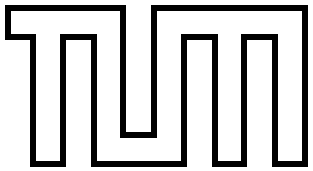


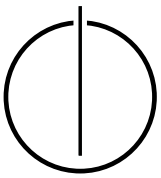
FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Guided Research in Informatics

Performance Analysis and Code Generation for the Force Calculation in Molecular Dynamics Simulations

Author:	Ludwig Gärtner
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Fabio Alexander Gratl, M.Sc.
Date:	October 18th 2019



Abstract

This guided research paper walks the reader through a domain specific language was created to generate a force calculator for the N-body simulation library AutoPas. It explains how to use the AutoPasDSL, describes the code generation process, and compares the performance of the generated force calculator to a pre written one.

The concept of code generation is very useful in this case, to make AutoPas more accessible to non-computer scientists. On the other hand it requires some improvements regarding the automatic improvement of user given code to be competitive, performance wise.

Contents

1 Introduction

AutoPas is an auto tuning library for N-body simulations. It only requires the user to provide the definitions for two things: a particle and implementation of a force calculation called a *functor*. Once these things are given to AutoPas, it automatically searches for the optimal way to perform a simulation with the given particle and functor. This is done at runtime and by adjusting many different options. These options, however, make the force calculation functor more and more complicated, mainly because it has to be able to parse its input in different forms. The result of this is a large functor file with mostly repetitive code. As AutoPas is aimed at scientific use by non-computer scientists, problems like this make it inaccessible to its target audience.

From this dilemma comes the motivation for this guided research. Namely to create a tool that lets a user of AutoPas provide only the essential information for how forces between particles have to be calculated, and automatically generate the necessary rest. For this purpose a domain specific language was created: the AutoPasDSL.

This paper presents the necessary theoretical background to build a domain specific language and an according compiler. Afterwards, it is described how this language and its compiler can be used together with AutoPas. To conclude the chapter, there is a comparison of performance between a handwritten functor and its generated equivalent.

2 Theoretical Background

This section covers the theoretical background that was required for the work of this paper. The first subsection is about the target project AutoPas. After that, there are two subsections about formal grammars and templating which are relevant for the code generator. The last subsection covers Intel Advanced Vector Extensions that were used throughout the entire work.

2.1 AutoPas

AutoPas is an automatic tuning library for N-body simulations. It therefore takes a user-defined particle class and functor and simulates interactions between particles while optimizing the process at runtime. This optimization is done by switching between particle containers, data layouts, traversal patterns for parallelization, and other options. The example that is most referenced here uses an implementation of the Lennard Jones 12-6 potential as the functor. [GST+19]

This subsection covers all parts of AutoPas that are relevant for the rest of this paper. Included are the Lennard Jones 12-6 potential, the particle containers, the data layout, and the functor itself. For a detailed description of the entire AutoPas project please refer to F. Gratl's paper "AutoPas: Auto-Tuning for particle simulations" [GST+19].

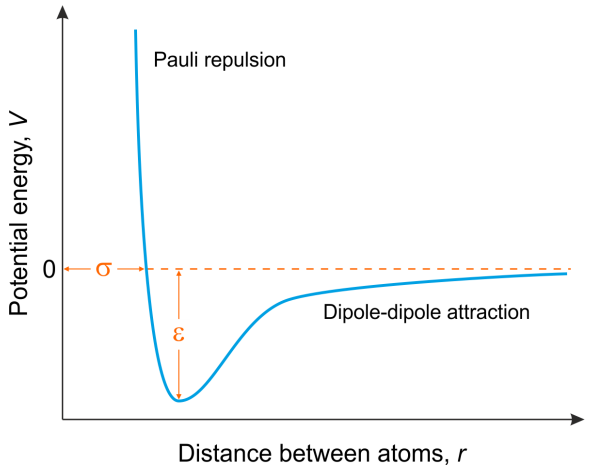


Figure 1: A graphical representation of the Lennard Jones 12-6 potential, and the impact of variables σ and ϵ. Taken from [Gen18]

2.1.1 The Lennard Jones 12-6 potential

The Lennard Jones 12-6 potential is a model for particle interactions that takes into account Van der Waals forces and the Pauli repulsion. [GST+19]

$$U(r_{ij}) = 4\epsilon \left(\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right) \tag{1}$$

In Equation 1, r_{ij} is the distance between particles i and j, and σ and ϵ are energy and size parameters of the employed model. See Figure 1 for a visualization of the Lennard Jones 12-6 potential in relation to particle distance.

The Lennard Jones 12-6 potential models short-range forces, meaning for larger distances between particles it quickly converges to 0. This convergence is exploited to reduce the computational complexity by implementing a cutoff distance r_c, beyond which the potential is set to 0. [GST+19]

2.1.2 Particle Containers

A particle container in this context is a datas tructure which stores all particles in the simulation domain. The goal is to derive information about the distance of two particles in the simulation domain from this structure. [GST+19]

The three available particle containers are *Direct Sum*, *Linked Cells*, and *Verlet Lists*. A graphical representation of all Containers can be seen in Figure 2.

Direct Sum is the simplest container. It calculates the forces of all particle pairs directly and has, therefore, the advantage of being the easiest to implement and having the least memory overhead of the three. The disadvantage of the *Direct Sum* container is its scaling in $O(n^2)$ with n being the number of particles, and a large amount of wasted distance calculations for particles further apart than the cutoff distance. [GST+19]

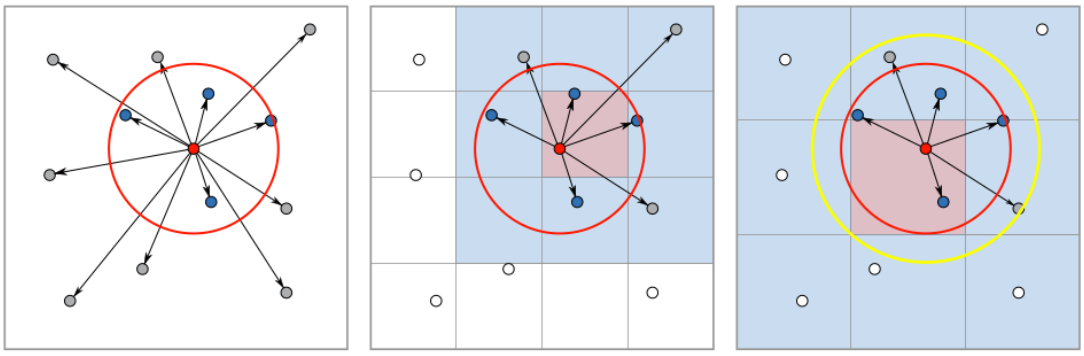


Figure 2: The different containers available for AutoPas simulations. From left to right: *Direct Sum*, *Linked Cells*, *Verlet Lists*. Taken from [GST+19]

Linked Cells utilizes the cutoff distance, and splits the simulation domain into cubic cells with the cutoff distance as length. Therefore only particles in neighboring cells are relevant for force calculations. This reduces the scaling of the simulation to $O(n)$ calculations. Additionally, the data locality gained by ordering and storing the particles in a cell together offers the possibility to optimize the calculation with vectorization. Nevertheless, only 15% of the particle pairs are within the cutoff distance as shown in Equation 2. [GST+19]

$$\frac{\textit{cutoff volume}}{\textit{search volume}} = \frac{\frac{4}{3} \cdot \pi \cdot r_c^3}{(3r_c)^3} \approx 0.155 \tag{2}$$

The last container option is *Verlet Lists*. Here a neighborlist (also called verlet list) is computed for every particle. This list stores all particles that are within the cutoff distance and thereby reduces the number of unnecessary distance calculations in the functor. However, building these verlet lists requires the distances being calculated between all particles. The complexity of building neighbor lists can be reduced by additionally using a Linked Cells container, to only consider particles in neighboring cells. [GST+19]

As most simulations are for moving particles, it is advisable to extend the distance relevant for building the verlet lists to be larger than the cutoff distance, such that the neighborlists have to be rebuilt only every m steps. This buffer is also called skin and is illustrated by a yellow circle in Figure 2. With the verlet distance being 1.2 times the cutoff distance the amount of unnecessary distance calculations is around 42% due to particles being in the skin but not within the cutoff distance. [GST+19]

Another downside of this container option is the additional memory requirement for storing the neighbor lists. [GST+19]

2.1.3 Data layouts

The two different data layouts are *Array of Structures (AoS)* and *Structure of Arrays (SoA)*. For *AoS* each particle is stored as a struct containing all its parameters. With all the data

of one particle being stored in successive memory, moving particles between containers or nodes is very easy. [GST+19]

For the *SoA* layout, there is one array for every parameter. For any given particle i its parameters are stored in the corresponding parameter array at index i. The data locality here enables simple loads of successive memory cells for vectorization. In the following SoA is used to describe either the data layout or an instance of said layout. [GST+19]

2.1.4 Functor

The term *Functor* as used repeatedly in this paper describes the implementation of the force calculator. Its objectives are taking two particles, calculating the force they are exerting on each other and writing the result back into the respective parameter of the particles, and accumulating the forces into a global variable. Due to the different optimization methods that AutoPas uses the functor has to provide four different modes: *AoS*, *SoA one cell*, *SoA two cells*, and *Verlet SoA*.

AoS This is the most simple of the four modes. It just takes two particles as input, performs the calculations, and writes the output back into the particle parameters.

SoA one cell This mode is used in two scenarios. In case Direct Sum is the current container all forces are calculated here. The second case is for calculating the pairwise forces within one cell when using the Linked Cells container. The input is one SoA containing all particles. The forces of all particle pairs (i, j) with $i \neq j$ are calculated here.

SoA two cells To calculate forces between particles in two different cells, this mode is used. It is either used in Linked Cells or when calculating forces between two particle domains in Direct Sum. The inputs are two SoAs of two separate cells. Here the potential between particle pairs (i, j) with $i \in SoA_1, j \in SoA_2$ is calculated.

Verlet For this mode the inputs are one SoA containing all particles, from and to indices, and a list of verlet lists for every particle. For force calculations here, we iterate over all particles with index $from < i < to$ in the SoA, and perform the calculations with all particles in the according verlet list.

2.2 Domain Specific Language

A Domain Specific Language (DSL) is a high level programming language that offers an additional layer of abstraction for programmers. It is created for a specific use case or application domain.

A DSL's goal is the separation of concerns: the end-user only cares about his solution to a problem, while the programmer implements the automatic optimizations and works on the target platform. Therefore a compiler translates the abstract specification of a solution given by the user into a specific and optimized low-level implementation.

To create a DSL one has to define its formal grammar (see Subsection 2.3) and then build such a compiler.

2.3 Formal Grammars for programming languages

In this subsection, the theoretical definitions of formal grammars that define programming languages are covered. Additionally, a specific notation for such grammars in the Extended Backus Naur Form is explained, and the python tool that was used to define the AutoPasDSL grammar, Lark, is described.

2.3.1 Formal Grammar and Formal Language

A *formal language* is defined by a finite set of terminal symbols Σ and a set of finite sequences/words of these terminals. The empty sequence is written as $\wedge$. [Yeh76]

To define a formal language one can either give its entire set of words or use a *formal grammar* that describes which sequences are contained in the language.

A *formal grammar* is a 4-tuple (N, Σ, P, S):

- N: is the finite set of nonterminals which are used as variables in the production rules P.
- Σ: is the finite set of terminal symbols.
- P: is the finite set of production rules. A production rule is written as (α, β) with $\alpha, \beta \in (N \cup \Sigma)^*$ and means β can be derived from α.
- S: is the starting symbol $S \in N$. It is the root of every derivation of words in the final language.

The formal language that is generated by this grammar contains every sequence of terminal symbols that can be produced by applying the production rules P. [Yeh76]

2.3.2 Context free grammar

A programming language is just another formal language with a formal grammar that defines which code is legal. Most programming languages, however, are produced by a more specific kind of grammar: a *context free grammar*. For a context free grammar, the production rules are limited to (A, α) with $A \in N$ and $\alpha \in (N \cup \Sigma)*$. [Yeh76]

So the difference between a standard formal grammar and a *context free grammar* is, that for the latter only a singular non-terminal symbol makes up the left side of a production rule.

2.3.3 Derivation tree

Derivation trees are structures that describe how words from a formal language are derived by using the production rules that make up the language's grammar. Note that only words from languages that are defined by context free grammars can be derived in this way.

A derivation tree is defined as a tuple (T, y) with a rooted tree $T = (V, R)$, and a mapping y of Nodes in the tree to a sequence of terminals and nonterminals $y : V \mapsto N \cup \Sigma \cup \wedge$. The construction rules for a derivation tree are:

- $y(v_0) = S$ if v_0 is the root of the tree

(Digit, 0) (Digit, 1) ⋮ (Digit, 9)	Digit = ”0”|”1”|”2”|”3”|”4”|”5”|”6”|”7”|”8”|”9”
(Number, Digit) (Number, Digit Number)	Number = Digit{Digit}
(FPNumber, Number) (FPNumber, Number ’,’ Number)	FPNumber = Number[”,” Number]

Table 1: See the difference in notation when describing production rules for a context free grammar normally (left) or in EBNF (right). The language that will be produced by these grammars are all floating point numbers. Besides the production rules, the grammars are defined with $N = \{\text{Digit, Number, FPNumber}\}$, $\Sigma = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 0, ','\}$, and $S = \text{FPNumber}$.

- if for a vertex v exists an edge $(v, u) \in R$ then $y(v) \in N$
- if a vertex v has the immediate successors v_1, v_2, v_3 then there is a production rule $y(v) = y(v_1)y(v_2)y(v_3)$ in P

2.3.4 Extended-Backus-Naur-Form

One notation to define the production rules of a context free grammar, is the Extended-Backus-Naur-Form (EBNF). [MR]

- terminals are indicated by surrounding ”-marks
- nonterminals are written plainly
- a production rule (A, α) is now written as $A = \alpha$

Additionally, there are some functionallity changes for the EBNF compared to the normal context free grammar:

- In the production rule: A = ”a”[”b”] the expression within the ’[ ]’ is optional.
- In the production rule: A = ”a”{”b”} the expression within the ’{ }’ can be repeated 0 or more times. In a formal grammar repetition can only be achieved via recursion.
- If multiple production rules exist for A, they can be summarized with a ’|’: A = ”a” | ”b”

The EBNF is clear and concise in describing context free grammars. For a comparison between the definition of production rules in a standard context free grammar as well as in EBNF notation see Table 1.

2.3.5 Lark

The python library lark[1] was used as a syntax parser for the AutoPasDSL. Lark lets one define a context free grammar with a slightly changed and extended EBNF notation. Additionally, it generates a derivation or syntax tree from inserted code in the AutoPasDSL which can then be used for code generation purposes.

2.4 Templating for Code Generation

Templates in the context of this work are predefined structures and snippets of code. They consist of already final code which will be unchanged during the code generation part, and placeholders which are replaced with content. The python template engine that was used in this project is called Jinja[2].

Jinja produces its rendering output by taking a python dictionary and a template and then filling all placeholders defined in the template according to data from the dictionary. It also allows for nested dictionaries to be used as input. That means that an entry in the top-level dictionary can have another dictionary as value. Its two main functionalities that were used for AutoPasDSL templates are:

- simple placeholders {{ variable }}: This snippet is replaced by taking the value of the key ”variable” from the top-level dictionary.
- for loop placeholders {% **for** var **in** variables %} {{ var.name }} {% **endfor** %}: To replace this snippet, the dictionary is searched for the key ”variables” which value is expected to hold a list (of dictionaries). Afterwards, the simple placeholder within the for statement is replaced by looking in each of those dictionaries for the value of the key ”name”.

2.5 Intel Advanced Vector Extensions

The Intel Advanced Vector Extensions[3] (AVX) extends the standard instruction set for Intel architecture processors with single instruction multiple data (SIMD) instructions. The concept of SIMD is, to load memory aligned data into vector registers, perform one instruction on the entire vector(s), and then store the result into aligned memory cells.

As the version of AVX that was used for this work does not natively support gathering data from non-aligned memory, one has to either perform this gathering step manually or use an SoA data layout (see subsubsection 2.1.3) to begin with. Note that newer extensions like AVX-512 do support gathering data from non-aligned memory. For a visualized example of a SIMD addition see Figure 3.

3 Code Generator for AutoPas Functor

This section will cover everything about the code generator for the AutoPas Functor. Included are sections about the syntax/formal grammar of the AutoPasDSL and a subsection that

[1] `https://github.com/lark-parser/lark`
[2] `https://github.com/pallets/jinja/`
[3] `https://software.intel.com/sites/landingpage/IntrinsicsGuide`

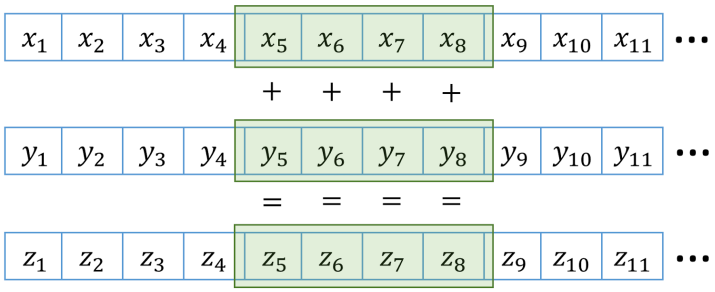


Figure 3: A visualized example of adding two four-value vectores. Taken from [Fra15]

covers the flow of information between AutoPas and the Functor. These two describe how one has to write a new functor in the AutoPasDSL and then how to use it coupled with AutoPas.

Following is a step-by-step description of the code generation process. Lastly there is also a performance comparison of the Lennard Jones 12-6 (in the following section abbreviated by LJ) functor that was written by hand and a generated one.

For this code generator a DSL, named AutoPasDSL, has been created. Its purpose is to define the pairwise force calculation between particles on a highly abstract level. A compiler then translates this definition to C++ code which fits all requirements demanded by the AutoPas library.

3.1 AutoPasDSL Syntax

This subsection covers all necessary information to write a Functor in the AutoPasDSL including the basic structure of code files and syntax. For the entire grammar of the AutoPasDSL refer to Figure 16.

When writing a functor in the AutoPasDSL, one has to define two major components: the static values and the functor kernel.

3.1.1 Static Values

The static values are predefined and final, meaning they cannot be changed during force calculation. They depend on the force for which the functor is written. For example, for the Lennard Jones 12-6 potential (see subsubsection 2.1.1) the static values include the cutoff radius, ϵ, and σ. This is how the statics are defined for the LJ functor:

```
STATIC ( cutoff, epsilon, sigma, shift )
```

3.1.2 Functor Kernel

The functor kernel itself consists of three parts: the actual name of the functor, its parameters, and its function body.

The name of the functor influences the name of the generated functor file as well as the class name of the functor. So the three lines

`vec result = 1.5;`	`double resultX = 1.5;`
	`double resultY = 1.5;`
	`double resultZ = 1.5;`

Table 2: This is how the initiation of a vectorized variable (left) is translated to C++ (right).

```
LJFunctor ( vec pos_i, vec pos_j,
            vec &force_i, vec &force_j,
            vec &virial, &upot ) {
```

in the functor file create a file LJFunctor.h that contains a class called LJFunctor.

Within the parentheses of the previous code snippet are all parameters of the functor, including the parameters of particles as well as the accumulators of global forces.

The kernel body is the part where the actual force calculation code is written. It consists of *formulas* and *function calls* performed on *variables*.

Variables Variables in the AutoPasDSL do not have to be declared with a traditional type from for example C++ as they all have the same type. For the SoA modes of the functor, this type is an Intel AVX `__m256d` (256 bits filled with four 64 bit floating point number). For the AoS mode, this type is a normal floating point number which size depends on a definition in the particle class. The following line creates a variable and initiates it with a value:

```
foo = 3.5;
```

All user-defined variables are stored as variable objects in a list and each time a variable appears on the left side of an equation it is either found in this list or newly created and added to the list.

Despite all variables having the same type, there is the option to create a vectorized variable with the *vec* keyword. The purpose of this is to reduce the amount of code one has to write when most of the operations are to be executed on variables of all spatial dimensions. This is implemented by substituting a vectorized variable or line of code by its spatial counterparts. An example of this can be seen in Table 2. Please do not confuse this vectorization with AVX vectors which are talked about throughout this paper. To avoid confusion this technique is from now on referred to as spatial vectorization.

When a vectorized variable is created, this information is stored in the matching variable object.

Formulas Formulas are mathematical expressions. They syntactically consist of a result variable on the left and mathematical operations on the right side of a '=' sign.

Currently supported operations are additions, subtractions, multiplications and divisions. Additionally, any group of operations within one formula can be prioritized regarding the order of evaluation by parentheses.

`vec result = force * dir;`	`double resultX = force * dirX;`
	`double resultY = force * dirY;`
	`double resultZ = force * dirZ;`

Table 3: This is how vectorized formulas are evaluated. In this form, `result` is created as vectorized variable, while `dir` already exists as vectorized and `force` as singular variable.

If the result variable of a formula is vectorized, the entire formula is vectorized. This means that the formula is evaluated d times, with d being the number of dimensions. Vectorized variables on the right side of the equation are substituted by their components, while singular variables are used in all evaluations. For an example of this see Table 3.

Function Calls There are currently two predefined functions in the AutoPasDSL: `mask()` and `squared_distance()`.

The `squared_distance()` function takes either two vectorized variables or six singular variables as an input. For this function it is assumed that these describe the positions of two particles. With these arguments, the squared distance of the two particles is computed, and then stored in the variable `squared_distance` for later uses.

This function serves additional purposes, depending on whether it is called for an AVX or No-AVX backend. In the case of a No-AVX backend, this method triggers a `return` statement if the two compared particles are not within the cutoff distance. When working with AVX vectorized values a distance mask is created. For each particle pair that is within the cutoff distance their matching vector slot is filled with `1`s, `0`s otherwise. This mask is later used in the `mask()` function. The reasoning behind calculating the squared distance here is performance. Evaluating the square root in Equation 3

$$\texttt{cutoff\_distance} \geq \sqrt{\texttt{posX}^2 + \texttt{posY}^2 + \texttt{posZ}^2} \tag{3}$$

is much harder than evaluating the powers that can be expressed by multiplications in Equation 4

$$\texttt{cutoff\_distance}^2 \geq \texttt{posX}^2 + \texttt{posY}^2 + \texttt{posZ}^2 \tag{4}$$

even though the two expressions produce very similar results.

The `mask()` function is ignored when used from a No-AVX background. It is only useful when working with AVX vectors, due to them always packing groups of parameters into one vector. The problem here is whenever a number of particles that is not divisible by the AVX vector size is handled, there are calculations performed on unintentionally loaded data. Those results are then written back into the memory which might result in bad behavior of the code. Additionally, the unintentionally calculated forces are accumulated in the globals. To solve this problem the `mask()` function ensures that this unintentionally calculated data is filtered out of the passed variables at the time of calling. Furthermore when the `mask()` function is called after the `squared_distance()` function the previously computed distance mask is applied to all passed variables.

It is advised to always call this function before working on any output variables.

```
STATIC ( cutoff, epsilon, sigma, shift )

explicit LJFunctor( double cutoff, double epsilon,
                    double sigma, double shift, [...])

ljFunctorAVX(_cutoff, _epsilon, _sigma, 0.0);
```

Figure 4: This is how the static values are defined in AutoPasDSL (top), then translated into arguments of the constructor of the functor (middle), and finally how the values of the statics are provided (bottom). Note that the argument(s) in the square brackets are independent of any code in the AutoPasDSL and therefore blanked out here for simplicity's sake.

3.2 The Interface between AutoPas and the Functor

This subsection covers how information is passed between the AutoPas library and the functor. There are three separate parts to this flow: the static values, particle parameters, and globally accumulated forces.

3.2.1 Static Values

At the time of writing, the static values are values that are the same for the entire simulation. Refer to subsubsection 3.1.1 for an example of which static values are used in the Lennard Jones functor.

In the AutoPasDSL the static values are defined in a specially created line:

```
STATIC( static1, static2, ... )
```

On the C++ side, these statics are implemented as attributes of the functor class. Therefore they have to be provided as parameters when constructing a new instance of the functor. For an example of how this is written, translated, and then used please refer to Figure 4.

Note that this passing of data is highly reliant on keeping a consistent order of arguments. There is no automatic matching of passed values, but the static value that is written in first position in the AutoPasDSL code has to be provided first when calling the constructor of the functor.

3.2.2 Particle Parameters

Particle parameters are provided in a syntax that is similar to function parameters in C++. The following line is a snippet from the code in Appendix Figure 17.

```
LJFunctor ( vec pos_i, vec pos_j,
            vec &force_i, vec &force_j,
            vec &virial, &upot ) {
```

Particle parameter in AutoPasDSL (vectorized left, singular right):

```
vec pos_i   or  posX_i,
                posY_i,
                posZ_i
```

Particle parameter as searched for in particle object:

```
double posX;
double posY;
double posZ;
```

Figure 5: A particle parameter how it is defined in the AutoPasDSL and by which name its values are extracted from the particle object.

Particle parameters can only be defined in this list. When defining particle parameters there are a few rules that have to be followed:

- First of all, a particle parameter has to end its name with either the index `_i` or `_j` to indicate its origin particle.
- Secondly, for every parameter of particle i there has to be a matching parameter of particle j.
- And lastly, these parameters have to be named exactly as they are named in the currently used particle class. Please note that the index is disregarded for the purpose of finding the parameters in the particle class. Additionally, one has to be careful when using spatially vectorized parameters here. Refer to Figure 5 for a visualized clarification of this process.

Results can only be extracted from the functor by using referenced parameters. The concept here is to provide a function parameter such that the called function changes the value of said parameter. In the AutoPasDSL the `&` sign indicates whether a parameter has to be passed by reference or by value if no `&` sign is given. Additionally, this marks parameters to be either input only (*needed*), or in- and output (*computed*). This distinction is made for the data gathering and result storing in the code generation step (see Section 3.3.1).

When applying this knowledge one can see that the parameters `pos_i` and `pos_j` are passed by value in the above code snippet and cannot be changed. The parameters `force_i` and `force_j`, however, are passed by reference, and can be changed in the functor body.

Note that the two ways of defining particle parameters in the AutoPasDSL as shown in Figure 5 produce equal results when translated, but only the vectorized one allows to use the argument in vectorized formulas when used in the functor body. This is due to how parameters are stored in the list of variables mentioned in Section 3.1.2.

3.2.3 Globally Accumulated Forces

Globally accumulated forces are an optional addition to the functor. They are also defined in the previously cited line of code from the LJFunctor:

```
        upot  double getUpot();
vec virial  std::array<double, 3> getVirial();
```

Figure 6: These are two example declarations of getter methods for globally accumulated forces.

```
LJFunctor ( vec pos_i, vec pos_j,
            vec &force_i, vec &force_j,
            vec &virial, &upot ) {
```

Every parameter that does not end with an index is automatically assumed to be a global force variable. The only rule for these global force variables is that they have to be passed by reference, as they are typically changed during the calculation phases.

When translated into the functor class, these globals are stored as private attributes of the functor. They can, after the simulation has finished, be accessed via getter methods. The naming of these getter methods is essentially `get[globalName.toUpper]()`. For an example of this see Figure 6.

A getter method of a singular global variable returns a normal `double`. On the contrary, a getter method for a vectorized global variable returns a C++ `std::array` containing d elements, with d being the number of spatial dimensions.

3.3 Code Generation

This section covers the process of generating the final functor class file. In particular, it is about the generation of the structure of the functor via templating, and the visitor that works through the derivation tree of the provided AutoPasDSL code.

3.3.1 Generating the Structure of a Functor

The structure of the AutoPas functor is predefined by a parent functor class, from which it has to inherit all signatures of methods that are called by AutoPas. To make sure that the user cannot accidentally change those method signatures they are hardcoded and hidden within the template. For a brief introduction about templates please refer to Subsection 2.4.

This functor template consists of prewritten code for content that should be independend of user-written AutoPasDSL code. Examples of that are:

- the class definition of the functor class.
- necessary imports.
- code snippets that check whether cells or particles are owned by the current node, or are contained in ghost layers.
- a data structure that handles a race condition free collection of global forces.

The process of code generation starts immediately after the user-written AutoPasDSL code file has been interpreted and parsed into a derivation tree by Lark. For a visualization of a derivation tree that is formed by AutoPasDSL code, please refer to Figure 18.

Fetching metadata from the provided AutoPasDSL file Metadata, in this case, refers to the following things:

- name of the functor
- static values
- particle parameters
- global variables

To fetch this data the derivation tree is searched for the subtrees that hold this information, namely the `paramlist` subtrees of both the `statics` and the `kernel` nodes.

All of this metadata is then prepared and stored in python dictionaries for the final rendering step, as well as the previously mentioned parameter object list (see Section 3.1.2).

Additionally, in this step, it is checked whether all parameters follow all rules that are defined in subsubsection 3.2.2.

Generating Kernel Code Kernel code in this context means the main force calculation part of the functor. This piece of code gets called after the functor has fetched all required data for the force calculation from whichever data structure it received as input.

To generate the kernel code for the functor the variable list and the `body` subtree of the derivation tree are passed to the `kernel_code_generator`. For a detailed explanation of how the kernel code is generated by traversing the derivation tree please refer to subsubsection 3.3.2.

The generated kernel code is also stored in the python dictionary that is used for the final rendering.

Final Rendering Step The final rendering step takes the python dictionary that has been built over the previous steps. At this point, it contains information about the static values, particle parameters and global variables that serve as input for the functor, the generated kernel code, and the code for the getter methods of each global variable.

Kernel code and getter methods are simply plugged into the template, as they are fully functioning code on their own.

The largest amount of work which is done here is to generate all data-gathering operations for the particle parameters. Please refer to Figure 7 for a visual example of how particle data is gathered. This step is done in each function for each time particle data has to be gathered for calculations or results have to be stored after calculations.

3.3.2 Visitor for Kernel Code Generation

This subsubsection covers how the `body` part of the derivation tree is traversed, and parsed into the force calculation kernel of the functor. It is advised to refer to the grammar of AutoPasDSL (see Figure 16) as well as the example derivation tree from Figure 18.

As displayed in the following excerpt of the AutoPasDSL grammar a line in the kernel body can either be a formula or a function (call).

```
// Particle &i;
{% for param in parameters.particle_i.needed %}
auto {{ param.particleAttributeIndexed }} = i.template get
      <Particle::AttributeNames::{{param.particleAttribute}}>();
{%endfor%}

auto posX_i = i.template get<Particle::AttributeNames::posX>();
auto posY_i = i.template get<Particle::AttributeNames::posY>();
auto posZ_i = i.template get<Particle::AttributeNames::posZ>();
```

Figure 7: The gathering of all *needed* parameters of a particle in the Lennard Jones functor. The above lines show a snippet from the functor template, the below lines shows the rendered version.

```
body : ([function | formula] ";")+
formula : param "=" a_expr
function : CNAME "(" param* ")"
```

Formulas A formula consists of a result side (left) and an expression side (right). To start the generation of the formula, the variable is looked up in the list which stores all previously defined variables. If it is not found the variable has to be declared in the generated C++ code and it has to be created for the variable list, as it now is defined.

The expression side of a formula is again represented as a subtree of the derivation tree. In the grammar, it is defined as:

```
a_expr : m_exp | a_expr PLUS m_exp | a_expr MINUS m_expr
m_expr : u_exp | m_expr MUL u_exp | m_expr DIV u_expr
u_expr : powe | "-" u_exp | "+" u_expr
power : primary [POW u_expr]
primary : number | param | "(" a_expr ")"
```

This nesting of operations is necessary for keeping the right order of evaluation: parentheses > powers > multiplications > additions.

On the downside, this creates very deep derivation trees for even the simplest of formulas, as can be seen in Figure 18.

To parse this derivation tree a *visitor* is implemented which traverses the tree similar to a depth-first search algorithm. Evaluating an expression is done by recursively evaluating all children of the expression first, and then the expression itself.

The visitor has the following parameters:

- a token generator, which generates previously unused tokens for temporary results.
- a `result_variable` which is expected to hold the result of the current expression after evaluation.
- the subtree of the expression that is currently evaluated.

- the list of currently defined variables.
- a string which is the currently generated `kernel_code`.

There are currently only two possible numbers of children for each node: one or three. The complexity of this visitor is reduced by skipping the evaluation of a node if it only has one child, with the exception being if the node is a `param` or `number`. If a node has only one child, and this node is a `param` or `number` the code generator produces the following line of code:

```
result_variable = child.value();
```

and adds it to the `kernel_code`.

If a node has three children, these children are: an expression, one operator, and another expression, in exactly that order. In this case, the visitor goes through the following steps:

- generate two temporary result variables and add a line of code that declares both to the `kernel_code`
- evaluate both child expressions by giving each one of those result variables, their respective subtree of the expression, and all other parameters.
- generate a line of code: `result_variable = tmp_variable1 operator tmp_variable2;`
- add that line of code to the `kernel_code`

Function Calls Function calls are easier to parse than formulas. The only two functions that are currently supported are: `squared_distance()` and `mask()`.

For the `squared_distance` function it is checked whether the right number of parameters is provided. The right number, in this case, is twice the number of spatial dimensions or two spatially vectorized variables. Afterwards, there is just a call to the in the template already existing function `squared_distance()` added to the `kernel_code`.

The `mask()` function can be called for any number of variables. For each variable `var` the following line of code:

```
var = mask ? var : 0;
```

is added to the `kernel_code`. The mask in this example is set as previously described in Section 3.1.2.

3.4 Comparison of Performance

This subsection covers the comparison of performance between a handwritten and optimized implementation of the Lennard Jones 12-6 functor, and a generated one. Therefore there is one detailed comparison for each of the three container options explained in subsubsection 2.1.2.

The paragraphs of this subsections represent a comparison of performance for each of the previously mentioned functor modes (see subsubsection 2.1.4).

For the overall performance of the handwritten functor please refer to Figure 19, for the generated functor Figure 20.

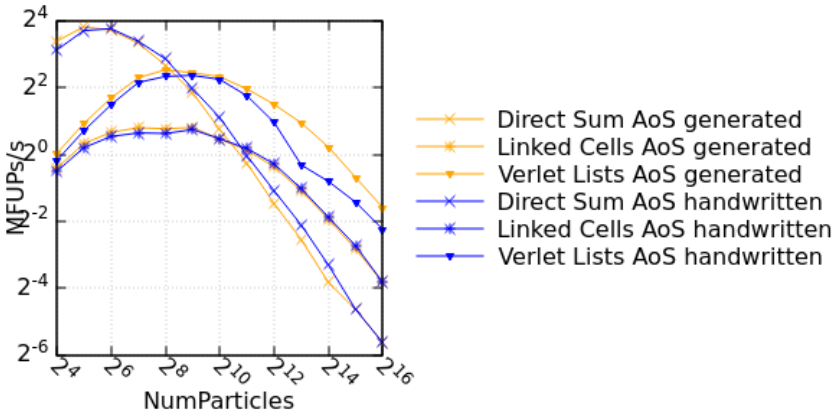


Figure 8: The performance comparison of a handwritten and a generated LJ functor when using the AoS data layout

AoS This functor mode is used with each container if the current data layout is AoS.

Figure 8 shows the performance of both functors with three different containers. The first thing that strikes the mind is the very close performance of the generated and the handwritten functor when using the AoS data layout. Even without the optimizations that were implemented by hand, the generated functor performs nearly as good as the handwritten one. This essentially shows the optimization powers of modern C++ compilers.

SoA one cell This functor mode is used mainly in the Direct Sums container and in the Linked Cells container when comparing particles within one cell. Due to the performance of the Linked Cells container being dominated by the comparison of two cells, this paragraph focuses on the performance of the Direct Sums container when used with an SoA data layout.

Figure 9 shows the visualization of the performance for the two functors when using this configuration.

Here the performance difference of hand-optimized and generated code becomes obvious for the first time. This is most likely due to the compiler being not very proficient with optimizing code that makes heavy use of Intel AVX intrinsics.

SoA two cells This mode is used in the Linked Cells container when calculating forces between particles in two different cells. Please keep in mind that the performance graph here does not only represent the performance of this mode but also the SoA one cell mode. However, as every cell has 26 neighboring cells, the force calculations within one cell can be disregarded in this analysis.

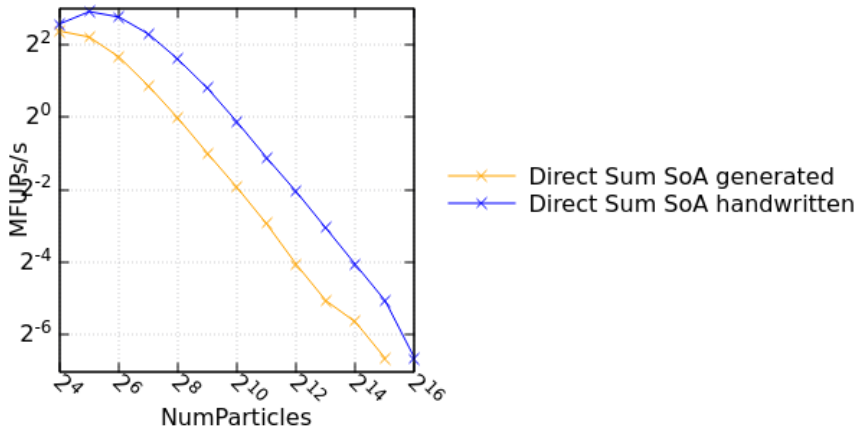


Figure 9: The performance copmarison of a handwritten and a generated LJ functor when using the SoA one cell mode.

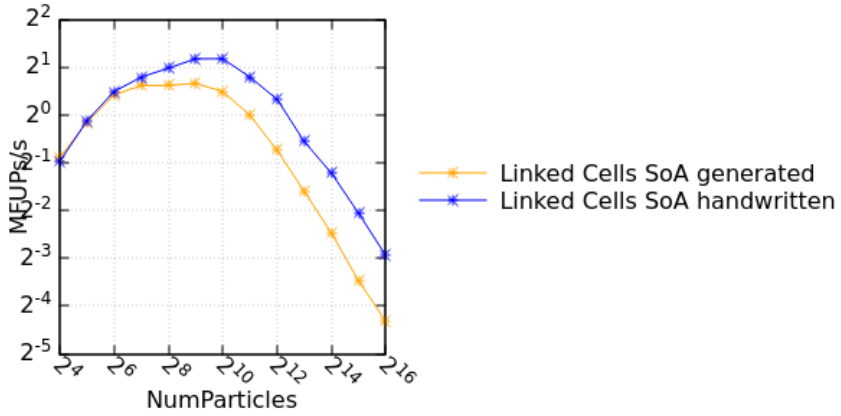


Figure 10: The performance comparison of a handwritten and a generated LJ functor when using the SoA two cells mode.

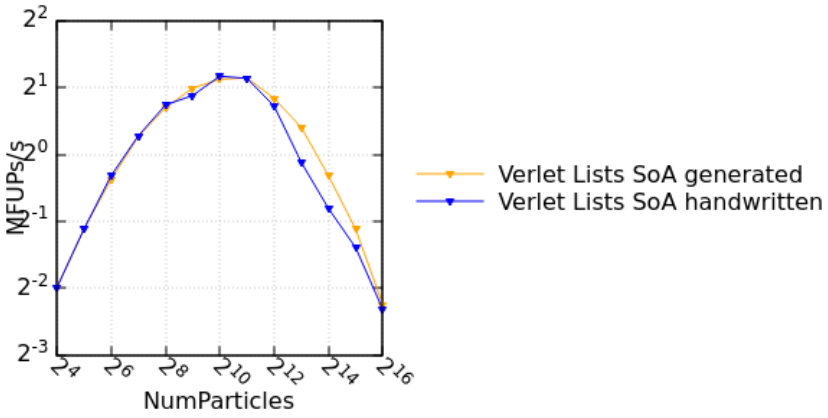


Figure 11: The performance comparison of a handwritten and a generated LJ functor when using the Verlet mode.

The difference in performance is slightly less here, compared to the SoA one cell mode, especially when working with fewer particles. When scaling the simulation up to a higher particle count, this difference becomes bigger again. The explanation is the same as for the SoA one cell mode: it is less efficient code, and the compiler cannot make up for it via optimizations.

Verlet The last mode of the functor is the Verlet mode. It is applied when using Verlet lists as the container and having the data in an SoA layout. Please note that this comparison is less meaningful than the three before, as the original code for Verlet lists is automatically vectorized, while the generated code uses Intel AVX intrinsics. This also makes the analysis of the comparison a little harder, as many more factors play a role in this.

As can be seen in Figure 11 this is the first time the generated functor is faster than the originally handcrafted one. Even though the code is less efficient due to unnecessarily repeated calculations, the AVX vectorized code is superior to the auto-vectorized implementation.

4 Optimizations for AutoPas Lennard-Jones Functor

The original implementation of the Lennard-Jones functor for AutoPas has a data access pattern which seems inefficient. This pattern is explained first, before presenting the idea of an improved version in the second subsection of this section. Afterwards, the performance of the two implementations is compared.

The focus of this section is solely the SoA two cells mode of the Functor.

```
input: SoA_i, SoA_j
  for particle_m in SoA_i
    vector_m = broadcast(particle_m)
    for [particle_n .. particle_n+3] in SoA_j
      vector_n = load([particle_n .. particle_n+3])
      calculate_forces(vector_m, vector_n)
```

Figure 12: The pseudocode for how force calculation for particle pairs is done in the original implementation of the SoA two cells functor mode.

4.1 Original Implementation

A visualized example of how the original implementation works can be found in Figure 21.

When calculating the pairwise forces between two cells/groups of particles, one has to calculate the forces between every particle from cell i with every particle from cell j.

This process is vectorized when working with an SoA data layout, meaning particles are grouped and loaded into vectors. To simplify this process the current implementation broadcasts the information of one particle m from cell i into the entire vector and then loads particles n to $n+3$ from cell j into the second vector. After having calculated all forces between particle m and the particles n to $n+3$ are calculated, and the calculation can be resumed for the next group of particles. For a pseudocode implementation of this please refer to Figure 12.

This way the number of memory accesses for particles is:

- every particle from cell i has to be loaded once
- every particle from cell j has to be loaded once for every particle in cell i

Therefore the number of memory accesses is

$$sizeof(\text{cell i}) + sizeof(\text{cell i}) * sizeof(\text{cell j}) \tag{5}$$

4.2 Optimized Implementation

The idea behind this improvement was to reduce the number of memory accesses. This is achieved by now loading four particles m to $m+3$ from cell i into one vector, then loading four particles n to $n+3$ from cell j into the second vector, and then calculating all forces. However, now the pairwise forces can no longer be calculated by only running the two vectors through the same calculation steps as before. To calculate all pairwise forces, one vector has to be transmuted in such a way, that all forces can be calculated. For a visualization of this process please see Figure 22 and for the pseudocode see Figure 13.

In theory, this reduces the number of memory accesses to:

- every particle from cell i has to be loaded once
- every particle from cell j has to be loaded once for every vector of particles in cell i

Therefore the number of memory accesses is reduced to

$$sizeof(\text{cell i}) + \frac{sizeof(\text{cell i})}{vectorsize} * sizeof(\text{cell j}) \tag{6}$$

```
input: SoA_i, SoA_j
  for [particle_m .. particle_m+3] in SoA_i
    vector_m = load([particle_m .. particle_m+3])
    for [particle_n .. particle_n+3] in SoA_j
      vector_n = load([particle_n .. particle_n+3])
      repeat 4 times:
        calculate_forces(vector_m, vector_n)
        permute(vector_n)
```

Figure 13: The pseudocode for how force calculation for particle pairs is done in the improved implementation of the SoA two cells functor mode.

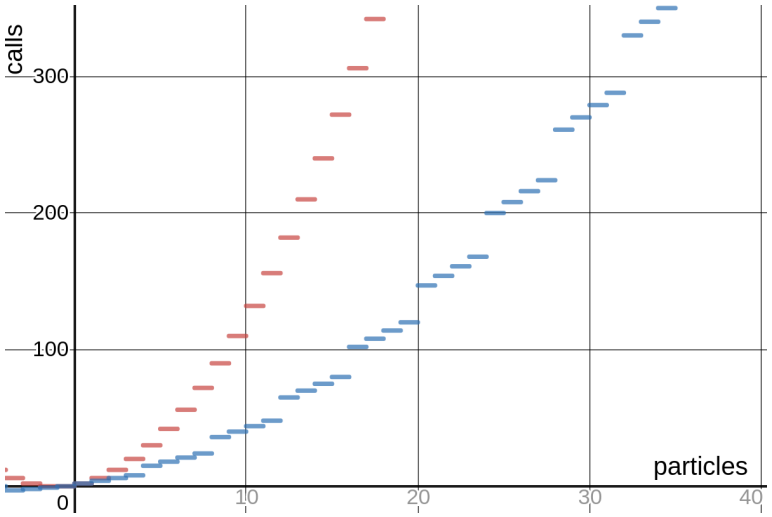


Figure 14: A plot to compare the number of memory accesses between the original LJ functor (red) and the improved one (blue). This plot was created with the website desmos.com and can be found at `https://www.desmos.com/calculator/vvm57b2vnr`

4.3 Comparison

This subsection covers the comparison of performance between the original implementation of the SoA two cells mode from the LJ functor and an improved one.

To start off, the theoretical number of memory accesses is plotted for both versions which can be seen in Figure 14.

As the plot shows, the number of memory loads is reduced by around 66%. Of course in the age of caches, not every memory load is equally expensive which makes this only a theoretical gain.

To compare the actual runtime of the two algorithms please refer to Figure 15. This graph shows that the theoretical gain in performance due to a reduced number of memory accesses, is not translated to an actual reduction in execution time.

Reasons for this are mainly, that this part of the force calculation was not memory bound to begin with. And also the time that is gained by a reduction in memory accesses is nullified

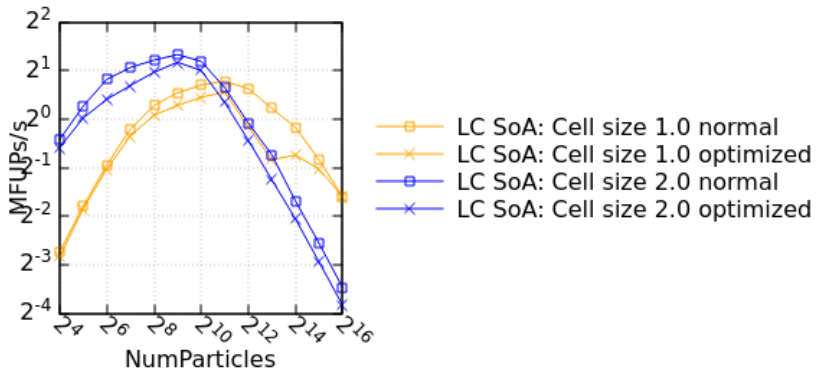


Figure 15: The comparison of performance between the original SoA two cells implementation and an improved one.

by the increase of overhead via permutations.

5 Conclusion

The conclusion of this paper is split into two separate parts, as the work for the code generator and the (not so) improved functor were independent.

Code Generator The concept of a domain specific language and a code generator worked very well in this concept. If further improved this definitely makes the AutoPas library more accessible to users that don't want to write and optimize an 800 line C++ code file.

Implementing the code generator was more work than initially thought. The most problems were caused by how to transfer data between the generated code and the main part of AutoPas. Namely getting particle parameters into the functor had to be resolved via string matching the parameters with attributes from the particle class. This also brings a very unintuitive and confusing set of different names for various stages of particle parameters to the code template. For example for the SoA one cell mode the SoA is not indexed, while for the SoA two cell mode there are two indexed SoAs provided. For some variables there exist accumulators.

Once this issue was solved though, it was less of a problem to actually generate the kernel code for the force calculation from the derivation tree of the AutoPasDSL code.

The performance of the generated functor was for the most part worse than the user implemented one. This was expected due to the obvious repetition of instructions that are placed at better positions in the handcrafted functor. However, the number of performance increase that is gained solely by compiler optimizations was really impressive, at least for the AoS mode of the functor. Unfortunately, the compiler is limited in optimizing the arithmetic part of the functor which was vectorized with Intel AVX intrinsics.

Optimizations of current Lennard Jones implementation The performance increase which was sought after by reducing the number of memory accesses was, at least for now, not satisfied. Even though the reduction of memory load operations was quite drastic, there was no reduction of runtime.

6 Future Work

Code Generator The first thing that has to be improved with this functor is its generalization capabilities. Meaning it has to be able to generate everything a user defines. Currently, this generator has been created and improved to the point, where it can generate a working implementation of the Lennard Jones 12-6 potential. Generated implementations of other forces will have to be tested against existing implementations to verify the correctness of this code generator.

As seen in the analysis of the generated functor, it very much lacks in performance. The automatic optimization of code is definitely a point where the compiler of the AutoPas-DSL has a lot of potential and should be one of the next points of approach for improvements.

Additionally, the number of platforms for which the compiler can generate code has to be increased. This includes newer (and maybe older) versions of Intel vector instruction sets, gpgpus and other acceleration hardware, etc.

Optimizations of current Lennard Jones implementation The theoretical model promises an improvement that is not reflected in the performance analysis. To further analyze the performance bottleneck of the current LJ functor more specific tools could be consulted.

With the help of those tools, there may even be a possibility to work the theoretical improvement into future functor implementations.

References

[Fra15] Niemeyer Frank. Simd fundamentals. part 1: From sisd to simd. `http://frankniemeyer.blogspot.com/2015/06/simd-fundamentals-part-i-from-sisd-to.html`, 2015. Accessed: Oct 11th 2019.

[Gen18] Eni Generalic. Lennard-jones potential. `https://glossary.periodni.com/glossary.php?en=Lennard-Jones+potential`, 2018. Accessed: Oct 10th 2019.

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

[MR] Daniel D. McCracken and Edwin D. Reilly. Backus-naur form (bnf). In *Encyclopedia of Computer Science*, pages 129–131. John Wiley and Sons Ltd., Chichester, UK.

[Yeh76] Raymond Tzuu-Yau Yeh. *Applied computation theory : Analysis, design, modeling.* Prentice-Hall series in automatic computation. Prentice-Hall, Englewood Cliffs, N.J. :, c1976.

7 Appendix

```
start : statics kernel

statics : "STATIC" "(" paramlist ")"

kernel : functorname "(" paramlist ")" "{" body "}"

paramlist : (param "," )* param

body : ([function | formula] ";")+

COMMENT : "//" /(.)+/

formula : param "=" a_expr

function : CNAME "(" param* ")"

a_expr : m_exp | a_expr PLUS m_exp | a_expr MINUS m_expr

m_expr : u_exp | m_expr MUL u_exp | m_expr DIV u_expr

u_expr : powe | "-" u_exp | "+" u_expr

power : primary [POW u_expr]

primary : number | param | "(" a_expr ")"

param : (TYPE)? (REF)? CNAME

number : NUMBER
functorname : CNAME

TYPE : "vec"
REF : "&"
PLUS : "+"
MINUS : "-"
MUL : "*"
DIV : "/"
POW : "**"
```

Figure 16: The entire formal grammar that defines the syntax of the AutoPasDSL, as used in the code generator. Note that NUMBER and CNAME are Lark imports for numbers and strings that start with a letter.

```
STATIC ( cutoff, epsilon, sigma, shift )
LJFunctor ( vec pos_i, vec pos_j,
            vec &force_i, vec &force_j,
            vec &virial, &upot ) {

  _sigmasquare = sigma * sigma;
  _epsilon24 = epsilon * 24;
  _shift6 = shift * 6;
  _cutoffsquare = cutoff * cutoff;

  squared_distance(pos_i pos_j);

  invdr2 = 1 / squared_distance;
  lj2 = _sigmasquare * invdr2;
  lj6 = lj2 * lj2 * lj2;
  lj12 = lj6 * lj6;
  lj12m6 = lj12 - lj6;
  fac = _epsilon24 * (lj12 + lj12m6) * invdr2;

  mask(fac);

  vec fac_dir = (pos_i - pos_j) * fac;

  force_i = force_i + fac_dir;
  force_j = force_j - fac_dir;

  virial = virial + fac_dir * (pos_i - pos_j);

  upot_temp = (_epsilon24 * lj12m6 + _shift6) / 6;
  mask(upot_temp);
  upot = upot + upot_temp;

}
```

Figure 17: The entire code to generate a Lennard Jones 12-6 functor in the AutoPasDSL.

```
start
  statics
    paramlist
      param cutoff
  kernel
    paramlist
      param pos_i
      param pos_j
      param
        &
        force_i
      param
        &
        force_j
      param
        &
        upot
    body
      formula
        param force_i
        a_expr
          a_expr
            m_expr
              u_expr
                power
                  primary
                    param pos_i
          +
          m_expr
            u_expr
              power
                primary
                  param pos_j
      function
        mask
        param upot
```

```
STATIC ( cutoff )

KERNEL ( pos_i,
         pos_j,
         &force_i,
         &force_j,
         &upot ) {

  force_i = pos_i + pos_j;
  mask(upot);

}
```

Figure 18: This is a direct visualization of how AutoPasDSL code (right) gets parsed into a derivation tree (left) created by lark using the grammar from Figure 16.

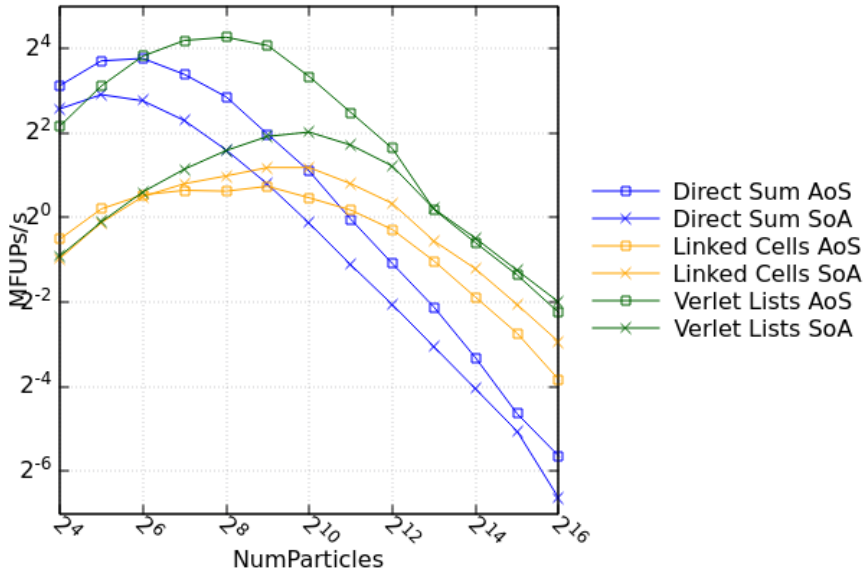


Figure 19: The overall performance of the handwritten LJ functor. MFUPs/s is an abreviation for Million Force Updates per Second.

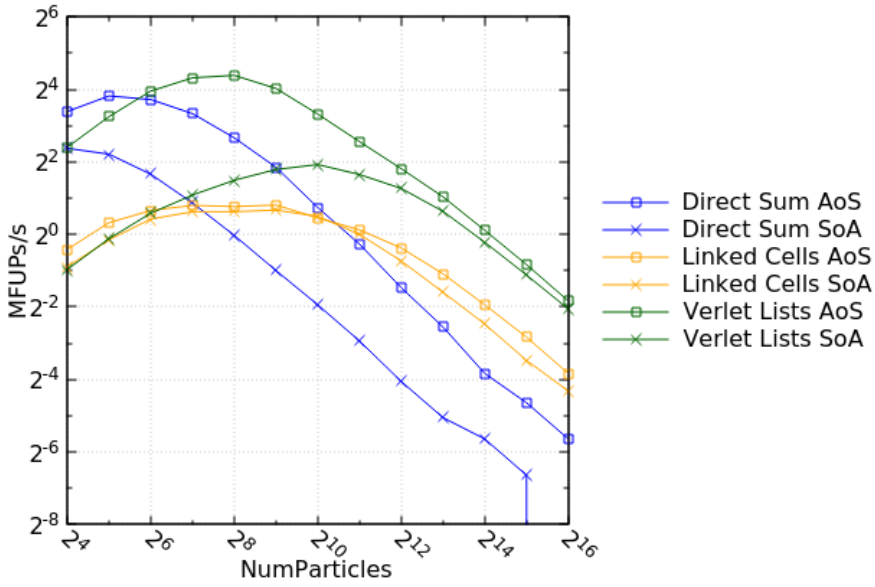


Figure 20: The overall performance of the generated LJ functor. MFUPs/s is an abreviation for Million Force Updates per Second.

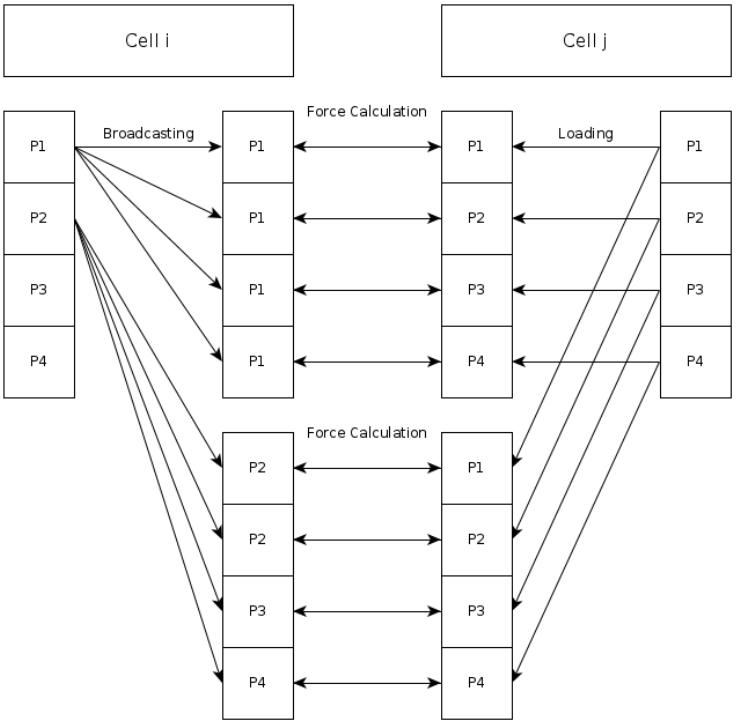


Figure 21: The original functor when calculating the pairwise forces between particles from different cells when the process is vectorized.

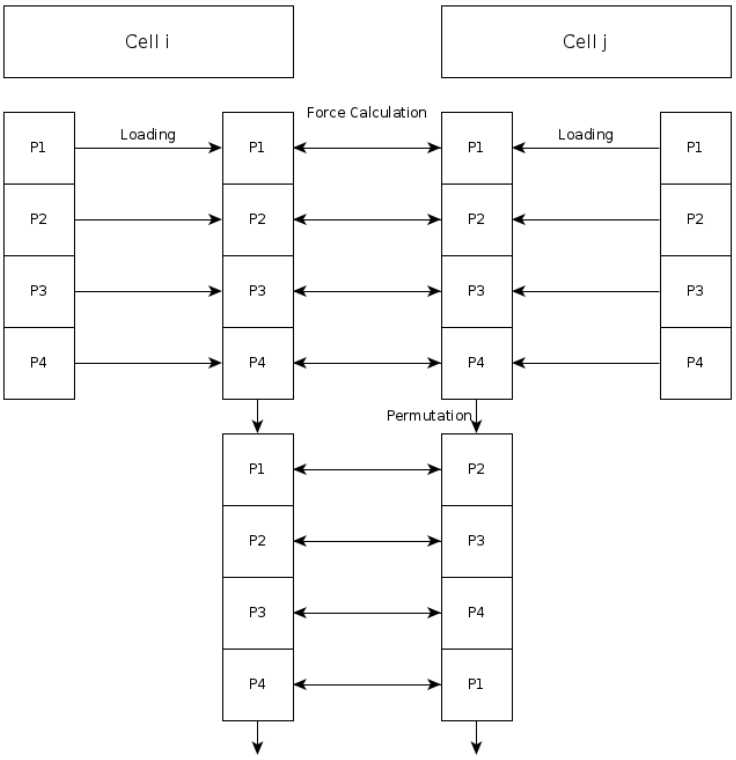


Figure 22: The improved functor when calculating the pairwise forces between particles from different cells when the process is vectorized