SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Simulation of Supercooled Argon Gas using the Smoothed Lennard-Jones Potential in md-flexible

Ivander Alson Tanjaya

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Simulation of Supercooled Argon Gas using the Smoothed Lennard-Jones Potential in md-flexible

Simulation von unterkühltem Argongas unter Verwendung des smoothed Lennard-Jones-Potentials in md-flexible

Author:	Ivander Alson Tanjaya
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Samuel James Newcome, M.Sc.
Submission Date:	10.10.2024

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

Munich, 10.10.2024 Ivander Alson Tanjaya

Acknowledgments

I want to thank Samuel James Newcome, M.Sc, for his advice and help during this Thesis, and also Luis Gall and the Autopas team for providing the basis code for this project. I also gratefully acknowledge the computational and data resources provided by the Leibniz Supercomputing Centre[1].

[1] `www.lrz.de`

Abstract

In molecular dynamics, the Lennard Jones potential is a popular first choice for the force calculation between particle pairs, but for some scenarios, such as supercooling of Argon, the standard truncated Lennard Jones potential might introduce inaccuracies in the simulation due to the sudden jump at the cutoff point[GKZ07]. This paper presents the implementation of the vectorized smoothed Lennard Jones potential using three different approaches: auto-vectorized, masked, and gather scatter. Furthermore, the SIMD implementation done in this project is written using the SIMD wrapper library Google Highway to provide a balance of portability and performance.

Contents

Part I.

Introduction and Theoretical background

1. Introduction

Molecular dynamics (MD) simulations have become an important tool in scientific research, as they can provide information on how atoms and molecules interact by simulating their movements. This can help scientists prototype quickly without the costs associated with physical experiments. MD is currently used in almost every field of research, from materials science to physics; an example of their use is in simulating experiments of phase transition in noble gases such as Argon[Rut+17]. The traditional choice used to calculate the force exerted between particles is the Lennard-Jones potential[Len31]. This potential describes the interaction between a pair of atoms or molecules. It is characterized by a balance between attractive and repulsive forces, with the attractive force caused by van der Waals interactions and the repulsive forces arising from the Pauli exclusion principle at shorter distances.

The Lennard-Jones potential is a short-range potential, which means that it quickly converges to zero and the greater the distance between the particles. Thus, the accuracy gained by calculating the force exerted between these far-across particles provides a diminishing return but has a great computational cost. The solution usually taken is to provide a cutoff distance, a predetermined distance in which the force exerted between particles exceeding this distance would be counted as 0. However, another problem arises: there is now a discrete jump in force when approaching the cutoff distance. This discrete jump in force can impact accuracy; the proposed solution is to use a modified version of the Lennard Jones potential that smoothly goes to 0 at the cutoff distance[GKZ07].

Furthermore, since the bulk of the runtime of a simulation is spent on the force calculation, many optimization techniques have been used to reduce its computational cost. One of the most common methods is single instruction multiple data (SIMD), in which multiple data can be processed at once; this is very useful during force calculation, as multiple particles can have their force calculated at once, massively reducing the runtime.

In this work, we will discuss the proposed smoothed Lennard Jones potential and three different vectorized implementations of this potential: auto-vectorized, masked, and gather scatter and test them on CoolMuc4 to compare their performance.

2. Theoretical Background

2.1. Molecular Dynamics

Molecular dynamics simulation is a powerful tool that is used to simulate interaction between particles in a given time period, it simulates the movement of the particles by calculating the force exerted on each particle by others in given set of time and updates their positions. A popular choice for this force calculation is the Lennard-Jones potential.

2.1.1. Lennard Jones

The Lennard-Jones potential[Len31] is a short-range potential that describes the change in the force exerted between a particle pair given their distance. The following formula defines this potential:

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

Where r_{ij} is the distance between the particle pair, where σ and ε are parameters defined by the model based on the type of particle. The Lennard-Jones potential converges to zero quickly, continuing to calculate forces between particles that are far apart, providing quickly diminishing returns on the accuracy. To reduce the computational cost of the simulation, a cutoff radius is chosen, and only particle pairs within this distance have their force calculated.

2.1.2. Smoothed Lennard Jones

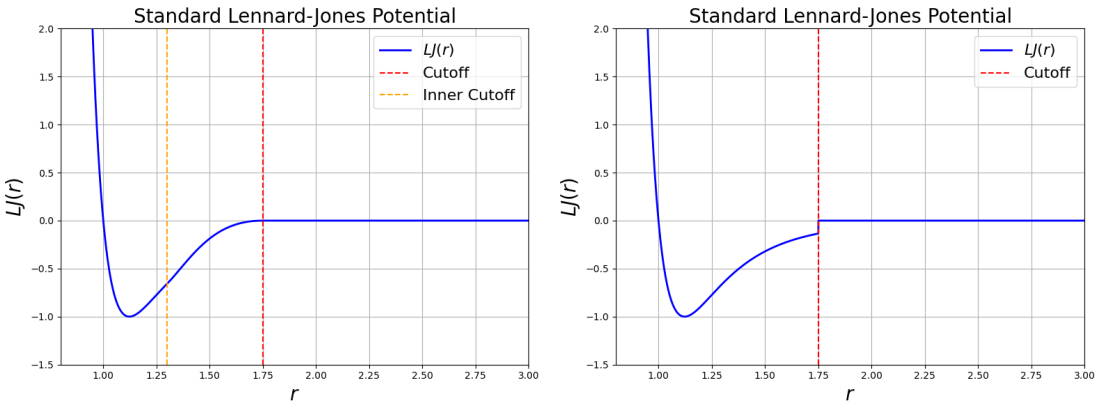
A problem arising from the standard Lennard Jones potential is a sudden jump at the cutoff distance. This discrete jump can cause inaccuracies in the simulation[GKZ07]; a solution to this problem is to smoothen the curve of the Lennard Jones potential, thus making it go to 0 at the cutoff distance without the sudden jump. This is achieved by multiplying a smoothing factor by the original formula. This specific definition of the

smoothed Lennard Jones potential was obtained from [GKZ07].

$$U(x_i, x_j) = 4\epsilon \cdot S(x_i, x_j) \cdot \left[\left(\frac{\sigma}{x_i - x_j} \right)^{12} - \left(\frac{\sigma}{x_i - x_j} \right)^{6} \right] \tag{2.2}$$

$$S(x_i, x_j) = \begin{cases} 1 & : ||x_i - x_j||_2 \leq r_l \\ 1 - \frac{(||x_i - x_j||_2 - r_l)^2 \cdot (3r_c - r_l - 2||x_i - x_j||_2)}{(r_c - r_l)^3} & : r_l \leq ||x_i - x_j||_2 \leq r_c, \\ 0 & : ||x_i - x_j||_2 \geq r_c \end{cases} \tag{2.3}$$

In this factor, the term r_l denotes the inner cutoff, and r_c the original cutoff distance. Particle pairs below the inner cutoff would experience the normal Lennard Jones potential, but for pairs that lie between the r_l and r_c, the smoothing factor would scale down the potential such that it would smoothly go down to 0 at r_c.



(a) Smoothed Lennard Jones

(b) Standard Lennard Jones

Figure 2.1.: 2.1a shows the Lennard Jones that has been smoothed so that it goes to 0 at r_c, 2.1b shows the normal truncated LJ potential

2.1.3. Newton's third law

According to Newtons third law of motion[New87], for every force exerted on a body i by j, there must be an equal force exerted in the opposite direction on j. This fact can be used in our simulation to halve the calculations needed to be done by subtracting the force exerted on i from j, thus updating the force of two particles at once.

2.2. Autopas

Autopas[1] is a molecular dynamics library that provides runtime tuning for its simulation. In molecular dynamics, there are many different configurations and algorithms that have to be chosen to provide the optimal result for a given scenario[Gra+19]. Determining the optimal configuration for a scenario non-trivial and require extensive knowledge in both computer science and physics, thus Autopas aims to solve this problem by dynamically adjusting the configuration during runtime, even automatically changing to the optimal configuration to adjust to the state of the simulation.

2.2.1. Data Layout

Autopas supports two data structures that are also dynamically adjusted during runtime. These data structures contain information alike the particles position, velocity and force, the difference between the two is how they are stored in memory.

1. AoS (Array of structure): In this data structure the data for a single particle is packaged in a single particle data type, these particles are then stored inside an array like container. This data structure is easier to navigate but is not optimal for vectorization, as the data isn't stored contiguously in memory.

2. SoA (Structure of Array): In this data structure, each data point for the particle is stored in different array containers, and these arrays are stored in an overarching SoA data structure. This layout provides good support for vectorization since the attributes of the same type are stored contiguously in memory, making it easier to load multiple values for vector instructions.

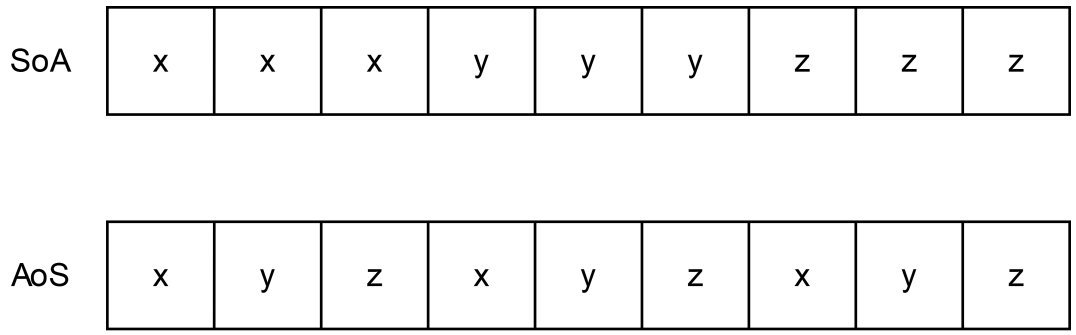


Figure 2.2.: AoS and SoA layout

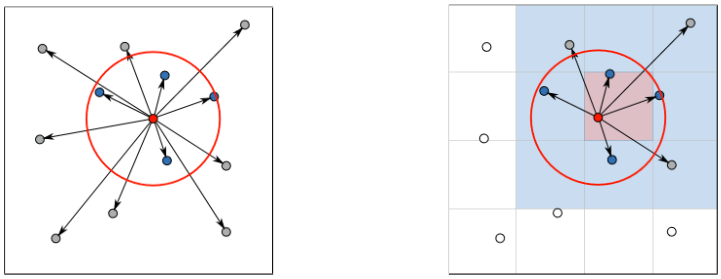
[1] https://github.com/AutoPas/AutoPas

2.2.2. Particle Container

In N body simulations, a problem arises with the exponential growth of the number distance calculation with the number of particles. Autopas deals with this problem by providing the user with multiple particle containers. Each container implements different ways to determine which particle pairs are relevant to the force calculation. This can provide huge performance increases in the simulation. In this project, we will discuss two different containers in Autopas:

1. Direct sum: The simplest approach to this problem is to just to perform the distance calculation for all particles in the container, although this approach would have a time complexity of $O(N^2)$ as each particle has to have its distance to every other particle calculated, despite the high computational cost, this approach have the advantage of avoiding costly overheads and or storing complicated data structure[Gra+19].

2. Linked cells: A way to avoid calculating the force for particles outside the cutoff range is to divide the domains into distinct cells with widths bigger or equal to the cutoff range. In this way, only neighboring cells needed to be included in the force calculation. Because for each cell, there is a constant amount of cells neighboring it, if the number of cells is chosen proportionally with the number of particles, a computational complexity of $O(N)$ can be achieved[Gra+19].



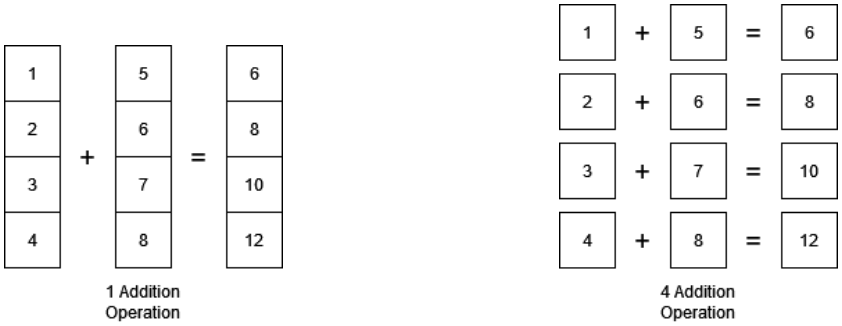
(a) Direct Sum (b) Linked Cells

Figure 2.3.: Relevant Autopas containers [Gra+19]

2.3. SIMD

SIMD stands for single instruction multiple data[Fly72]; it is a type of parallelization method where a single instruction can be applied to multiple data points; another word

for this type of operation is vectorization, as it can be visualized as doing mathematical operations using vectors of numbers instead of individual numbers. In Autopas, SIMD is implemented in multiple ways, one of them is to rely on the compiler to automatically convert non-SIMD code into SIMD intrinsics. This is not ideal, as more complex data flow makes it hard for the compiler to correctly vectorize the code. Another way to implement SIMD is to write it directly using SIMD intrinsics



(a) SIMD (b) Scalar

Figure 2.4.: SIMD and Scalar addition

CPU designers like Intel and Arm provide built intrinsics that can be used to implement vectorization into the code. Although code that is made using native SIMD intrinsics tends to perform much better than auto-vectorized code, it is bound to a specific CPU architecture since the intrinsics are architecture-specific, for example, AVX2(x86), AVX512(x86), SVE(Arm). These SIMD intrinsics also have their nuances and names, making it quite difficult to port programs made for a certain architecture to another. One solution to this problem is SIMD wrappers; these SIMD wrappers are libraries that provide a layer of abstraction above the native intrinsics, such that code written using these wrappers would be able to run on other architecture while preserving the performance gain you get from using explicit intrinsics.

2.3.1. SIMD Terms

Here are some commonly used SIMD terminologies used in this project.

- Mask: a binary array that is used as an extra input for some operations; the result of these operations would change depending on whether the value of the mask at that index is equal to 1.
- Compress: Compress operation applies a mask to a vector that stores all values with a positive mask contiguously on the lower side and sets all remaining indices

to 0.

- Align: Concatenate two vectors together, shift them by a given amount, and return the lower half.
- Gather/Scatter: Gather and Scatter instructions allow the user to load or write data in a nonaligned manner by providing a vector of indices that are used as an offset to the base address of the data to be accessed.

2.3.2. Autovectorization

Modern C++ compilers like GCC utilize auto-vectorization to automatically turn scalar code into vectorized code[Fou], improving performance greatly. By analyzing the code, compilers can automatically identify vectorizable code segments and compile a vectorized version of the segment[Jel23]. This can provide the user with the performance benefits of SIMD while avoiding the complexities associated with it. It also helps with portability across different architectures as the compiler uses the native SIMD intrinsics available to the target architecture.

Auto-vectorization, however, has its drawbacks, as the developer does not have complete control over which SIMD intrinsics are used to vectorize the code and which code segments would even be vectorized[Jel23]. This may lead to sub-optimal performance in some scenarios. The process is also highly compiler-dependent, with the auto-vectorization process varying between different compilers. Moreover, debugging the code generated by auto-vectorization can be difficult, as the transformations made by the compiler obscure the original code's logic.

Tools like OpenMP help alleviate some of these issues by using directives like `#pragma omp simd` to tell the compiler that a loop segment can be vectorized[DM98]. This tool helps developers guide the compiler on which sections of the code can be safely vectorized.

2.3.3. Google highway

The Google Highway SIMD wrapper[Goo24] is a C++ library that provides a layer of abstraction to make SIMD code portable for multiple architectures. The library provides functions that translate to the corresponding native SIMD intrinsics on the compiled machine. This means that code written using Google Highway can be compiled into multiple versions according to the chosen architecture.

For example the Highway instruction `hwy::add(a,b)` when adding 2 double vectors would compile to `_mm256_add_pd(a, b)` for AVX2 and `_mm512_add_pd(a, b)` for

AVX512. This layer of abstraction makes it much easier to program in SIMD parallelism while making the code generic enough to be able to be ported to multiple architectures.

Currently, Google Highway is able to be compiled into 24 different architectures; some notable examples include x86 (SSE, SSE2, AVX2, AVX3, etc.), ARM(SVE, SVE2, SVE_256, etc.), the complete list of targets is available here 8.1. Google Highway, in particular, is chosen for this project due to its easy-to-read syntax and minimal performance cost[Roc23].

2.3.4. Vectorization techniques

1. Masked: Masked vectorization is used in autopas to implement the cutoff mechanic discussed in 2.1.1. Since particle pairs with a distance more than the cutoff shouldn't be included in the force calculation, a binary mask calculated from the distance of each pair is created and applied at the end to make the force 0 depending on the value in the mask at the corresponding indices. A major drawback to this approach is that for lower hitrates, a lot of calculations are wasted since the results would be masked away at the end.

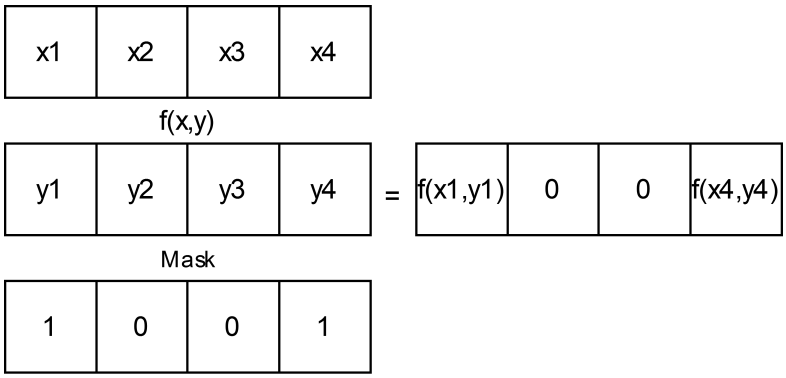


Figure 2.5.: Masked Operation

2. Gather scatter: Gather scatter seeks to improve performance at lower hitrates by storing relevant indices and only calculating their forces when the indices vector is full; this way, only particle pairs inside the cutoff would have their force calculated. This approach is not without downsides since there is significant overhead with the gather and scatter instruction, and in general, the data flow is more complex than the masked approach.

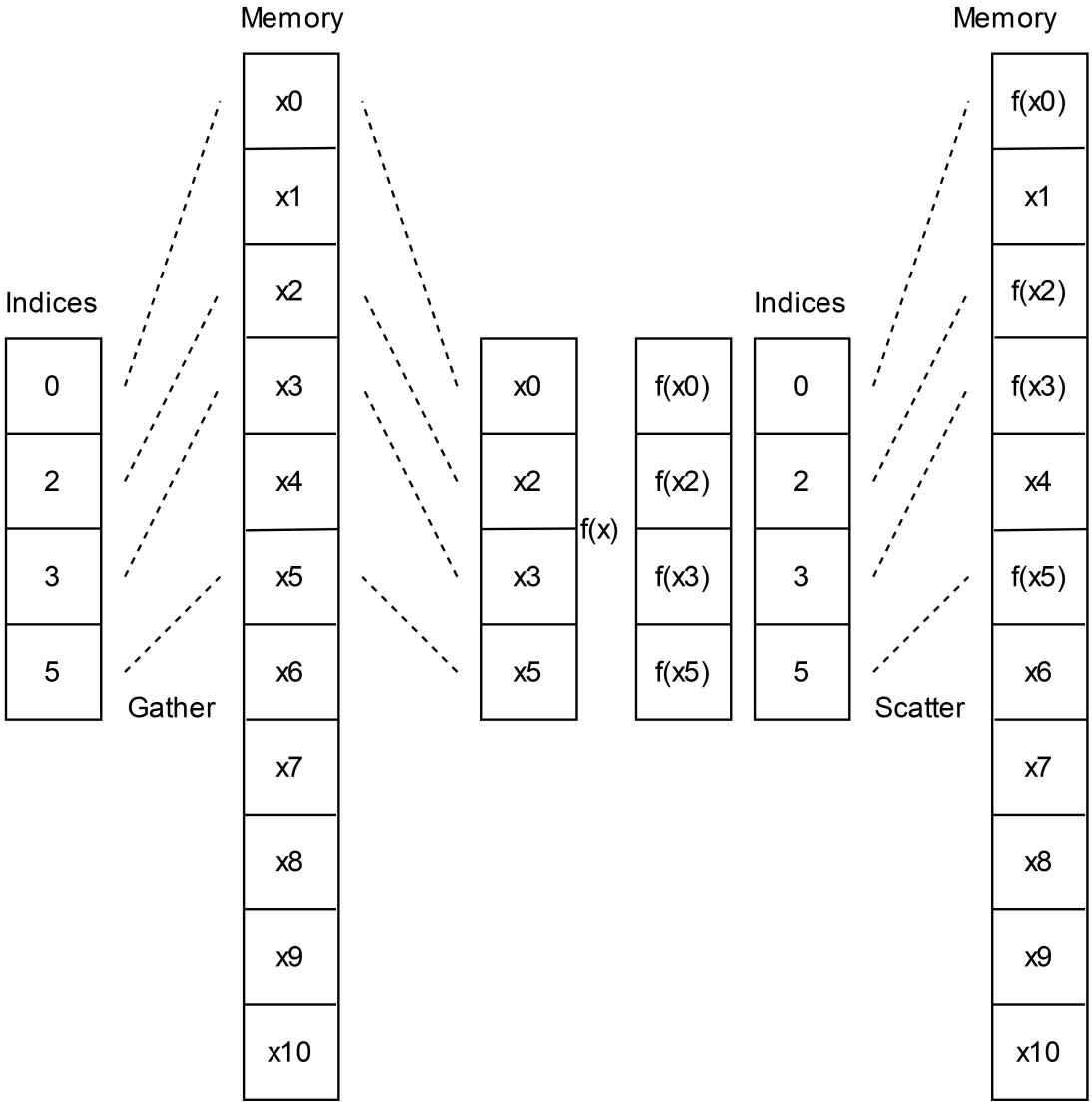


Figure 2.6.: Gather-Scatter operation

3. Related Works

While there are many other works on the theme of vectorization of MD potentials, such as [Eng24] for the Axilrod-Teller potential and [Col24] for the mie potential, [WN19] for the standard 12,6 Lennard Jones potential with gather scatter, the vectorization of the smoothed Lennard Jones functor that implements the gather scatter approach that takes into account the piecewise nature of the function has not been found during the literature search.

Unlike implementations of the smoothed Lennard Jones functor such as one made by LAMMPS[Tho+22], we included a gather scatter approach, which we theorized would be beneficial due to the higher computational cost of the smoothed LJ potential(2.1.2) compared to the standard 12,6 LJ.

Additionally, previous implementations of the vectorized potentials are mostly written in native intrinsics intended for a specific architecture. The usage of Google Highway in this project also aims to make the code more portable and easily maintainable for multiple platforms.

Part II.

Implementation

4. Implementation of smoothed Lennard Jones force

To implement the smoothed Lennard Jones force, we need to make a conditional branch that applies the smoothed force exclusively on particle pairs with distances between the inner cutoff and cutoff. Every functor has an SoA and AoS implementation of the Lennard Jones force; we would start by describing how the conditional branch is implemented in the AoS version. Below is a pseudocode representation of how the force calculation is implemented in the AoS Functor. Throughout this paper, the term r_l denotes the inner cutoff and r_c the normal cutoff.

Algorithm 1: Smoothed AoS Functor

```
1 displacement ← x_i − x_j
2 distance^2 ← displacement · displacement
3 if distance^2 >= r_c then
4 |   return
5 else if distance^2 >= r_l then
6 |   f ← calculateSmoothedForce(i, j, r_c, r_l)
7 else
8 |   f ← calculateStandardForce(i, j)
9 end if
```

This algorithm is based on the existing AoS functor, with the difference being the difference force calculation depending on the distance of the particle pair between the inner cutoff and cutoff. In contrast, due to the vectorized nature of the SoA functor, adding an if statement inside that checks for each pair is not possible; thus, three vectorization approaches are implemented in this project. The first one is the auto-vectorized approach, where we rely on the compiler to automatically vectorize our scalar code; the second is the masked vectorization approach; in this method, we make a binary mask for each element pair that is set to one if the distance between them is less than equal to the cutoff distance. The third one is the gather scatter approach that stores indices falling inside the cutoff range and only sends full arrays to the SoA kernel.

4.1. Autovectorized approach

The autovectorized approach is essentially a direct translation from the pre-existing implementation of the Lennard-Jones Functor in `LJFunctor.h` provided by AutoPas. In this implementation, no explicit SIMD intrinsics are used, and we rely on the compiler to correctly generate vectorized code. We can also guide the compiler to vectorize certain code segments using OpenMP[DM98] pragmas; these pragmas tell the compiler how the code segment should be auto vectorized, for example, Algorithm 2 shows how the SoAFunctorSingle functor processes the particles, the line `#pragma omp simd` tells the compiler that the loop can be vectorized.

Algorithm 2: Main loop structure of the Autovectorized SoASingle Functor

```
1  for i = 0 to soa1.size() − 1 do
2      #pragma omp simd
3      for j = 0 to soa2.size() − 1 do
4          displacement ← x_i − x_j
5          distance^2 ← displacement · displacement
6          mask ← distance^2 <= r_c
7          innerMask ← distance^2 >= r_l & distance^2 <= r_c
8          force ← mask ? calculateStandardForce(i, j) : 0
9          force ← innerMask ? calculateSmoothedForce(i, j, r_c, r_l) : force
10     end for
11 end for
```

The primary modification involves the integration of a smoothing term within the second loop. This is done by adding another variable called innerMask, which is applied to the calculated smoothing value and makes it 1 if the distance is less than the innerCutoff. The implementation details for the pair and verlet functors are analogous to how the single functor.

Even though auto-vectorization works fine for simple code, as soon as the data and control flow become more complex, the compiler may be unable to correctly identify vectorizable code segments by itself. This can cause the code to run much slower than explicitly vectorized code.

4.2. Masked approach

The masked implementation is, in principle, very similar to the auto-vectorized version, the biggest difference is that here, we use Google Highway that maps directly to

native SIMD intrinsics for the target architecture instead of relying on the compiler to auto-vectorize the code.

Previously, there were two major downsides of writing vectorized code using native intrinsics, firstly it is much more difficult to write native vectorized code than the auto-vectorized version due to the domain knowledge of the specific architecture required to write correctly vectorized programs, secondly, the written program would only be usable on the specific architecture it is written for and cannot be easily rewritten for other architectures. However, by using Google Highway, the development cost of writing explicitly vectorized programs is greatly reduced. The force calculation in the SmoothedSoAKernel would be done similarly as in algorithm 2.

The main loop structure of the masked approach is as shown here:

Algorithm 3: Main loop structure of the Masked SoASingle Functor

```
for i = 0 to soa1.size() − 1 do
    for j = 0 to (soa2.size() & ∼ (_vecLengthDouble − 1)) step _vecLengthDouble
     do
        SmoothedSoAKernel(i, j)
    end for
    SmoothedSoAKernelRest(i, j)
end for
```

Here, the variable `_vecLengthDouble` is automatically set by Highway as the target architecture's vector length. In this way, the algorithm would still work for different vector lengths. Other highway variables are also used to automatically adjust to the target architecture. For another example, listing 4.1 shows a generic double vector that automatically adjusts itself for the target architecture.

```
1 const highway::ScalableTag<double> tag_double;
2 using VectorDouble = decltype(highway::Zero(tag_double));
```

Listing 4.1: Google Highway example of a generic double vector type

The SoA kernel is also written generically using Google Highway so that the code can run on multiple architectures. To do this, we converted the SIMD intrinsics into its highway equivalent. Listing 4.2 shows how the force calculation part of the SoA kernel is written using Google Highway.

```
1 // compute LJ Potential
2 const VectorDouble invDr2 = highway::Div(_oneDouble, dr2);
3 const VectorDouble lj2 = highway::Mul(sigmaSquareds, invDr2);
4 const VectorDouble lj4 = highway::Mul(lj2, lj2);
```

```
5 const VectorDouble lj6 = highway::Mul(lj2, lj4);
6 const VectorDouble lj12 = highway::Mul(lj6, lj6);
7 const VectorDouble lj12m6 = highway::Sub(lj12, lj6);
```

Listing 4.2: Force calculation in the masked kernel

```
1 // compute LJ Potential
2 const SoAFloatPrecision invdr2 = 1. / dr2;
3 const SoAFloatPrecision lj2 = sigmaSquared * invdr2;
4 const SoAFloatPrecision lj6 = lj2 * lj2 * lj2;
5 const SoAFloatPrecision lj12 = lj6 * lj6;
6 const SoAFloatPrecision lj12m6 = lj12 - lj6;
```

Listing 4.3: Force calculation in the autovectorized kernel

In comparison to listing 4.3, 4.2 implements all operations directly as vectors while the variables in 4.3 are written as scalar values that need to be vectorized by the compiler. The highway functions used in 4.2 also directly translate to SIMD intrinsic for the target architecture during compilation.

Both the Autovectorized and Masked versions use masking to implement the cutoff mechanism. A problem with this approach is that the force and smoothing term still have to be calculated, even for particle pairs that lie outside of either cutoff. In higher hitrates, this would not cause a problem, but at lower hitrates, this algorithm would perform many unnecessary calculations that would be masked away at the end.

4.3. Gather Scatter approach

The gather scatter approach provides various benefits compared to the standard masked approach. One major difference is that in the masked implementation, the force and smoothing factor would have to be calculated for each particle pair regardless of distance; this problem is even more important in the smoothed Lennard Jones potential than in the standard implementation due to the additional cost of calculating the smoothing term.
In this implementation of the gather scatter, the distance is pre-calculated before being sent to the kernel, those that fall between the inner cutoff and cutoff would be collected and sent to the smoothed kernel, and particle pairs that have a distance of less than the inner cutoff would be sent to the normal kernel instead. This saves expensive operations only for those particles that actually need it and avoids wasted operations. This gathering of indices happens inside the second loop of the functor. Below is a pseudocode describing how the indices are gathered.

Algorithm 4: Second loop for gathering separate Indices

```
1  indicesSmooth ← ∅
2  indicesStandard ← ∅
3  for j = 0 to soa2.size() − 1 do
4  |  if distance^2 > r_c then
5  |  |  continue
6  |  else if distance^2 > r_l then
7  |  |  indicesSmooth ← indicesSmooth ∪ {j}
8  |  else
9  |  |  indicesStandard ← indicesStandard ∪ {j}
10 |  end if
11 end for
```

To implement this, changes were made to the masks used for the inner and outer cutoff. The Cutoff mask previously is a mask that is set to 1 if the distance < r_c and for innerCutoffMask id distance < r_l. The modified mask would separate particles that would experience exclusively the smoothed potential and the ones that would exclusively experience the standard LJ potential. Since the we want particles that is both greater than r_l and less r_c, the new InnerCutoffMask is simply the result of a logical and operation with the old cutoffmask. Since r_l is always smaller than r_c, the new cutoffMask can simply be all the particles with distance less than r_l. With this, the old cutoff mask can be split into two separate masks.

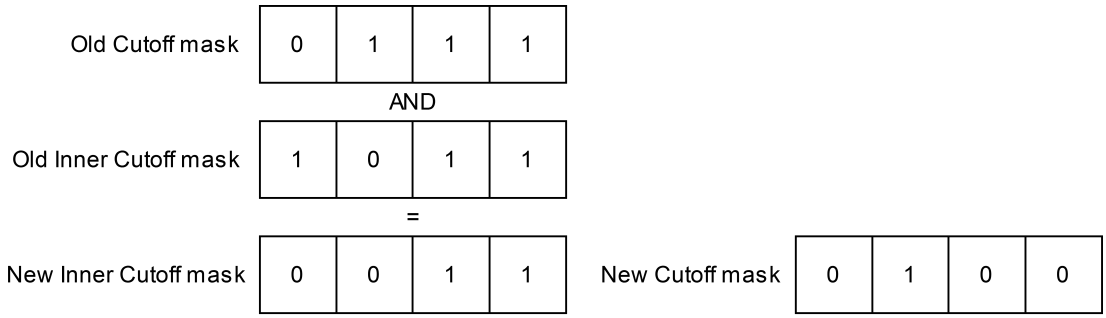


Figure 4.1.: New Masks

The masks are then applied to the current indices being loaded into the loop. In each loop, the number of 1s in each mask is counted and added to the current number of indices stored in the accumulator. This is then compared with the length of the vector register to see enough indices has been gathered to be sent to the SoA kernel.

Algorithm 5 shows how the indices are gathered and sent to the appropriate SoA kernel. With this. The masks used in this algorithm are the newly calculated masks from figure 4.1.

Algorithm 5: Compress and align

```
popCountSmooth ← countTrue(innerCutOffMask)
popCountStandard ← countTrue(cutOffMask)
if numAssignedRegisters + popCountStandard < _vecLengthDouble then
    newInteractionIndices ← compress(...)
    interactionIndices ← alignr(...)
    numAssignedRegisters ← numAssignedRegisters + popCountStandard
else
    newInteractionIndices ← compress(...)
    interactionIndices ← alignr(...)
    SoAKernelGS(interactionIndices, ...)
    interactionIndices ← mask already processed indices
    interactionIndices ← alignr(...)
    numAssignedRegisters ←
      popCountStandard − _vecLengthDouble + numAssignedRegister
end if
if numAssignedSmoothRegisters + popCountSmooth < _vecLengthDouble then
    newInteractionSmoothIndices ← compress(...)
    interactionSmoothIndices ← alignr(...)
    numAssignedSmoothRegisters ←
      numAssignedSmoothRegisters + popCountSmooth
else
    newInteractionSmoothIndices ← compress(...)
    interactionSmoothIndices ← alignr(...)
    SoAKernelSmoothGS(interactionSmoothIndices, ...)
    interactionSmoothIndices ← mask already processed indices
    interactionSmoothIndices ← alignr(...)
    numAssignedSmoothRegisters ←
      popCountSmooth − _vecLengthDouble + numAssignedSmoothRegister
end if
process rest...
```

One problem with Google Highway was the lack of an alignr function for AVX2. Thus

a custom alignr function was implemented for this project that emulates the behaviour of `_mm512_alignr_epi64`. This alignr function would concatenate two vectors together, shift the vector by $shift$ elements, and return the lower half.

Algorithm 6: Alignr

Data: Vector a, Vector b, Int Shift
Result: Aligned Vector

$concatenated[2 * sizeofVectorRegister]$
$Store(a, concatenated)$
$Store(b, concatenated + sizeofVectorRegister)$
$Return\ Load(concatenated + shift)$

Google Highway also allows the user to make conditional statements based on the target architecture. We can use the highway-provided function call for alignr: `highway::detail::CombineShiftRightI64Lanes<x>(b,a)` in targets where it is available instead of our custom implementation. Here x is how much the concatenated vector is shifted to the right and b, and a are the hi and lo vector, respectively.

Algorithm 7: Target Detection

Data: Vector a, Vector b, Int Shift
Result: Aligned Vector

if $HWY\_TARGET <= HWY\_AVX3\_DL$ **then**
  $highway :: detail :: CombineShiftRightI64Lanes < shift > (b, a);$
else
  $alignr(a, b, shift)$ $//Algorithm\ 6$
end if

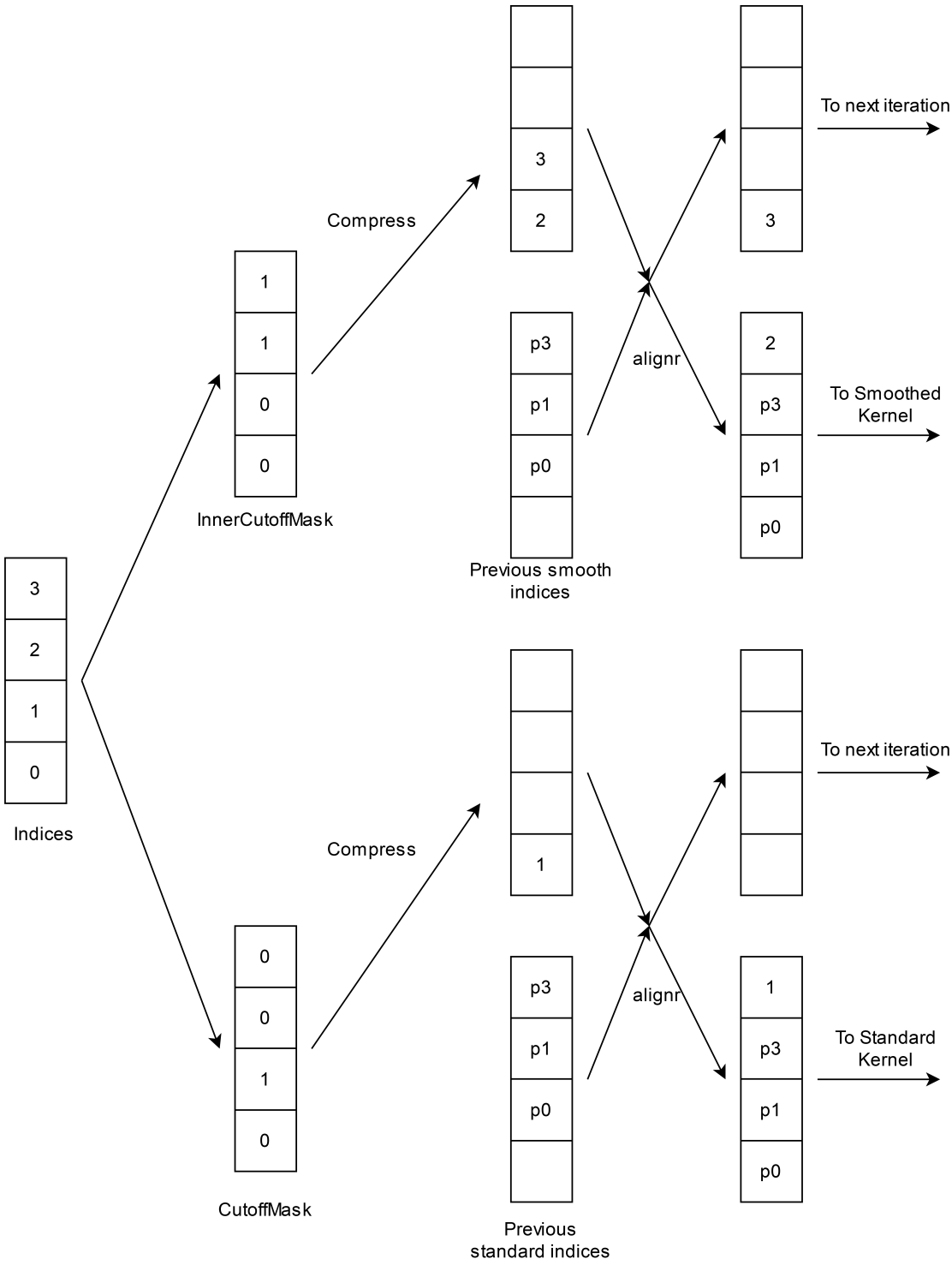


Figure 4.2.: Visualization of the separated compress and align algorithm

Part III.

Results and Conclusion

5. Results

5.1. Hardware

The tests were done on CoolMuc4 due the native support for gather scatter instrinsics provided by AVX512. The commit id for the version tested is 59dbd56. The Hardware specification of the cluster is as follows:

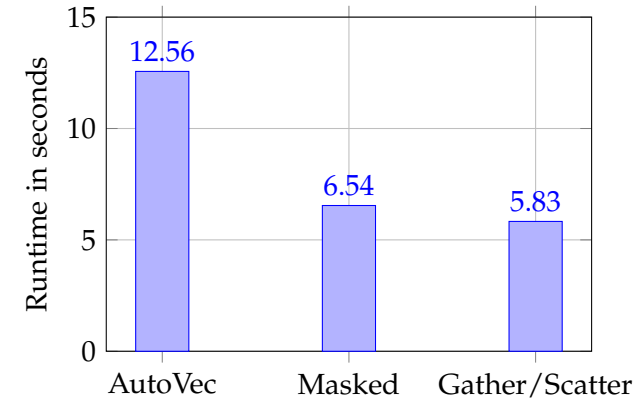
Table 5.1.: Hardware overview[Eng24].

	CoolMUC-4
CPU	Intel®Xeon®Platinum 8380
CPU Architecture	Icelake
Frequency	2.3 GHz
Vector Extensions	SSE, AVX, AVX2, AVX-512

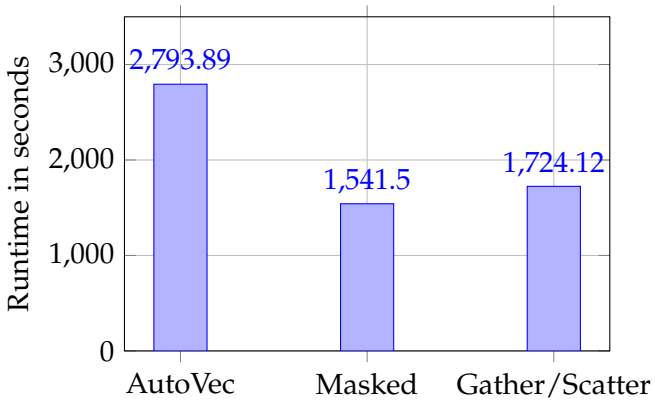
5.1.1. Argon simulation

MD-Flexible is a molecular dynamics simulator provided by Autopas; in this test, a modified MD-Flexible that includes a parameter for innerCutoff is used to perform the simulation. The parameters of this experiment are obtained from the book cited here [GKZ07]. For the base version of the experiment, we start with 8x8x8 Particles, a cutoff distance set at 2.3, and an innerCutoff of 1.9. The container size is 9.2, and particle spacing is set to 1.15. The temperature starts at 3 and goes down by 0.0025 per iteration, with a target temperature of 0.02 after 10000 iterations. A second version of the experiment with 50x50x50 particles is done to see if more particles would give different results. The YAML file used as a base for both experiments can be found in 8.2.

The results in 5.1a show the masked and gather scatter approach having significant speedup compared to the auto vectorized approach, with the performance of the gather scatter being slightly better than the masked approach. In 5.1b the result is quite similar to 5.1a because the hitrate stays relatively low at around 4-3 percent in both scenarios as MD-Flexible automatically adjust the container size.



(a) 8x8x8



(b) 50x50x50

Figure 5.1.: Simulation of Supercooled Argon with different number of particles

5.1.2. Autopas Functor Benchmark

Autopasfunctorbenchmark[1] is a tool made to measure the performance of the functor by itself. This benchmarking tool also allows us to set a particular hitrate value to compare the performance at different hitrates. All tests were done with 1000 iterations and 2000 particles per cell and cutoff and innercutoff set to 3 and 1.5 respectively. The

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

speedup is how much faster the implementations are compared to the auto-vectorized version. Figure 5.2 shows that the masked approach is generally the best option across multiple hitrates, with the gather scatter being only slightly worse at very low hitrates and starts to fall off starting from a hitrate of 10 percent. Although the masked approach maintains a significant speedup of around 5 up until 100 percent hitrate, the gather scatter approach only shows minimal speedup at high hitrate levels. The other configurations also exhibit a similar pattern, but with the pair simulation, disabling newton3 optimization seems to have improved the speedup of the masked version. Still, it seems to have no effect on gather scatter.

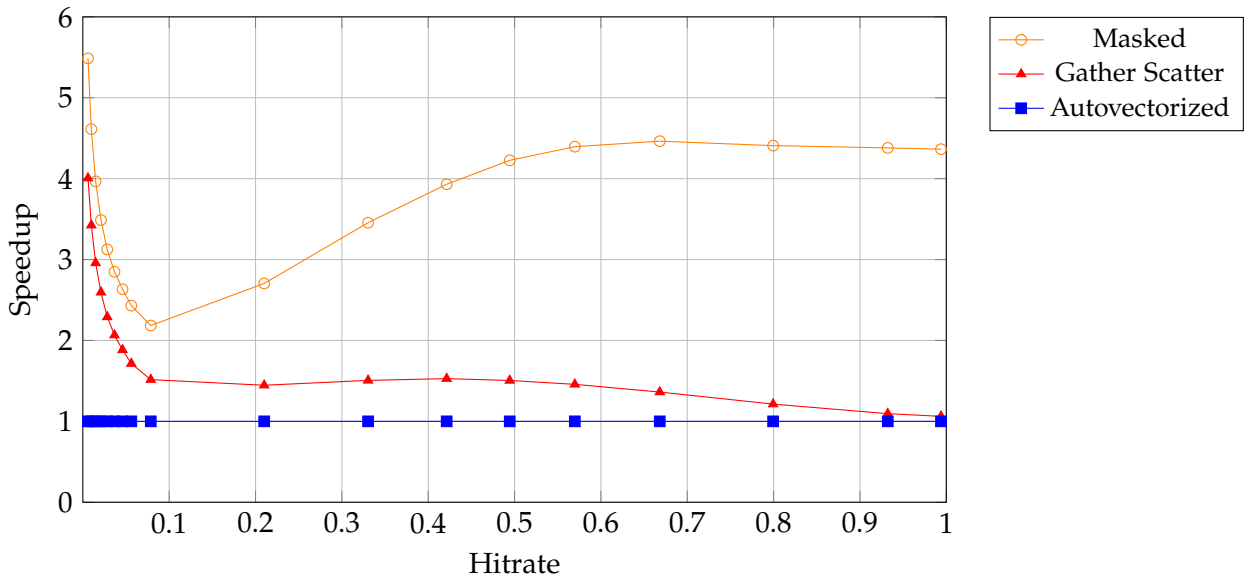


Figure 5.2.: Newton3, Single

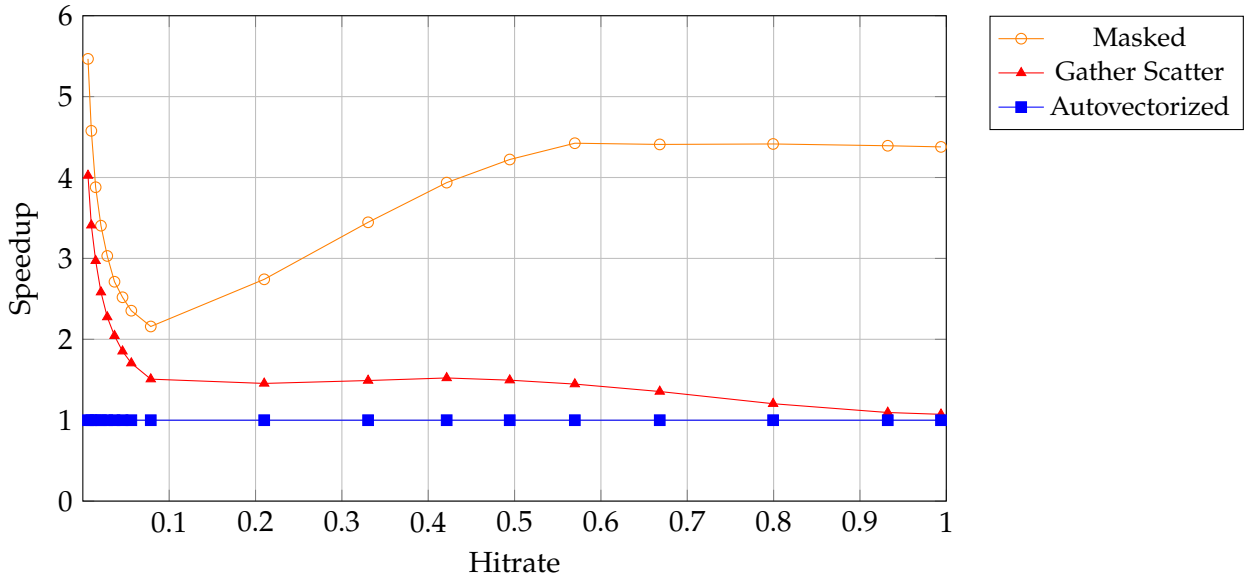


Figure 5.3.: No Newton3, Single

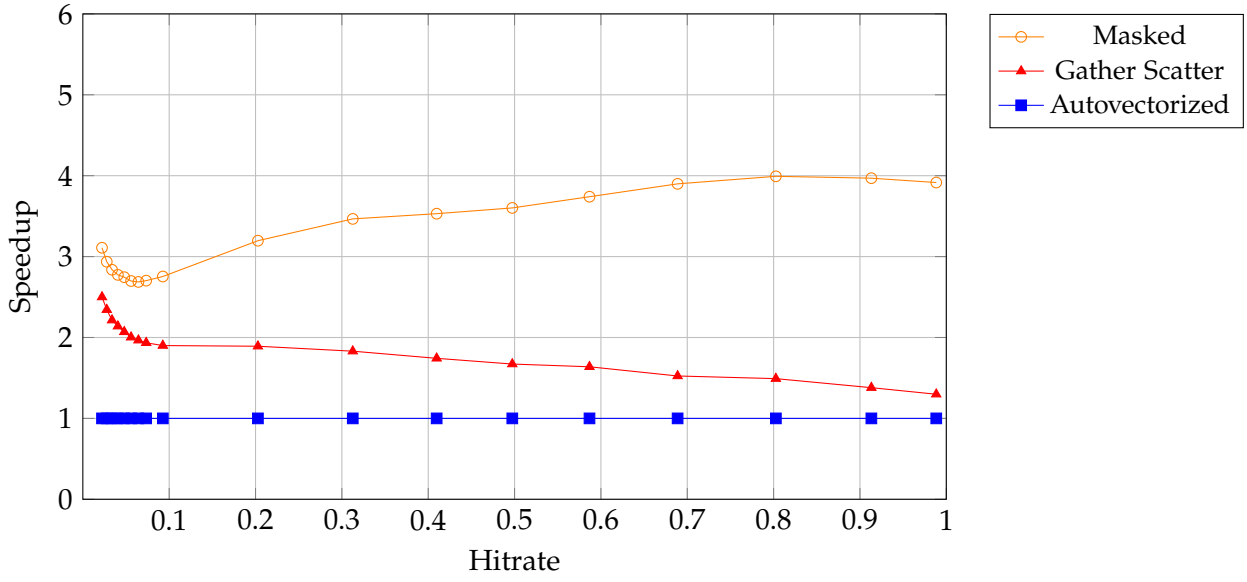


Figure 5.4.: Newton3, Pair

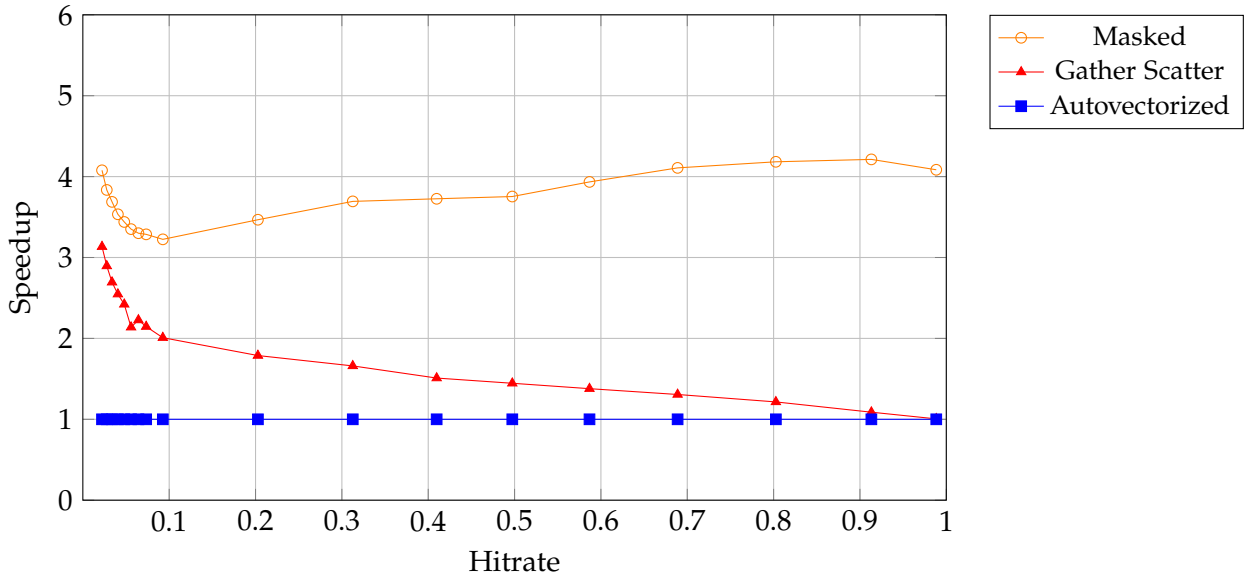


Figure 5.5.: No Newton3, Pair

6. Conclusion

In this Thesis, we have successfully implemented the Smoothed Lennard Jones potential in three different approaches: autovectorized, masked, and gather scatter using Google Highway. Although we hypothesize that the gather scatter approach would provide significant speedup due to the increased computational cost of the modified potential, the results have shown that the masked approach is still the best in this particular case. Although the gather scatter does fare comparatively well with the masked approach at very low hitrates (consistent with our results for the argon simulation 5.1); however, its speedup rapidly drops at hitrates above 10 percent.

The worse performance of the gather scatter implementation can be due to the unaligned nature of the memory access, causing slow memory access or the overhead cost of gathering indices for the kernel[Pen+13]. The usage of Google Highway, however, has been a success, as the gather scatter algorithm is heavily dependent on architecture-specific details, such as vector length, etc. Google Highway makes this generic form of the gather scatter algorithm easily adaptable for other architectures, as all three implementations are able to run on AVX2 and AVX512 with minimal changes.

7. Future Works

In the future, a comparison of how the the three different implementations would perform in other target architectures, for example, SVE might be interesting. These functors are written in Google Highway, and we expect it to run with minimal changes, as different architecture may provide better performance for gather scatter instructions than AVX512. Another thing that may be interesting to explore further is another implementation of the smoothed Lennard Jones potential such as the polynomial method used in LAMMPS[Tho+22], as there is no one standard version. This can quickly be done as the force calculation is isolated in the kernel and can be changed without much modification needed in the other parts of the program.

Modifying the Autopasfunctorbenchmark to distribute the particles with respect to the inner cutoff may also give us more insight into how good the gather scatter approach is compared with the masked when only a small amount of particles fall and need smoothing. Currently, we are unable to set the exact ratio of the smoothed and normal force calculation.

It might also be worthwhile to conduct a thorough profiling of the gather scatter implementation to determine which parts need to be optimized. Profiling tools such as Intel® VTune™ can provide vital information such as memory access and vectorization usage of the code[Int], giving us important data to improve performance.

Part IV.

Appendix

8. Appendix

1. **Any**: EMU128, SCALAR;
2. **Armv7+**: NEON_WITHOUT_AES , NEON , NEON_BF16 , SVE , SVE2 , SVE_256 , SVE2_128
3. **IBM Z**: Z14, Z15
4. **POWER**: PPC8 (v2.07), PPC9 (v3.0), PPC10
5. **RISC-V**: RVV (1.0)
6. **WebAssembly**: WASM, WASM_EMU256
7. **x86**: SSE2, SSSE3, SSE4, AVX2, AVX3, AVX3_DL, AVX3_ZEN4, AVX3_SPR

Figure 8.1.: Supported Platforms for Google Highway

```
container                         :  [LinkedCells]
verlet-rebuild-frequency          :  10
verlet-skin-radius-per-timestep   :  0.1
verlet-cluster-size               :  4
selector-strategy                 :  Fastest-Absolute-Value
data-layout                       :  [SoA]
traversal                         :  [lc_c08]
tuning-strategies                 :  []
tuning-interval                   :  2500
tuning-samples                    :  5
tuning-max-evidence               :  10
functor                           :  smohwygs #this is changed depending on which
    implementation is being used
newton3                           :  [enabled]
cutoff                            :  2.3
innerCutoff                       :  1.9
box-min                           :  [0, 0, 0]
box-max                           :  [9.2, 9.2, 9.2]
cell-size                         :  [1]
deltaT                            :  0.000
iterations                        :  10000
boundary-type                     :  [periodic, periodic, periodic]
globalForce                       :  [0, 0, 0]
Sites:
  0:
    epsilon                       :  1.
    sigma                         :  1.
    mass                          :  1.
Objects:
  CubeGrid:
    0:
      particles-per-dimension: [ 50, 50, 50 ]
      particle-spacing: 1.15
      bottomLeftCorner: [ 0.575, 0.575, 0.575 ]
      velocity: [ 0, 0, 0 ]
      particle-type-id: 0
thermostat:
  initialTemperature              :  3
  targetTemperature               :  0.02
  deltaTemperature                :  0.0025
  thermostatInterval              :  25
  addBrownianMotion               :  true
vtk-filename                      :  argoncoolOriginalLJ
vtk-write-frequency               :  1000
vtk-output-folder                 :  argoncoolOutput
no-flops                          :  false
no-end-config                     :  true
log-level                         :  info
```

Figure 8.2.: YAML Input file

List of Figures

List of Tables

Bibliography

[Col24] K. Cole. "Implementation and Vectorization of the Mie Potential in AutoPas." In: (2024).

[DM98] L. Dagum and R. Menon. "OpenMP: an industry standard API for shared-memory programming." In: *IEEE Computational Science and Engineering* 5.1 (Jan. 1998). Conference Name: IEEE Computational Science and Engineering, pp. 46–55. ISSN: 1558-190X. DOI: 10.1109/99.660313.

[Eng24] J. A. Englhauser. "Vectorization of Three-Body Potentials in AutoPas." In: (2024).

[Fly72] M. J. Flynn. "Some Computer Organizations and Their Effectiveness." In: *IEEE Transactions on Computers* C-21.9 (Sept. 1972). Conference Name: IEEE Transactions on Computers, pp. 948–960. ISSN: 1557-9956. DOI: 10.1109/TC.1972.5009071.

[Fou] F. S. Foundation. *Auto-vectorization in GCC - GNU Project*.

[GKZ07] M. Griebel, S. Knapek, and G. Zumbusch. *Numerical Simulation in Molecular Dynamics: Numerics, Algorithms, Parallelization, Applications*. en. Springer Science & Business Media, Aug. 2007. ISBN: 978-3-540-68095-6.

[Goo24] Google. *Google Highway*. Sept. 2024.

[Gra+19] F. A. Gratl, S. Seckler, N. Tchipev, H.-J. Bungartz, and P. Neumann. "AutoPas: Auto-Tuning for Particle Simulations." In: *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. May 2019, pp. 748–757. DOI: 10.1109/IPDPSW.2019.00125.

[Int] Intel. *Fix Performance Bottlenecks with Intel® VTune™ Profiler*.

[Jel23] J. Jelínek. *Vectorization optimization in GCC*. en. Section: Compilers. Dec. 2023.

[Len31] J. E. Lennard-Jones. "Cohesion." en. In: *Proceedings of the Physical Society* 43.5 (Sept. 1931), p. 461. ISSN: 0959-5309. DOI: 10.1088/0959-5309/43/5/301.

[New87] I. Newton. *Philosophiae naturalis principia mathematica*. Latin. Londini: Jussu Societatis Regiæ ac Typis Josephi Streater. Prostat apud plures Bibliopolas, 1687.

[Pen+13] J. Pennycook, C. Hughes, M. Smelyanskiy, and S. Jarvis. "Exploring SIMD for Molecular Dynamics, Using Intel® Xeon® Processors and Intel® Xeon Phi Coprocessors." In: May 2013, pp. 1085–1097. ISBN: 978-1-4673-6066-1. DOI: 10.1109/IPDPS.2013.44.

[Roc23] F. J. Rocke. *Evaluation of C++ SIMD Libraries*. 2023.

[Rut+17] G. Rutkai, M. Thol, R. Span, and J. Vrabec. "How well does the Lennard-Jones potential represent the thermodynamic properties of noble gases?" en. In: *Molecular Physics* 115.9-12 (June 2017), pp. 1104–1121. ISSN: 0026-8976, 1362-3028. DOI: 10.1080/00268976.2016.1246760.

[Tho+22] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. in 't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, R. Shan, M. J. Stevens, J. Tranchida, C. Trott, and S. J. Plimpton. "LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales." In: *Computer Physics Communications* 271 (Feb. 2022), p. 108171. ISSN: 0010-4655. DOI: 10.1016/j.cpc.2021.108171.

[WN19] 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). arXiv:1806.05713 [cs], pp. 1–7. ISSN: 00104655. DOI: 10.1016/j.cpc.2018.10.028.