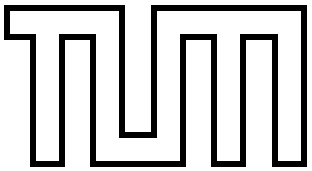


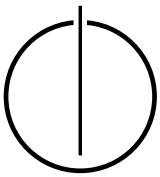
FAKULTÄT FÜR INFORMATIK

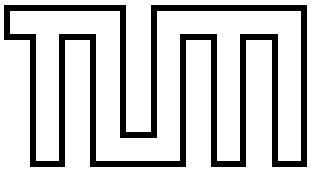
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Comparison of performance-portable Vectorization Tools for the Lennard-Jones Force Calculation

Robin Quedzuweit





FAKULTÄT FÜR INFORMATIK

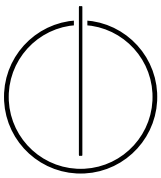
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Comparison of performance-portable Vectorization Tools for the Lennard-Jones Force Calculation

Vergleich von performanz-portablen Vektorizierungstools für die Kalkulation der Lennard-Jones Kraft

Author:	Robin Quedzuweit
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Samuel James Newcome, M. Sc.
	Markus Mühlhäußer, M. Sc.
Date:	16.08.2023



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

Munich, 16.08.2023 Robin Quedzuweit

Abstract

Vectorization holds a significant role within the domain of molecular dynamics. While compiler auto-vectorization tools often fall short in vectorizing complex computational scenarios, and manually coding multiple SIMD intrinsics can be labor-intensive when striving to ensure implementation performance across a wide spectrum of SIMD extension sets, SIMD wrappers aim to simplify this process by introducing an abstraction layer for crafting vectorized code.

In this thesis, we will compare three distinct SIMD wrappers, *xsimd*, *MIPP* and *SIMDe*, implementing them into Autopas to vectorize the Lennard-Jones force calculation. Afterwards, we evaluate the performance of these implementations on two different systems - one supporting AVX and the other NEON and SVE intrinsics. Our findings indicate that SIMD-Wrappers present a viable method to achieve performance portablity in any vectorizable implementation, even in the demanding field of molecular dynamics. However, it's important to be mindful of certain limitation of the wrappers.

Contents

Part I.

Introduction and Background

1. Introduction

Vectorization is a powerful method in the field of parallel programming. Presently, the so-called **Single instruction, multiple data (SIMD)** operations are especially significant in multimedia applications, such as audio and image processing, while also assuming an important role in High-Performance Computing (HPC) applications. They offer several advantages, most notably the reduction of runtime.
Over the years, various architectures have been developed by different manufacturers, each supporting different SIMD instruction sets. For example, Intel has introduced AVX, AVX2, AVX-512, SSE and ARM-Architectures employ SVE and NEON intrinsics. This diversity poses a challenge when aiming for program portability across different types of architectures to benefit from the advantages of vectorization. Frequently, the complexity of the program surpasses the capabilities of auto-vectorization tools, requiring separate vector-intrinsics functions to be manually implemented for each instruction set. As a result, programming and maintaining such code can become a tedious task. The solution to this problem resides in **SIMD-Wrappers** which introduce an abstraction layer between the codebase and the specific vector intrinsics.
Several performance-portable SIMD-Wrappers are available, including xsimd, MIPP, SIMD Everywhere or E.V.E. Each of these wrappers supports multiple SIMD architectures, bringing SIMD-instructions to a performance-portable level.

In this thesis, we delve deeper into a range of SIMD-Wrappers and compare their performance within an HPC environment. We will examine whether the advantages and disadvantages of SIMD-Wrappers make them a viable alternative to conventional SIMD-intrinsics. By implementing the most suitable wrappers into the molecular simulation library Autopas, we can assess if they are applicable in a computationally demanding field such as molecular simulation. Finally, the implementations will be tested on two supercomputers with support for different vector extension sets to verify the portability of the SIMD-Wrappers.

2. Background

2.1. Autopas

Autopas[1] is a short-range particle simulation library. It is designed as a black box to allow the user to simulate problems and scenarios with minimal responsibility for achieving the best possible results in performance.[GSBN21] It implements several different optimization and parallelization strategies with OpenMP and algorithms, that Autopas is then able to automatically choose from at runtime. This way it can tune the simulation to the fastest and most optimal way to simulate any scenario the user has provided.[GST+19]

2.1.1. Molecular Simulation

For the force calculation between molecules, Autopas implements the Lennard-Jones potential, since it is a frequently used potential in the field of particle simulation studies.[GSBN21] Figure 2.1 illustrates the function for the potential with the distance r_{ij} between two particles i and j and the Lennard-Jones parameters σ and ϵ. These parameters describe the behavior of the interaction between the particles.

$$U_{ij} = 4\epsilon \left(\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right) \tag{2.1}$$

When simulating particles within a domain, one approach is to compute the interactions of every pair of particle in the entire simulation. While this might be reasonable for certain small-scale scenarios, when dealing with a larger number of particles, it can rapidly lead to scalability issues. As the forces between all particle pairs need to be calculated, the computational complexity grows to $O(N^2)$, with N being the number of particles in the simulation.
However, in most pair potentials, like the Lennard-Jones potential, the force between two particles tends to approach zero as the distance between them becomes increasingly larger. For these short-range potentials, it is unnecessary to calculate every particle interaction within simulation domain, as greater distances result in force values that can be neglected without having a noticeable impact on the final force of the particles.[MG07] This being the case, a cutoff radius r_c can be introduced to specify the maximal distance between a particle pair, for which the forces should be evaluated.

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

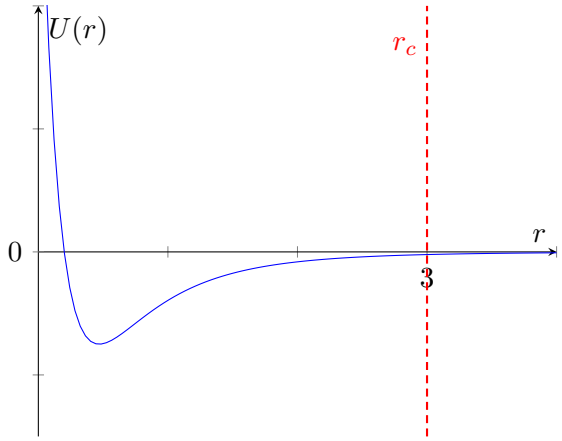


Figure 2.1.: Lennard-Jones potential

In figure 2.1, we observe the Lennard-Jones potential gradually approaching zero with progressively larger distances r. To reduce the runtime complexity of the force calculation, we can set the cutoff radius as demonstrated in the graph to $r_c = 3$ and exclude any force calculations between particle pairs with distances exceeding this threshold.
As a result of this improvement, the force calculation now scales with $O(N)$, which offers improved manageability in terms of scalability.

2.1.2. Particle Data Structures

There are two different ways Autopas can represent particles. An **Array-of-Structures**, or short **AoS**, refers to storing the particles as a class object that has their properties stored as member variables. The main advantage of AoS is the ability to access the particle through random access, but is not particularly suitable for vectorization.[GSBN21]
Therefore, Autopas introduces a **Structure-of-Arrays**. We now have multiple arrays that each holds a property of the particles. The values of force, velocity, or position are now located in their own array, or in this case, a $std :: vector$, and are accessible with the index of the particle. This means we have excellent memory accessibility, ideal for vectorization, since loading values occurs seamlessly in a single operation.[GSBN21]

Autopas offers a variety of algorithms for managing the particle interactions in the simulation. These algorithms are responsible for identifying the correct particles that are relevant for force calculating. The selection of these methods can have a substantial impact on the simulation's efficiency. Three such algorithm present in Autopas are described here:

1) **Direct Sum:**
   The Direct Sum algorithm is the simplest way of handling particle interactions. All particles in the simulation are stored within a single data structure, as seen in figure 2.2. Therefore, distances must be calculated for all possible particle pairs to determine whether they are within the cutoff radius and should be considered in the force calculation. Consequently, the computational complexity of the distance calculation in a simulation containing N particles is $O(N^2)$.
   This approach is only feasible for small-scale particle simulations, as other methods

can have certain drawbacks in such scenarios, like additional overhead of the algorithm itself and/or memory usage. [GSBN21]

2) **Linked Cells:**
To solve the computational complexity problem of the Direct Sum method, the Linked Cells Algorithm divides the simulation domain in multiple smaller cells (as illustrated in figure 2.3), that each are able to manage the particles within their sub-domain. The size of the cells are typically set to be equal to or greater than the selected cut-off radius, since therefore we only have to calculate the distances between particles in the cell itself and those in directly adjacent neighbour cells. This greatly reduces the number of distance checks between particles during simulation and therefore results in an algorithm with a runtime complexity of $O(N)$. [MG07]

However, with this method, there are still numerous unnecessary distance checks, since the cut-off sphere is poorly represented by the cell structure.
As demonstrated in [GST+19], when comparing the 3D volumes of the cutoff sphere and the neighboring cells, there is only an approximately 16% overlap, if the cell size is equal to the cutoff radius r_c:

$$\frac{V_{cutoff}}{V_{neighbours}} = \frac{\frac{4}{3}\pi {r_c}^3}{(3r_c)^3} \approx 0.155 \tag{2.2}$$

This phenomenon can be observed in the figure 2.3, since we still have to compute the distances to all particles within the blue area, even though some of them are well beyond the cut-off radius.
Another drawback is that we still have to make sure to accurately transfer the particles in the correct cells if they cross cell borders, leading to additional computational overhead.[GSBN21]
The advantage to this method is the relativly modest memory overhead of the cell structure itself, along with better memory access capabilities compared to other available alternatives. This allows the algorithm to excel in vectorization and similar parallel optimizations. [Fom11]

3) **Verlet Lists:**
To further reduce the number of unnecessary distance calculations, we can introduce Verlet Lists [Ver67]. We assign each particle an own list of neighbouring particles, which are tracked within a radius r_v. In most simulation scenarios, it is not essential to update the list every cycle. Therefore, we can introduce an update frequency, so that the calculations of distances occur only every n-th iteration. Although these calculations still have a computational complexity of $O(N^2)$, the decreased number of checks represents a significant advancement compared to the Direct Sum method. We can enhance this further by combining the Verlet Lists with the Linked Cell algorithm so we only need to calculate the particle distances in the cell and its neighbours, instead of the entire simulation domain.[YWLC04]
For the force calculation, we only consider the particles present in this list for the current particle. Since we simulate a dynamic environment with moving particles, this radius should be set greater than the cutoff radius r_c, since in between the list updates

a particle could outerwise enter the radius unnoticed by the particle, thus resulting in incorrect results. Often it is possible to determine a good value with the average velocity in the system. [Ver67] This area between the cutoff radius and r_v is also called verlet-skin and, in figure 2.4, is displayed as the red dotted circle. All particles within this area are saved in the neighbour list of the center particle.
If the size of the verlet-skin is poorly chosen, updating the verlet list in fixed intervals could lead to incorrect results, as fast moving particle might enter the cutoff radius prior to the list update. Some variations of the Verlet Lists algorithm propose methods to automatically determine, if updating the lists is required, which can further reduce the number of unnecessary distance checks. [CD90] For example, one possibility is keeping track of the relative distance traveled of each particle between time steps, to decide if updating a neighbour list is necessary.[FR81]
The big drawback of this algorithm is the increased complexity and the large overhead in memory, since for every particle a own list has to be maintained.

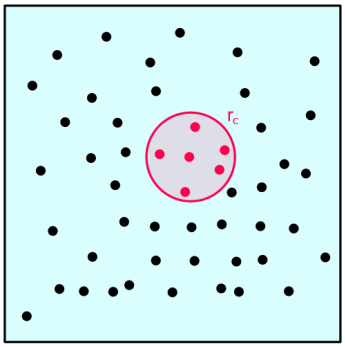


Figure 2.2.: Direct Sum

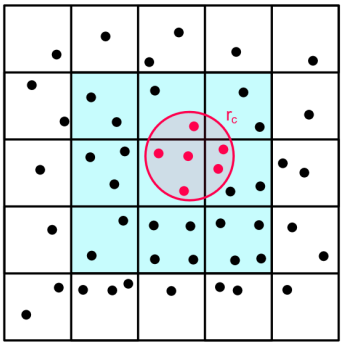


Figure 2.3.: Linked Cells

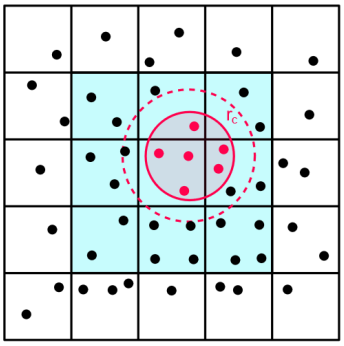


Figure 2.4.: Verlet Lists

The above figure depicts the different algorithms for interaction of particles. For all particles located in the blue area we have to calculate the distance to the current particle (exception being the Verlet List where only the distances to the particles in the red dotted radius have to be calculated everytime). All red particles have their forces calulated, as they are located within the cutoff radius r_c (red circle).

2.2. Single Instruction, Multiple Data

SIMD (Single Instruction, Multiple Data) is a set of instructions that allows CPUs to execute an operation on multiple data values at the same time. SIMD-supporting processors have special SIMD registers available, that are larger than the usual registers. The size can vary depending on the architecture of the system:

Instruction Set Extension	Register Size
SSE	128 bit
NEON	128 bit
AVX/AV2	256 bit
AVX512	512 bit
SVE	up to 2048 bit

With the register being this large, it allows to load more than one value into a register and with the vector intructions (or: SIMD intrinsics) provided by the instruction set extensions, it is possible to execute common operations, like addition, subtraction, multiplication etc. on the multiple stored values in the register simultaneously.
As an example, the NEON extension for ARM is able to use its 128 bit size to load up to four 32 bit single-precision floating point numbers or two 64 bit double-precision floating point numbers into one register.
In figure 2.5, we can examine how an addition between two SIMD registers would function. We have two registers a and b in which the values $a1 - a4$ and $b1 - b4$ respectively are loaded into. The provided addition-operation adds up both registers to the register c, so that $c\{i\} = a\{i\} + b\{i\}$ for all $i \in [1, 2, 3, 4]$ as illustrated by the arrows.

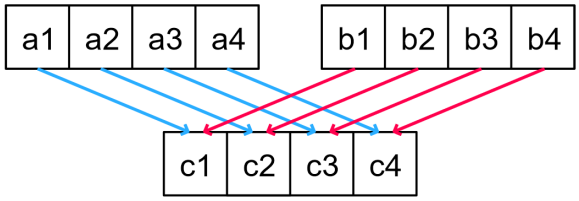


Figure 2.5.: SIMD register add operation

SIMD can be a powerful tool for increasing performance in various situation. Molecular simulations especially, are highly suitable for vectorization, since we have multiple data, the particles, for which the same instruction, the force calculation, has to be performed. So, when calculating all the interactions for one particle, we do not have to do it with one particle after the other. With SIMD we are able to load the variables from multiple particles in a SIMD-Register at once and do the force calculation simultaneously, resulting in substantial speed-up in the simulation.

2.3. SIMD-Wrappers

SIMD-Wrappers are intended as an intermediate stage between SIMD-intrinsics and auto-vectorization tools. Various compilers implement some form of auto-vectorizations. They especially struggle to detect code-fragments, when the code's potential vectorizable parts do not show clear patterns, that the code analysis of the compilers heavily rely on [EGF+12]. Other limitations, like data dependecies between different loops, nested loops, other certain function calls in the code that should be vectorized, can lead to having a lot of sections of the code, that could be vectorized, to not be compiled with SIMD instructions.[Intb]
Therefore, to achieve optimal results when trying to capitalize on the advantages of vectorization, it is a good approach to implement the code using SIMD Intrinsics. Especially when the target machine for the program is already known, the most effective method is to implement the desired code utilizing the vectorization extension set, that the machine supports. This ensures the best possible performance of the vectorization for the given hardware.

One drawback of this approach is its heavely reliance on the specific computer architecture that was used when developing the implementation. If you implement the code with AVX-

instructions, it is only possible to run this code on computers with processors that have the support for the AVX-Extension. Consequently, for each targeted systems, the code sections requiring vectorization must be re-written and adapted to the particular architecture specifications.
Another disadvantage is the formulation of the SIMD intrinsics themselves. Like for example, in AVX-instruction, the add-operation for two 256 bit registers is:

```
__m256d c = _mm256_add_pd(a, b);
```

Evidentally, this can significantly compromise the readability of the code, especially in large projects.
Additionally, when implementing code with SIMD intrinsics, it is required to have a deeper understanding of the concept of vectorization and how the registers work for the specific architecture. This sometimes can be a challenging and time-consuming task in comparison to just letting the compiler automatically vectorize the code for you.

Introducing a solution to some of these issues, we can utilize performance-portable SIMD wrappers. They try to provide an efficient way to make SIMD instruction portable across different architectures without much more effort. By now, there are a lot of different SIMD-Wrappers available to the public. Projects like *xsimd*, *MIPP*, *SIMDe*, *E.V.E* all bring different concepts for making SIMD instructions performance-portable. Some wrappers provide a higher level of abstraction for the underlying intrinsics while others try to keep as much control as possible in the hand of the programmer, to facilitate implementing SIMD optimizations into various projects.
In this case, the abstraction also allows the wrappers to improve the code readability. For example, the *MIPP* library even provides operator overloads for its register abstraction, resulting in much cleaner looking code. The same add-operation from before can be implement like this:

```
Reg<double> c = a + b; //or: add(a + b)
```

2.4. Related Works

There are not many papers that compare different SIMD Wrappers to eachother. The release paper for the SIMD-Wrapper MyIntrinsics++ (MIPP) does compare its implementation to other available wrappers, like VC, Boost.SIMD, T-SIMD, xsimd etc. At the time of the paper, xsimd only had support for Intel processors, so it was not tested in ARM environments. The paper concludes that most of the tested wrappers perform similarly to the intrinsics, including the *MIPP* library itself.[CAB+18]

GROMACS[2] is an open-source software package for molecular simulations. Its goal is to provide best perfomances on all systems with a collection of algorithms an parallelization strategies, like GPU acceleration, multithreading and utilizing SIMD registers. *GROMACS* goal is to be able to run on any system available. Therefore, when optimizing the code with vectorization, to make the code portable across different systems. they implemented an abstraction for SIMD that works independently from the underlying SIMD-architecture properties, like register size. Additionally they extended this module with a wide ranging SIMD math library.

[2] https://github.com/gromacs/gromacs

Part II.

Comparison

In this chapter, we will look into a variety of different SIMD-Wrappers that are considered for implementation into Autopas.

3. xsimd

The library *xsimd*[1] is a header-only wrapper for SIMD intrinsics. It uses a register abstraction, $xsimd :: batch < T >$, with T as the value type. Optionally, if you are aware of the intended architecture the code will run on, it is possible to specify the target architecture with a second template paramter: $xsimd :: batch < T, Arch >$. However, if the specified set extension is not supported on the machine, the code will not compile. When the second parameter is left empty, *xsimd* will automatically attempt to choose the best SIMD extension for the system.
By including the architecture, it allows better control when dealing with different architectures, for example, it becomes easier to implement specific optimizations only suited for specifc SIMD extensions.
xsimd achieves its portabitity with the template based approach. The compiler is therefore able to decide at compile-time which parts of the library are needed to compile and replace, based on the Architecture template parameter. This also allows for compiling multiple versions for different set extension with function dispatching, that enables architecture detection even during runtime, with the function in the listing below:

```
template<class ArchList = supported_architectures, class F>
inline detail::dispatcher<F, ArchList> dispatch(F &&f)
```

This provides a way for compiling the functor F as a dispatched function for all architectures specified in the ArchList template parameter. Subsequently, when calling the dispatched functor during runtime, it chooses the fitting implementation with the available information of the architecture. [JM]
The library supports the following instruction set extensions:

- SSE(2, 3, 4.1, 4.2)
- AVX, AVX2
- FMA
- AVX512
- NEON
- SVE (fixed register sizes: 128, 256, 512bit)

Next to the regular batches, which are able to hold scalar values, like integer and floating-point numbers, *xsimd* also implements a specialized batch for boolean values: $xsimd :: batch\_bool < T >$, that are used for masks and comparison-operators. Additionally, the library has the ability to handle complex numbers by specifying a batch with the $std :: complex < T >$ data type as template parameter like in this example: [JM]

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

```
xsimd::batch<std::complex<double>> complex\_batch;
```

xsimd is a capable wrapper designed to make portable SIMD code across a wide range of supported architectures. It facilitates this process by providing a high-level abstraction to the programmer, that can, for instance, improve the readability of the code, while still leaving enough control in order to maximize the performance with SIMD.

4. MyIntrinsics++ (MIPP)

MIPP, similar to *xsimd*, is a header-only library and provides a layer of abstraction for writing performance-portable SIMD code. Like *xsimd*, it utilizes the C++ template functionality to replace the abstract functions provided by *MIPP* with inline SIMD instruction for the current architecture at compile time[CAB+18].
The supported architectures are as followed:

- SSE
- AVX, AVX2
- FMA
- AVX512
- KNCI
- NEON

MIPP provides two levels of abstraction available to the programmer to choose from:

1) **Low Level Interface:**
   This level of abstraction is pointed at the most flexibility when programming, while still providing a performance portable interface. As an abstraction to registers, *MIPP* offers the $mipp :: reg$ datatype, as well as $mipp :: msk$ for conditional masks. This is especially useful on certain architectures, like the ones that support AVX-512, since then *MIPP* can take advantage of the specialized hardware mask registers. Otherwise, the library replaces it with standard registers.[CAB+18]
2) **Medium Level Interface** The medium level interface presents a more abstract view for the programmer over the SIMD library. It empodies the Low Level Interface from above to a more user-friendly form of programming. The register abstraction data types $mipp :: reg$ and $mipp :: msk$ are encapsulated as template-based objects: $mipp :: Reg < T >$ and $mipp :: Msk < N >$. Therefore, as programmer, we can benefit of the object-oriented concepts of C++ like assignement operators, initializer lists, overloading operators etc, for the price of less flexibility of the low level interface.[CAB+18]

MIPP provides similar features to the *xsimd* library. It gives the programmer the abitility to have even more control than *xsimd* over the portable library with a low level interface, but still has the ability of the more convenient medium level interface, that *xsimd* also has to offer.

5. SIMD Everywhere (SIMDe)

SIMDe[1] has already a wide variety of vector extension sets implemented in the library, with currently a few of them having partial support:

- SSE(2, 3, 4.1)
- AVX, AVX2
- FMA
- SIMD128 (WebAssembly)
  Partial support for:
- AVX-512
- SVE
- NEON

This SIMD-Wrapper is the most unique of the wrappers presented here, as the library does not provide an own abstraction layer per-se, like the other libraries. This means for example, it does not have a register abstraction. Instead, *SIMDe* establishes a method to run code with any of the vector extension sets listed above on almost any machine, independent if the system supports them or not. This way, this SIMD wrapper makes it really simple to make an already implemented vectorized code portable without much effort.
For instance, this allows code with AVX intrinsics to run on ARM machines, that only support NEON/SVE. It even allows for two completely different SIMD intrinsics to exist side by side in the same implementation, like for example AVX and SSE in the same code. The library achieves this by providing the programmer with *SIMDe* function- and type-aliases for the supported instruction set extensions, that are simply the names equivalent to the SIMD intrinsics with "*simde_*" as prefix: $\_mm256\_add\_epi32 \rightarrow simde\_mm256\_add\_epi32$.
The library then attempts to compile the code with the best possible option available to the current system. So, even if the system does not support the implemented vectorization, like for instance AVX, it then tries to replicate the functionality with the available intrinsics: on a SSE system, it makes an effort to, for example, utilize two 128 bit register to simulate the functionality on the 256 bit AVX registers. On a system that supports the native intrinsics, SIMDe garantees identical performance, since the library just replaces the aliases with the original implementation in compile-time.
To better visualize what is happening in the background, we can take a look at the example implementation of the $\_mm256\_add\_epi32$ function of the AVX2 instruction set, from the *SIMDe* wiki on how to add additional functions to the library:

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

```
1  simde__m256i simde_mm256_add_epi32 (simde__m256i a, simde__m256i b) {
2    simde__m256i r;
3  #if defined(SIMDE_AVX2_NATIVE)
4    r.n = _mm256_add_epi32(a.n, b.n);
5  #elif defined(SIMDE_SSE2_NATIVE)
6    r.m128i[0] = _mm_add_epi32(a.m128i[0], b.m128i[0]);
7    r.m128i[1] = _mm_add_epi32(a.m128i[1], b.m128i[1]);
8  #else
9    for (size_t i = 0 ; i < (sizeof(r.i32) / sizeof(r.i32[0])) ; i++) {
10     r.i32[i] = a.i32[i] + b.i32[i];
11   }
12 #endif
13   return r;
14 }
```

Listing 5.1: Example for the SIMDe implementation of an AVX2 function [Nem19]

SIMDe provides a number of precompiler flags, that indicate what SIMD architecture the system supports. This way, we can utilize these flags for the implementation to determine which parts of the code should be compiled and which not. With the preprocessor directive construct

```
#if cond1 ...
    #elif cond2 ...  //multiple #elif are possible
    #else ...
#endif
```

it is possible to tell the compiler beforehand to only add the code, if the corresponding condition is met. Therefore, we can have an implementation for each SIMD-intrinsics, that is only compiled, when the system really supports them.
In the example above we have support for the native AVX2 extension and additionally, the SSE extension. Since the SSE system only has 128 bit registers available, the implementation splits up the addition in two 128 bit instructions, that further can easily be replaced with suitable SSE intrinsics.
In the case that the system does not support any of the intrinsics, most implementations provide a fall-back version of the function, that usually is not optimized with vectorization and works on any machine. In the case of the AVX2 add function, this fall-back consist of a simple for loop over all elements of the simulated registers. Naturally, if a instruction set is not fully supported, the library have to rely on these not vectorized functions most of the time, which could lead to a significant slowdown in comparison to the native implementation.

In contrast to the previous discussed template-based SIMD wrappers, *SIMDe* implements its portability with preprocessor directives. This makes extending the library with better support for instruction sets really simple and add additional SIMD intrinsics is also possible. It does not provide a higher abstraction layer like the other libraries, but is able to easily make already existing SIMD implementations portable without much effort.
Since the library only has partial support for NEON, SVE and AVX512, it is possible we can observe significant decreases in performance when compiling for these extension sets, as not all functions are supported by *SIMDe* and therefore the library has to rely on less performant fall-backs.

Part III.

Implementation and Results

6. Implementation

6.1. Implementation Basis

To calculate the pairwise interations between particles, Autopas provides the *Functor.h* class as a basis. The functor is required to support both Array-of-Structures and Structure-of-Arrays, so when implementing a custom functor we have to implement the following functors:

- **AoSFunctor:** For simulations with the Array-of-Structures data structure it is necessary to implement this functor, which calculates the forces between two particles. Since this functor is generally not vectorizable, it will not be different to the standard implementation.

```
1 Function AoSFunctor(I, J):
      // check if distance between i and j < cutoff
2     F_IJ = CalculateLJ(I, J)
3     I.F+ = F_IJ;
      // Apply newton 3rd law to J
```

For the Structure-of-Array data structure, the implementation of the following three functors are required:

- **SoAFunctorSingle:** This functor takes one SoA as an argument and is intended to calculate the pairwise forces between all particles in the given data structure. This functor is used in the direct sum data structure and with linked cells when calculating the interactions with particles located in the same cell.

```
1 Function SoAFunctorSingle(soa1):
2     for i ← 0 to soa1.size - 1 do
3         for j ← i + 1 to soa1.size - 1 do
              // check if distance between i and j < cutoff
4             F_ij = CalculateLJ(soa1[i], soa1[j])
5             F_i+ = F_ij;
              // Apply newton 3rd law to j
```

- **SoAFunctorPair:** It is also necessary to implement the calculation of the force between the particles of two different SoAs, which both are passed to the functor as

parameters. For example, this finds a use in calculating the particle interactions with the neighbouring cell of the Linked Cell algorithm.

```
1 Function SoAFunctorPair(soa1):
2     for i ← 0 to soa1.size - 1 do
3         for j ← 0 to soa2.size - 1 do
              // check if distance between i and j < cutoff
4             F_ij = CalculateLJ(soa1[i], soa2[j])
5             F_i+ = F_ij;
              // Apply newton 3rd law to j
```

- **SoAFunctorVerlet:** The last functor to implement is the Verlet List Functor. Additionally to the index, the neighbour list (or verlet list) of the particle is passed as parameter. The functor should calculate the force between the indexed particle and all members of the list.

```
1 Function SoAFunctorVerlet(i, soa1, i_neighbours):
2     for j ← 0 to i_neighbours.size - 1 do
          // check if distance between i and j < cutoff
3         F_ij = CalculateLJ(soa1[i], soa2[i_neighbours[i]])
4         F_i+ = F_ij;
          // Apply newton 3rd law to j
```

Autopas has two functors available that vectorize the lennard-jones force calculation using SIMD, one being SVE and the other AVX instrinsics. For this thesis, the AVX-implementation is used as a starting point to implement the SIMD-Wrappers. The implementation is based on the knowledge that we have 256bit registers available, meaning we are able to fit up to four values into a register when using double-precision floating-point numbers.
The functor does the vectorization at the most inner for-loop and thus is able to have up to four particle interactions calculated in one iteration. The basic implementation for the vectorized pairwise particle interaction can be seen in figure 6.2, with $register\_size = 4$.
In some cases, the total number of particles that have to be loaded in happens to be not a multiple of the register size. Like shown in figure 6.1 we could end up having to load only two particles in a register.

First, the functor iterates through the particles until this case occurs. We then choose the mask according to the number of rest particles from the mask-array, that is hard-coded:

```
1   const __m256i _masks[3]{
2     _mm256_set_epi64x(0, 0, 0, -1),
3     _mm256_set_epi64x(0, 0, -1, -1),
4     _mm256_set_epi64x(0, -1, -1, -1),
5   };
```

Listing 6.1: Mask in the AVX-Implementation.

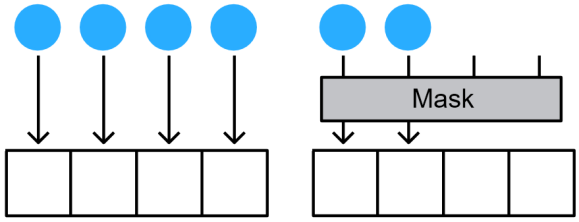


Figure 6.1.: Masked loading of particles

With this mask, we now are able to load the particles into the register without problems and fill up the rest of the register with zero values. The we can calculate the Lennard-Jones force with the masked register.

6.2. Portable Vectorization

When converting the Lennard-Jones functor into performance-portable code, we can not make assumption about the register of the system. Each supported architecture can accommodate various register sizes. AVX or AVX2 can handle up to 256bits, AVX512, as the name already suggests, 512 bits register. SVE offers support for registers up to 2048 bit, but the actual size is determined by the configuration of the processor in the system. The SSE- and Neon-extensions have normally up to 128bit registers available.
As the existing AVX implementation is designed to work exclusively on systems that support AVX-Intrinsics, the implementation can safely presume that the register size has 256 bits. To implement the functor with the different SIMD-Wrappers, each able to support different architectures, we have to made several adjustments to the base implementation:
The following algorithm outlines the fundamental principle of a portable vectorization of the Lennard-Jones functor. This is demonstrated through the pairwise force calculation between two SoAs in Autopas. While the non-vectorized algorithm is only able to calculate one pair i and j each iteration, the vectorization allows processing multiple particles simultaneously by loading their values into SIMD registers.
The one major difference in the basic algorithm, that sets the performance-portable vectorization implementation apart from the AVX implementation, lies within the nested for loop. The AVX implementation can load a maximum of four particle values, considering the use of double-precision floating-point numbers and the index j is incremented by 4 each iteration. However, we do not posess the knowledge of our vector size while implementing our functor and therefore we have to determine the size dynamically. Fortunately, we can get the register length with a function or constant usually provided by the SIMD-Wrappers and then assign it to the variable *register_size*. For instance, the xsimd wrapper offers the number of values of type T as $batch\langle T\rangle::size$, while MIPP provides the function $N\langle T\rangle()$.

Input: soa1, soa2

```
1 Function calculate_pairwise(soa1, soa2):
2     for i ← 0 to soa1.size; ++i do
3         F_i = 0
4         for j ← 0 to soa2.size; j += register_size do
              // check if distance between i and j < cutoff
              // Calculate LennardJones Force between particle i and the
              // particles j to j + (register_size - 1)
5             F_ij = CalculateLJ(i, j)
6             F_i+ = F_ij;
              // Apply newton 3rd
```

Figure 6.2.: Performance-Portable Vectorization of the pairwise LJ-calculation

As mentioned earlier in section 6.1, the base implementation uses a mask-based approach for its implementation. This approach proves advantageous when loading and storing the particle parameters:
If the amount of particles is not a multiple of the current register size, it is possible that an entire register may not be filled with particles. Evidently, adopting the strategy used in the AVX implementation, where all necessary masks are hard-coded(see figure 6.1) is impractical for a performance-portable functor designed to work on various register sizes. Therefore, upon initializing the functor, the mask array must be constructed dynamically based on the register size.

Input: register size
Output: initialized _masks

```
1 Function init_masks(register_size):
2     bool tmp[register_size] = false;
3     for i ← 0 to register_size - 1 do
4         tmp[i] = true;
5         _masks[i].set(tmp);
```

Figure 6.3.: Initializing the masks array dependent on register size

The figure above illustrates the construction of the array containing all the required masks for our functor. We start with an initial mask with only $false$ values. Subsequently, in each iteration, we progressively populate this masks with true values and add the mask with the first i values set to true into the final mask. This way, we ultimately generate masks that exclusively allow the loading and storing of the first, second and subsequent values into the register, regardless of its size.
A similar change to the AVX functor involves retrieving the mixed properties of a particle pair for the lennard-jones calculation, like epsilon, sigma or shift. These values are stored in

a *ParticlePropertiesLibrary* and can be accessed via the defined methods, with "i" and "j" representing the indices of the particles when mixing is required:

```
1        _PPLibrary->mixing24Epsilon(i, j);
2        _PPLibrary->mixingSigmaSquare(i, j);
3        _PPLibrary->mixingShift6(i,j);
```

Listing 6.2: Mixed particle properties

Similar to the masks, the register for the mixed properties are currently initialized in an inflexible way within the existing AVX intrinsics functor. Therefore, we have to rewrite the initialization using a straightforward for-loop, as shown in figure 7, taking the epsilon particle property as an illustrative example. The identical procedure is applied to the other properties by using their corresponding functions from figure 6.2.

Input: mask: particle mask; indexP1 : particle 1; indexP2 : particle 2
Output: mixed epsilon values

```
1 Function init_epsilon(register_size):
2     double tmp[register_size] = 0. ;
3     for i ← 0 to register_size - 1 or mask[i] do
          // get the mixed values of paticle 1 and 2 from _PPLibrary
4         tmp[i] = _PPLibrary.mixing24Epsilon(*indexP1, *(indexP2 + i));
      // load the array buffer into the final register epsilon
5     epsilon = load(tmp);
```

Figure 6.4.: Loading of the mixed epsilon values

6.2.1. xsimd

Implementing *xsimd* was quite straightforward, as most of the instructions from the AVX implementation are supported by the library. Therefore, a significant portion of the implementation could be directly translated, and the concepts discussed in section 6.2 were successfully realized. There is only one problematic exception that posed some challenges: at the time of writing, xsimd does not yet support masked load- and save-operations.Given that the AVX implementation heavely relies on these masks, we had to devise to a work-around for masked loading and storing data:
Instead of directly loading values into the register, we simply use an array with the size of the register to act as a buffer. We load the values into this buffer based on the provided mask. After that, we can load the contents of the buffer into the register using the load operations provided by *xsimd*.
A naive work-around for masked storing is to iterate over each value in a register and individually storing them based on the mask. This method relies on batch indexing, which *xsimd* provides to access individual values within a single batch: $xsimd :: batch < T >:: get(size\_t)$. However, each of these function calls results in a register load and it is not advisable to

implement them in performance-critical sections of the code.[JM]
As a better alternative, we can operate in a manner similar to the solution for the masked-load instruction: we initially store the contents of the register in a buffer array and continue to store only the masked-values into the destination. This approach leads to better performance of the masked store instruction, as it requires only a single register load.
Another notable change we have to take into account is the way the *xsimd* library represents boolean values. While in AVX intrinsics the mask register do not differ from a normal registers, the SIMD wrapper introduces the concept of $batch\_bool < T, Arch >$. Instead of scalar values, this batch type can only hold on boolean values. The purpose of this type of register is to serve as a return value for comparison operators, such as equality testing or ordering of the contents of two batches and other miscellaneous checks. For example, it can be used to check a register with floating point numbers for infinity or not-a-number.
Therefore, in this implementation of the lennard-jones functor, we need to be mindful of this concept. For instance, it is relevant when checking the particles for the cutoff radius, determining whether a particle should be factored into further calculations.

6.2.2. MyIntrinsics++ (MIPP)

The implementation of the functor for the *MIPP* (MyIntrinsics++) extension followed a similar process as that of *xsimd*. Additionaly, *MIPP* offers masked store and load instructions available, unlike the former SIMD-Wrapper. Consequently, the implementation of loading and storing register contents becomes much simpler compared to the *xsimd* functor.
Like *xsimd*, this library also incorporates the concept of conditional registers using the format of $Msk < N > () >$, with N representing the number of desired values in the register. This means, for a mask with the same number of boolean values like the number of double-values that can fit into a register, we can specify it as follows: $Msk < mipp :: N < double > () >$.
Otherwise, all the discussed concepts in section 6.2 were easily translated with the *MIPP* library, with only minor differences in the notation of the library's functions.

6.2.3. SIMD Everywhere (SIMDe)

SIMDe[1] (SIMD Everywhere) takes a different approach compared to the other SIMD-Wrappers. If there is already an existing implementation, the other mentioned functors require often demands extensive rewriting of code, like for example making sure the implementation can work with variable register sizes. The *SIMDe* library provides an simple way to convert existing SIMD intrinsics to be performance-portable.
Since we already have the AVX wrapper available, we will try to convert the AVX intrinsics into a portable implementation: The initial step is to include the appropriate header files from the *SIMDe* library.

```
1 #include < x86/avx.h >
2 #include < x86/fma.h >
```

Figure 6.5.: SIMDe header includes for AVX implementation

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

Since the base implementation is written in AVX intrinsics, we need to include the AVX header, as illustrated in figure 6.5. This enables the AVX support of the library. Additionally, we can include the fma header, which allows us to make use of the fused multiply-add (FMA) operation. This provides an optimized approach to calculate the operation $(a * b) + c$ with only a single instruction.[2] Since not all compilers and architectures that inherently support the AVX extension also support this operation, we must include this header seperately.
The next step in converting the functor involves adding the prefix "simde" to all AVX intrinsics found within the functor.

```
   // AVX intrinsics
1  __m256d tmp = _mm256_loadu_pd(&buf)
   // SIMDe instruction
2  simde__m256d tmp = simde_mm256_loadu_pd(&buf)
```

Figure 6.6.: Porting AVX intrinsics to SIMDe

As depicted in the above figure, the AVX intrinsics used to load values from an array "*buf*" into the 256bit register "*tmp*" can be made portable by adding the "simde"-prefix before the the native instructions. Even for more extensive implementations, such as the lennard-jones functor shown here, it is usually straightforward to convert them to *SIMDe*: Many integrated development environments (IDEs) and text editors provide a select and replace tool that can help to streamline most of the work.

[2] https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-8/mm-fmadd-pd-mm256-fmadd-pd.html

7. Results

7.1. Testing Environement

We will test the three implemented functors with the md-flexible example that Autopas provides. md-flexible allows the user to create and edit certain scenarios and configure the underlying Autopas library.
As testing environment, we choose a domain size of 120x120x120 with periodic boundaries. The cutoff radius will be set to 2.5, no thermostat will be used and no gravity is applied. This allows for a more controlled testing scenario. The simulation is set to run 10000 iterations with a deltaT of 0.00182367.

As an object we add a CubeGrid with a length of x particles for each side and with the Lennard-Jones parameter ϵ and σ all set to 1. This variable x is then gradually increased in order to test scenarios with low and high particle counts. This means, we will end up with x^3 particles in the simulation for each x.

7.2. CoolMUC-2

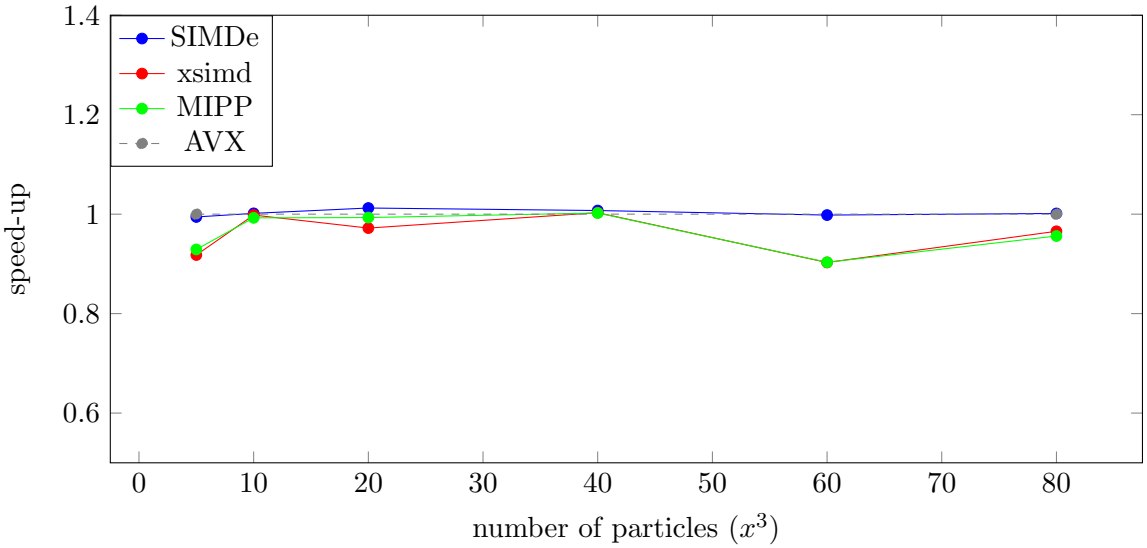
The CoolMUC-2, located in the Leibniz Rechenzentrum, is a HPC cluster by Lenovo. It integrates the "Haswell"-based Intel Xeon E5-2690 v3 as processors. The system has a total number of 812 nodes, each providing 28 Cores with 2 Threads and 64 GB RAM per node with a peak CPU performance of 1400 TFlops/s.[1]

Number of Nodes	812
Cores / Threads per Core	28 / 2
Clock Frequency	2.60 GHz
CPU peak performance	1400 TFlop/s
RAM per node	64 GB

Table 7.1.: CoolMUC-2 specification

In terms of SIMD instruction set extensions, the processor supports AVX2 for vectorization.[Inta] In the following graph, we can see the comparison of the speed-up to the native AVX intrinsics between the three implemented wrappers *SIMDe* (blue), *xsimd*(red) and *MIPP*(green).

[1] https://doku.lrz.de/coolmuc-2-11484376.html



The *SIMDe* functor shows nearly identical results in comparison to the AVX intrinsics, since the speed-up hovers around 1x. This was as expected, since the implementation of the SIMD wrapper is a direct port of the AVX-functor to performance-portable code with the *SIMDe* library, as discussed in section 6.2.3.
xsimd and *MIPP* also do have similar performance results. As they both have the same template-based approach, this was more or less to be expected. With less particles in the simulation, both functors are relatively close to the performance of the native AVX-implementation. With more molecules, the graph deviates and the implementations show an increased slow-down. This could result from the fact that the changes, that were made to the original code base logic - so that we can make the implementation portable with the wrappers (see: 6.2) - influence the performance too much, so that it causes this deviation. For example, we had to implement an alternative for masked store and load, since the *xsimd* library does not have a equivalent function. Naturally, the workaround is significantly slower than only one single instruction, like in the AVX implementation. This could be a reason for the performance losses with greater numbers of particles.

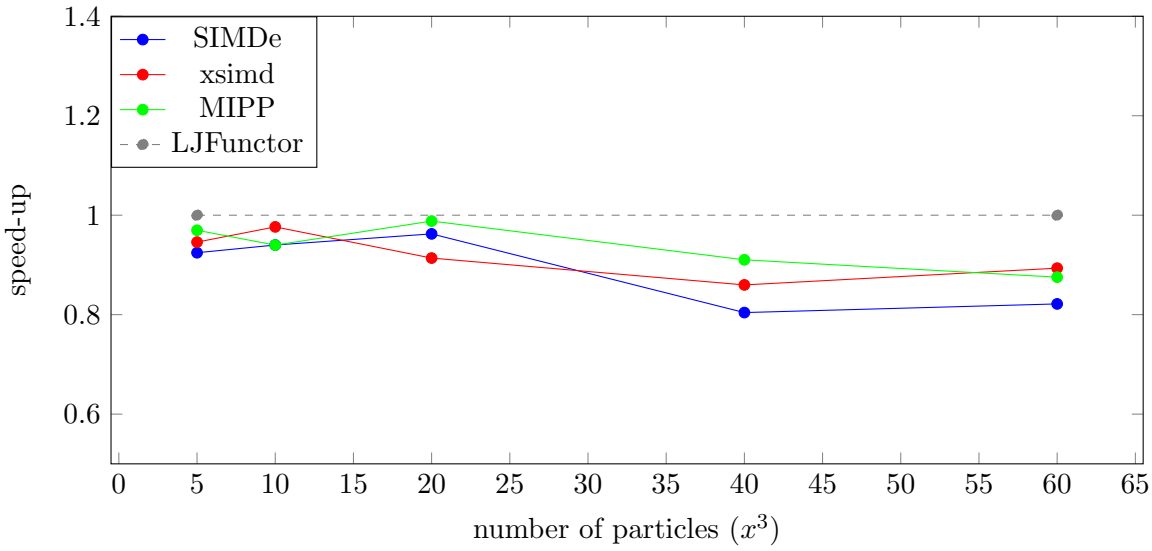
7.3. ARM FX700

The Fujitsu ARM FX700 system is located at the Helmut Schmidt Universität Hamburg and consists of 8 nodes with 48 cores each and 32 GB RAM. The built-in processor is the A64FX which is capable of up to 2.0 Ghz clock speed: [Fuj]

Number of Nodes	8
Cores	48
Clock Frequency	2.00 GHz
CPU peak performance	2.7648 TFLOPS
RAM per node	32 GB

Table 7.2.: ARM FX700 specification

The CPU of the AMD FX700 supports the instruction set Armv8.2-A with support for NEON and SVE. The maximal SVE register size is 512 bit.
Since our Functor *MIPP* does not support the SVE instruction set, we will test the three functors with NEON. We do not have a native NEON-intrinsics functor available, so as comparison we reside to the LJFunctor, which is auto-vectorized with OpenMP:



Firstly, for the *xsimd* and *MIPP* implementation, we can see some resemblances to the test result on the CoolMUC-2 system. Again performing good with a lower number of particles and slightly decreased performance with more particles. This can also be explained by the reasons oulined in section 7.2. The more interesting results is the performance of our third functor, *SIMDe*. This contrasts its excellent results in the AVX environment. The wrapper shows a substantial loss of performance on NEON as both other wrappers perform significantly better in simulations with high number of particles.
The most probable cause for this result is the partial support for NEON in the *SIMDe* library. If the library does not have a suitable vectorized implementation for NEON for a certain AVX function, that was used in the implementation of the *SIMDe* wrapper, *SIMDe* has to rely on the functions fall-back. As discussed in section 5, these fall-backs are usually a sequenced function without vectorization. Therefore, we have to take a significant loss in performance, which is confirmed by the tests.

Part IV.

Conclusion

8. Conclusion

In this thesis, three different SIMD wrappers, *xsimd*, *MIPP* and *SIMDe* were compared and their features presented. It was demonstrated to what extend SIMD wrappers are better or worse than native SIMD intrinsics and if the discussed wrappers are suitable for implementation for a use case in the field of molecular dynamics.

We then extended Autopas, a short-range particle simulation library, with three functors each implementing the Lennard-Jones potential and made them performance-portable with the SIMD wrappers. Each wrapper provides a unique way of SIMD abstraction, which help implementing performance-portable code.
In doing so, some limitations of the wrappers were identified as well: for instance, *xsimd* does not support masked load and store operation yet and a workaround had to be implemented, or wrappers have only limited or partial support for specific instruction set extensions, like NEON/SVE not fully supported by *SIMDe*.

These problems were also present when testing the functors on an AVX-based system, the CoolMUC-2 HPC system and the ARM FX700, which supports NEON and SVE. The tests showed that the functors is able to compete with the native implementation, as the slowdown in normal circumstances is at 10%.
SIMDe was able to demonstrate identical runtime performances on the AVX system. However, when tested with NEON, it was unable to replicate these result due to incomplete support on NEON, with an up to 20% slowdown.

While native intrinsics are still the best option when choosing raw performance, the SIMD wrappers are in many ways the more convenient solution to make your implementation performance-portable without having to sacrifice much in terms of performance.

9. Future Works

In the future, it would be interesting to conduct tests on a wider range of instruction set architectures and analyze their performance on architectures with register sizes of 512 bits or more, like AVX512 and SVE. Given that Autopas already includes a SVE functor, it becomes be possible to compare the wrapper implementations to the SVE intrinsics.
Furthermore, exploring additional optimizations that SIMD-Wrappers could enable and investigating other supplementary features the wrappers implement, like the runtime arch-dispatching capabilities of *xsimd*, would be valuable.
Given that the *SIMDe* library does not have full support for NEON and SVE yet and therefore does not perform well on ARM systems, it might be worth looking into expanding the existing support of these architectures and comparing it with the performance to before. Also, since we have another SIMD functor with SVE intrinsics available, making it portable with *SIMDe* and comparing it with the portable AVX implementation from this thesis would be quite interesting.

Part V.

Appendix

List of Figures

List of Tables

Bibliography

[CAB+18] Adrien Cassagne, Olivier Aumage, Denis Barthou, Camille Leroux, and Christophe Jégo. Mipp: A portable c++ simd wrapper and its use for error correction coding in 5g standard. In *Proceedings of the 2018 4th Workshop on Programming Models for SIMD/Vector Processing*, WPMVP'18, New York, NY, USA, 2018. Association for Computing Machinery.

[CD90] Ariel A. Chialvo and Pablo G. Debenedetti. On the use of the verlet neighbor list in molecular dynamics. *Computer Physics Communications*, 60(2), 1990.

[EGF+12] Pierre Estérie, Mathias Gaunard, Joel Falcou, Jean-Thierry Lapresté, and Brigitte Rozoy. Boost.simd: Generic programming for portable simdization. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[Fom11] Eduard S. Fomin. Consideration of data load time on modern processors for the verlet table and linked-cell algorithms. *Journal of Computational Chemistry*, 32(7):1386–1399, 2011.

[FR81] David Fincham and B.J. Ralston. Molecular dynamics simulation using the cray-1 vector processing computer. *Computer Physics Communications*, 23(2):127–134, 1981.

[Fuj] Fujitsu Limited. Intel xeon processor e5-2690 v3 specification. `https://www.fujitsu.com/downloads/SUPER/primehpc-fx700-datasheet-en-202302.pdf`. Accessed: 03.10.2023.

[GSBN21] Fabio 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, 12 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 Workshops (IPDPSW)*, pages 748–757, 2019.

[Inta] Intel Corporation. Intel xeon processor e5-2690 v3 specification. `https://www.intel.com/content/www/us/en/products/sku/81713/intel-xeon-processor-e52690-v3-30m-cache-2-60-ghz/specifications.html`. Accessed: 03.10.2023.

[Intb] Intel Corporation. Requirements for vectorizable loops. `https://www.intel.com/content/www/us/en/developer/articles/technical/requirements-for-vectorizable-loops.html`. Accessed: 25.07.2023.

[JM] Sylvain Corlay Johan Mabille. xsimd documentation. `https://xsimd.readthedocs.io/`. Accessed: 27.07.2023.

[MG07] Stephan Knapek Michael Griebel, Gerhard Zumbusch. *Numerical Simulation in Molecular Dynamics*. Springer Berlin, Heidelberg, 1 edition, 2007.

[Nem19] Evan Nemerson. Implementing a new function. `https://github.com/simd-everywhere/simde/wiki/Implementing-a-New-Function`, 2019. Accessed: 01.08.2023.

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

[YWLC04] Zhenhua Yao, Jian-Sheng Wang, G.R. Liu, and Min Cheng. Improved neighbor list algorithm in molecular simulations using cell decomposition and data sorting method. *Computer Physics Communications*, 161:27–35, 08 2004.