysis and the application of vector intrinsics specific optimizations. Furthermore correctness testing for the SVE functors remains an area for future research. However, since these SVE functors follow the algorithms of the previously implemented functors and share a similar structural framework, it is reasonable to infer that the results presented here are accurate, although subject to future validation.

The exponents of the Mie potential should be parsed at compile-time by employing meta-programming to increase usability. It would also be interesting to implement the construction of the double addition-chain at compile-time, and test if this leads to any increase in performance. Regarding the exponentiations, no algorithm performed best in all tested scenarios. Further tests with an wider range of exponents are required to gain more insights. While this was initially planned in the thesis, problems on the SuperMUC2 cluster significantly slowed down the execution of experiments. However, in the tested scenarios a simple for loop was a good method to handle the variable exponents. For smaller exponents it is suspected that the performance of the naive-loop will perform better relative to binary exponentiation and the double addition chain, while binary exponentiation and the double addition-chain approach will increase in performance relative to naive-loop for larger exponents. If the results support this thesis, it would be interesting to implement an mechanic to select the exponentiation method based on the values of the exponent.

Furthermore this thesis focused mainly on the force calculations. The calculation of additional physical properties like the virial and potential energy, can easily be enabled by adapting the current code slightly.

Until now the Mie potential was implemented in AutoPas using AVX and SVE intrinsics. This limits its applicability to architectures that support these specific SIMD instruction sets. However there are other instruction sets besides AVX and SVE. Maintaining a functor for each instruction set is complex and time consuming. A solution could be the usage of SIMD-wrappers, that offer an additional abstraction layer in between the code and the instructions. Examples for SIMD-wrappers are xsimd [1] and google-highway [2].

[1]https://github.com/xtensor-stack/xsimd

[2]https://github.com/google/highway

7. Conclusion

The Mie(n,m) potential is a generalized, form of the Lennard-Jones potential with its variable exponents providing additional flexibility for attractive and repulsive forces. It thus allows accommodating more complex interactions, making it more suitable for specialized cases such as the modelling of the properties of nobel gases.

This thesis focused on the implementation of force calculations based on the Mie potential into AutoPas, a particle simulation library. For this, three different implementations using AVX intrinsics, SVE intrinsics and autovectorization were developed. In all of these implementations, three different methods, introduced to handle the variable exponents of the Mie potential, were tested. Exponentiation was realized using a simple for-loop (naive loop), using binary exponentiation and using combined calculations for both exponents enabled by processing a double addition chain.

It was shown that providing the exponents at compile-time rather than at run-time can significantly accelerate the simulation. For compile-time exponents, the naive-loop approach performed best for the tested values of the *n* exponent, except for SVE intrinsics, where binary exponentiation performed best. The combined calculation approach was not convincing in the scenarios tested.

Furthermore, the implementations using AVX and SVE intrinsics outperformed autovectorization approaches. Several particle containers were tested, with Linked Cells being selected as the best-performing one. When using Linked Cells and providing the exponents at compile-time, force calculations based on the Mie(12,6) potential were able to compete with force calculations based on the Lennard-Jones potential, having only 20% longer run-times with autovectorization and only 4% longer run-times with AVX intrinsics in the tested scenario.

In future, it would be interesting to explore the performance of the Mie potential with a wider range of exponents, in order to gain deeper insights into the run-time behaviour of the different algorithms for the exponentiation. The collected results indicated that the Mie potential can be an interesting alternative to the Lennard-Jones potential in AutoPas.

A. Additional Figures

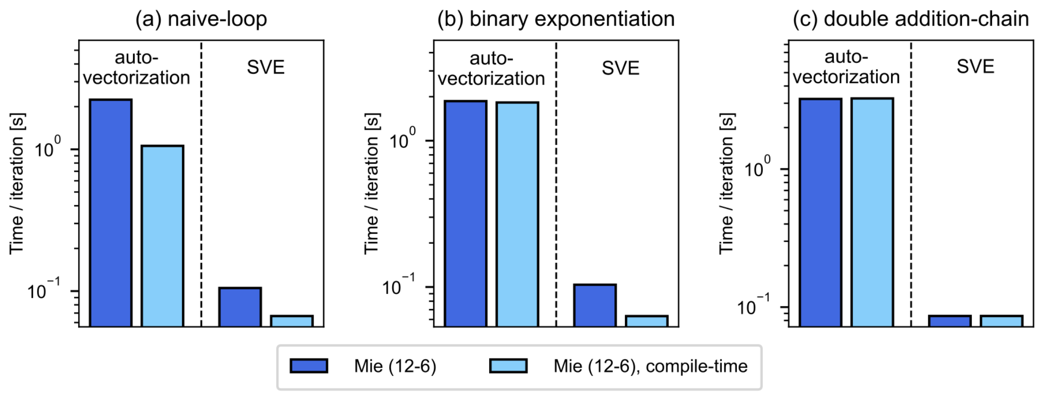

Figure A.1.: Effect of providing the exponents at compile-time for the case n=12 on the exponentiation algorithms on the Fujitsu ARM FX700 system.

List of Figures

Bibliography

[1] S. A. Hollingsworth and R. O. Dror. "Molecular dynamics simulation for all." en. In: *Neuron* 99.6 (Sept. 2018), pp. 1129–1143.

[2] B. J. Cowen and M. S. El-Genk. "On force fields for molecular dynamics simulations of crystalline silica." In: *Computational Materials Science* 107 (2015), pp. 88–101. ISSN: 0927-0256. DOI: `https://doi.org/10.1016/j.commatsci.2015.05.018`.

[3] R. Shukla and T. Tripathi. "Molecular Dynamics Simulation in Drug Discovery: Opportunities and Challenges." In: *Innovations and Implementations of Computer Aided Drug Discovery Strategies in Rational Drug Design*. Ed. by S. K. Singh. Singapore: Springer Singapore, 2021, pp. 295–316. ISBN: 978-981-15-8936-2. DOI: `10.1007/978-981-15-8936-2_12`.

[4] 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.

[5] A. Aasen, M. Hammer, Å. Ervik, E. A. Müller, and Ø. Wilhelmsen. "Equation of state and force fields for Feynman-Hibbs-corrected Mie fluids. I. Application to pure helium, neon, hydrogen, and deuterium." In: 151.6, 064508 (Aug. 2019), p. 064508. DOI: `10.1063/1.5111364`.

[6] S. Dufal, T. Lafitte, A. Galindo, G. Jackson, and A. J. Haslam. "Developing intermolecular-potential models for use with the SAFT-VR Mie equation of state." In: *AIChE Journal* 61.9 (2015), pp. 2891–2912. DOI: `https://doi.org/10.1002/aic.14808`. eprint: `https://aiche.onlinelibrary.wiley.com/doi/pdf/10.1002/aic.14808`.

[7] K. R. Hadley and C. McCabe. "Coarse-grained molecular models of water: a review." In: *Molecular Simulation* 38.8-9 (2012). PMID: 22904601, pp. 671–681. DOI: `10.1080/08927022.2012.671942`. eprint: `https://doi.org/10.1080/08927022.2012.671942`.

[8] G. Mie. “Zur kinetischen Theorie der einatomigen Körper.” In: *Annalen der Physik* 316.8 (1903), pp. 657–697. DOI: `https://doi.org/10.1002/andp.19033160802`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/andp.19033160802`.

[9] H. Watanabe and K. M. Nakagawa. “SIMD vectorization for the Lennard-Jones potential with AVX2 and AVX-512 instructions.” In: *Computer Physics Communications* 237 (Apr. 2019), pp. 1–7. ISSN: 0010-4655. DOI: `10.1016/j.cpc.2018.10.028`.

[10] F. A. Gratl, S. Seckler, N. Tchipev, H.-J. Bungartz, and P. Neumann. “AutoPas: Auto-Tuning for Particle Simulations.” en. In: *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, May 2019. ISBN: 9781728135106. DOI: `10.1109/ipdpsw.2019.00125`.

[11] M. Griebel, S. Knapek, and G. Zumbusch. *Numerical Simulation in Molecular Dynamics: Numerics, Algorithms, Parallelization, Applications*. 1st. Springer Publishing Company, Incorporated, 2007. ISBN: 3540680942.

[12] H. A. Lorentz. “Ueber die Anwendung des Satzes vom Virial in der kinetischen Theorie der Gase.” In: *Annalen der physik* 248.1 (1881), pp. 127–136.

[13] D. Berthelot. “r. hebd. Seanc. Acad Sci.” In: *Paris* 126 (1898), p. 1703.

[14] A. Stone. *The theory of intermolecular forces*. oUP oxford, 2013, p. 203.

[15] R. A. Messerly, M. C. Anderson, S. M. Razavi, and J. R. Elliott. “Improvements and limitations of Mie λ-6 potential for prediction of saturated and compressed liquid viscosity.” In: *Fluid Phase Equilibria* 483 (2019), pp. 101–115.

[16] R. J. Sadus. “Second virial coefficient properties of the n-m Lennard-Jones/Mie potential.” In: *The Journal of Chemical Physics* 149.7 (Aug. 2018), p. 074504. ISSN: 0021-9606. DOI: `10.1063/1.5041320`. eprint: `https://pubs.aip.org/aip/jcp/article-pdf/doi/10.1063/1.5041320/10979114/074504\_1\_online.pdf`.

[17] B. L. Shanks, J. J. Potoff, and M. P. Hoepfner. “Transferable Force Fields from Experimental Scattering Data with Machine Learning Assisted Structure Refinement.” In: *The Journal of Physical Chemistry Letters* 13.49 (2022), pp. 11512–11520.

[18] J. J. Potoff and D. A. Bernard-Brunel. “Mie potentials for phase equilibria calculations: Application to alkanes and perfluoroalkanes.” In: *The Journal of Physical Chemistry B* 113.44 (2009), pp. 14725–14731.

[19] J. J. Potoff and G. Kamath. "Mie potentials for phase equilibria: Application to alkenes." In: *Journal of Chemical & Engineering Data* 59.10 (2014), pp. 3144–3150.

[20] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Third. Section 4.6.3. USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN: 0201896842.

[21] P. Downey, B. Leong, and R. Sethi. "Computing Sequences with Addition Chains." In: *SIAM Journal on Computing* 10.3 (1981), pp. 638–646. DOI: `10.1137/0210047`. eprint: `https://doi.org/10.1137/0210047`.

[22] M. Rivain. "Securing RSA against Fault Analysis by Double Addition Chain Exponentiation." In: vol. 5473. Apr. 2009, pp. 459–480. ISBN: 978-3-642-00861-0. DOI: `10.1007/978-3-642-00862-7_31`.

[23] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker. "The ARM Scalable Vector Extension." In: *IEEE Micro* 37.2 (Mar. 2017), pp. 26–39. ISSN: 0272-1732. DOI: `10.1109/mm.2017.35`.

[24] T. Veldhuizen. "C++ Templates are Turing Complete." In: (July 2003).

[25] 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." en. In: *Computer Physics Communications* 273 (2021), p. 108262. DOI: `10.1016/j.cpc.2021.108262`.

[26] S. Páll and B. Hess. "A flexible algorithm for calculating pair interactions on SIMD architectures." In: *Computer Physics Communications* 184.12 (Dec. 2013), pp. 2641–2650. ISSN: 0010-4655. DOI: `10.1016/j.cpc.2013.06.003`.

[27] A. Ahmed and R. J. Sadus. "Solid-liquid equilibria and triple points of n-6 Lennard-Jones fluids." In: *The Journal of Chemical Physics* 131.17 (Nov. 2009), p. 174504. ISSN: 0021-9606. DOI: `10.1063/1.3253686`. eprint: `https://pubs.aip.org/aip/jcp/article-pdf/doi/10.1063/1.3253686/15912719/174504\_1\_online.pdf`.