SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNICAL UNIVERSITY MUNICH

Bachelor's thesis in Informatics

Development of a Microbenchmarking Framework for Enhanced Parameter Selection in AutoPas

Kristin von Milczewski

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNICAL UNIVERSITY MUNICH

Bachelor's thesis in Informatics

Development of a Microbenchmarking Framework for Enhanced Parameter Selection in AutoPas

Entwicklung eines Microbenchmarking-Frameworks zur Verbesserung der Parameterauswahl in AutoPas

Author:	Kristin von Milczewski
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Samuel Newcome
Submission Date:	15.05.2024

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

Munich, 15.05.2024 Kristin von Milczewski

Abstract

Molecular dynamics (MD) simulations are becoming increasingly popular and relevant in various fields. This is mainly due to the fact that they have become faster, more accurate, and more accessible. To further improve MD simulations, programs like AutoPas are continuously developed through tuning.

One parameter to be tuned in this thesis is the sortingThreshold, which indicates at what number of particles it is advisable to generate a sorted view. As described below, an executable was developed to find the optimal threshold for the ProcessCell and ProcessCellPair methods. It was found that this threshold depends on the system or compiler used and is also different for the two functions.

The executable provides a framework to add more tunable parameters and implement optimization methods, which will be described in more detail at the end of the thesis.

Kurzfassung

Moleküldynamik-Simulationen (MD) werden immer beliebter und relevanter in diversen Bereichen. Die liegt vor allem auch daran, dass sie schneller, genauer und zugänglicher wurden. Um MD Simulationen weiter zu verbessern werden Programme wie AutoPas durch Tuning immer weiter entwickelt . Ein Parameter, der in dieser Thesis getunt werden soll, ist der sortingThreshold. Dieser gibt an, ab welcher Anzahl von Partikeln es sinnvoll ist eine sortedView zu generieren. Wie im Folgenden beschrieben wurde eine Executable entwickelt um den optimalen Threshold für die Methoden ProcessCell und ProcessCellPair zu finden. Dabei stellte sich heraus, dass dieser vom verwendeten System beziehungsweise Compiler abhängig und außerdem unterschiedlich für die beiden Funktionen ist. Die Executable bietet ein Framework um weitere tunebare Parameter hinzuzufügen und Optimierungsverfahren zu implementieren, was am Schluss der Thesis genauer beschrieben wird.

Contents

1 Introduction

Molecular dynamics (MD) simulations are gaining popularity and importance in various fields. For example, they are widely used in computational chemistry [Gra+19] or in the approximation of fluids in particle hydrodynamics. Other examples include biomedical applications. [Gra+22]

The increasing attention to molecular dynamics simulations in molecular biology and drug discovery is due to two main factors: First, the number of experimental structures of certain classes of molecules critical to neuroscience has exploded in recent years. On the other hand, molecular dynamics simulations themselves have become more powerful and accessible. Until recently, conducting high-impact research with MD simulations typically required access to supercomputers. However, advances in computer hardware, particularly GPUs, have made it possible to run powerful simulations locally at reasonable cost. In addition, MD simulation software has become more user-friendly, providing better support for non-experts. Furthermore, while MD simulations are based on physical models that are inherently approximate, their accuracy has improved significantly over time. [HD18]

While MD simulations serve different purposes, they all share a common principle: simulating the behavior of particles governed by some form of interaction. These interactions, typically non-bonded pairwise interactions, often dominate the computational workload of particle simulations and are therefore a key focus for optimization. Several algorithms, such as Linked Cells or Verlet Lists, have been developed to efficiently handle these interactions. However, no single algorithm is universally optimal for all scenarios. To address this challenge, the open-source C++ library AutoPas provides a variety of algorithm combinations and optimizations to provide a flexible solution for different simulation needs. [Gra+22]

In Chapter 2, we revisit the fundamentals of Molecular Dynamics Simulations and provide background information on AutoPas.

The GPTuner and Kernel Tuner systems, which represent methods for parameter tuning, are described in more detail in Chapter 3. The main focus of this thesis is the tuning of the SortingThreshold parameter, a process described in detail in

Chapter 4. In Chapter 5, we present the results of our computations regarding the optimal SortingThresholds, along with tests involving different thresholds. Finally, in Chapter 6 we discuss ideas and considerations for future work, ending with a short summary and conclusion in Chapter 7.

2 Background

2.1 Molecular Dynamics Simulations

The movements and velocities of the particles in molecular dynamics simulations are calculated every time step using force terms. They interact with each other using Verlet integration for Newton's second law of motion which says that

$$F_i = m_i \cdot a_i, \quad a_i = \frac{v_i}{dt}, \quad v_i = \frac{x_i}{dt} \tag{2.1}$$

with m being the mass of the particle and a being its acceleration. The acceleration is the derivative of the velocity v and x is the particle's position [Mic07, p. 39]. The force that acts on a particle is usually the sum of pairwise forces:

$$F_i = \sum_{\substack{j=1 \\ j \neq i}}^{N} F_{ij} \tag{2.2}$$

[Mic07, p. 49]

2.1.1 Lennard-Jones-Potential

In molecular dynamics simulations, the commonly used potential is the Lennard-Jones 12-6 potential, denoted as

$$U\left(r_{ij}\right) = 4\epsilon \left(\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right) \tag{2.3}$$

In this formula, r_{ij} represents the distance between two particles i and j. The parameters ϵ and σ characterize particle properties and determine the strength and zero-crossing of the potential. The Lennard-Jones 12-6 potential consists of an attractive part, simulating Van-der-Waals forces, and a repulsive part representing Pauli repulsion. [All04] To parallelize particle simulations on distributed systems such as HPC clusters, the computational domain is divided into sub-domains, with each cluster node storing one. Data duplication occurs at the borders between sub-domains to ensure accurate force calculations. Practically, this means that particles near the boundary of one sub-domain must also be available in its neighboring domain. These replicated particles are known as halo particles and are treated as interaction partners during force calculations. However, in traditional domain decomposition setups, forces are not computed for halo particles; instead, they are updated based on the sub-domain they belong to. [Gra+22]

2.1.2 The Cutoff Radius

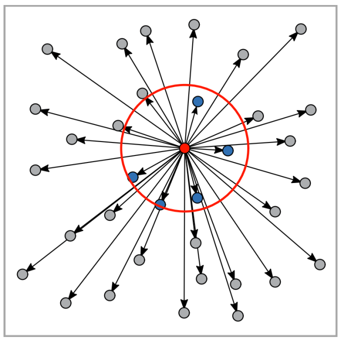
As the distance between particles increases, many force potentials typically decrease rapidly and converge to zero. To account for this, a cutoff radius is often implemented. When the distance between particles exceeds this cutoff radius, their interactions are ignored and assumed to have no effect. These types of potentials are called short-range potentials. Currently, only these are implemented in AutoPas; long-range potentials are not yet supported.

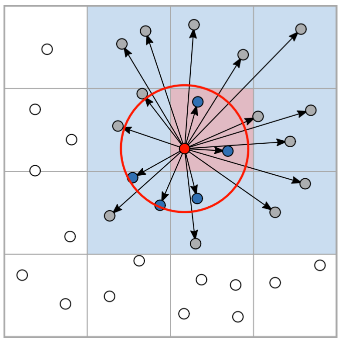
2.2 AutoPas

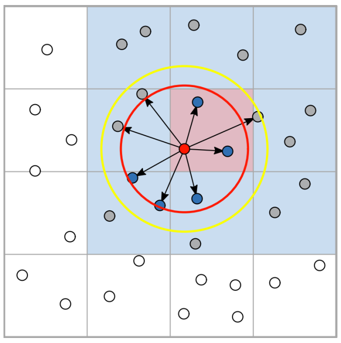
AutoPas is a versatile tool for high-performance short-range molecular dynamics simulations. It integrates various algorithms, parallelization strategies, and optimizations that are dynamically adjusted through auto-tuning. Because the most effective configuration can vary widely for different simulation scenarios and can evolve during runtime, AutoPas performs several tuning phases. During these phases, AutoPas evaluates the runtime of different configurations and selects the most efficient one. From the user's perspective, the C++ library acts as a black-box container, abstracting away its internal workings. [Gra+22]

2.3 Neighbor Identification Algorithms

The core of any particle dynamics simulation is the neighbor idenfication algorithm. The goal is to find the particles that are within the cutoff radius and thus relevant for the calculation of the pairwise forces. The chosen algorithm has a direct influence on the data structures and arrangements used to store the particles. Currently, four different algorithms are used in AutoPas, which will be introduced below. Along with the containers that implement the algorithms and variations of them. For this thesis, Linked Cells is particularly relevant. In Fig. 2.1 the different methods are visualized. The information in this section is taken from [Gra+22].


(a) Direct sum


(b) Linked Cells


(c) Verlet Lists

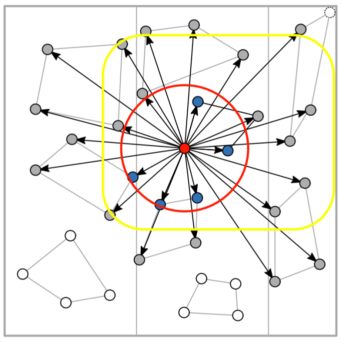

(d) Verlet Cluster Lists

Figure 2.1: Containers for the neighbor identification algorithms

2.3.1 Direct Sum

Looking at one particle, a simple way to identify particles within the cutoff radius is to calculate the distances to all the other particles. This is shown in Fig. 2.1a. The distances between the red particle and all potentially relevant particles are calculated. The red circle represents the cutoff radius, and the blue particles within it are relevant for the calculation of the pairwise forces. The gray particles are not included in the calculation of the force acting on the red particle.

This method is only practical for a small number of particles. But because it is very simple, it doesn't require any additional memory or algorithmic overhead, and therefore doesn't require any unique data organization. Thus, all standard particles are kept in a single vector. However, to manage the deletion or separate traversal of halo particles, a secondary array is maintained specifically for them.

2.3.2 Linked Cells

To organize the spatial data of the particles and thus improve the scalability of pairwise computations, the Linked Cells algorithm is introduced. It divides the area into a grid of cells with a mesh size equal to or larger than the cutoff radius and assigns the particles to these cells. As can be seen in Fig. 2.1b, the distances between the red particle and the particles in its own red cell, marked by arrows, are calculated first, followed by those in the adjacent blue cells. Pairwise forces are then determined for the blue particles identified within the cutoff radius. At this stage, the white particles outside the grid are not included in any calculations. An advantage of this algorithm is that it is independent of the number of particles in terms of memory consumption, since it is only concerned with maintaining a cell structure. Cache prefetching is feasible, and vectorization is relatively straightforward because sequentially processed particles are stored sequentially in memory. The main drawbacks of Linked Cells are the extra effort required to constantly organize moving particles into their designated cells, and the excessive distance computations caused by the imperfect match between the spherical boundary region and the Cartesian cells. While adaptive grid structures are conceivable to better approximate the cutoff sphere, their implementation would require more complex cell traversal schemes.

The Linked Cells container consists of a vector of uniform cells, each containing a vector of its actual particles. Particles that are spatially close to each other and are in the same cell are also nearby in memory. This facilitates effective traversal of smaller regions within the domain and allows efficient vectorization of particle computations. A disadvantage of linked cells is that they model space rather than particle relationships. This means that even regions of the domain without particles must still be checked during traversals.

2.3.3 Verlet Lists

Another way to identify neighbors is through Verlet Lists. In this method, for each particle a neighbor list is created that contains references to all particles within the cutoff radius, allowing the tracking of a particle's interaction partners. These lists are reused in as many iterations as possible, thus including particles slightly outside the cutoff radius. The so-called Verlet skin is shown as a yellow circle in Fig. 2.1c. The sum of the Verlet skin and the cutoff radius gives the interaction length, which is then used to compute neighbors by considering only the distances to the particles inside it. Comparing the number of distance calculations (arrows) in Fig. 2.1b and Fig. 2.1c, one can see that Verlet Lists require significantly fewer evaluations. One way to periodically generate the Verlet Lists is to use Linked Cells. This involves computing the distances from a particle to all particles within its own cell and those in adjacent neighboring cells. In subsequent iterations, the generated neighbor list is then used for force calculation. A notable drawback is the additional memory required for the neighbor lists. Furthermore, there is a lack of data locality.

The Verlet Lists container stores its particles inside a Linked Cells container instance. The neighbor lists are represented as vectors of particle pointers within the Linked Cells data structure. These lists are aggregated into a map of particle pointers, where each pointer corresponds to a specific neighbor list.

There are also variations of the Verlet Lists container. One of them is Var Verlet Lists, which represents a generalization and thus also builds on the Linked Cells container. This container offers a way to easily replace the implementation of the neighbor lists and their generation.

Another variant is Verlet Lists Cells. This assigns the neighbor lists to the cell where the corresponding particle is stored, instead of storing them in a single container vector.

2.3.4 Verlet Cluster Lists

Since neighboring particles have very similar Verlet Lists, a number M of particles are aggregated into a cluster. In Fig. 2.1c a cluster of size 4 is shown. To construct these clusters, the 3D domain is divided into a 2D grid along the x/y plane with a grid cell length of

$$\sqrt[3]{\frac{M}{N/V}} \tag{2.4}$$

where N is the total number of particles and V is the volume of the domain. This results in a partitioning of the domain into towers, where clusters are formed based on the order of the particles along the z-axis. Compared to Verlet Lists, this algorithm requires a higher number of distance computations. The more complex implementation is another limitation.

The Verlet Lists Cluster container consists of a grid structure arranged in the xy-plane, forming towers. Inside each tower, particles are stored in a single vector, arranged in order along the z-axis. Dummy particles are added to the last cluster of a tower if the number of particles in a tower is not divisible by M. These dummy particles are not included in the force calculations.

2.3.5 Algorithm selection problem

Although these algorithms appear to be incremental, each of them comes with a trade-off, such as higher memory consumption, ease of vectorization, or overhead due to a complicated data structure. There is no ideal method that will outperform all other approaches in every simulation scenario. For this reason, AutoPas uses automated algorithm selection to independently select the optimal configuration for a given scenario. The Automated Algorithm Problem is the task of selecting an algorithm for a given problem instance that maximizes an arbitrary performance metric. Such metrics can include precision, execution speed, memory usage, etc. In AutoPas, considerations are currently limited to time-to-solution. Depending on whether the selection of the algorithm is done only once or is re-evaluated repeatedly during runtime, a distinction is made between static and dynamic algorithm selection. Since the scenarios in particle simulations can change drastically during the course of a simulation, AutoPas uses dynamic tuning in periodic intervals.

2.4 Newton's Third Law

Newton's third law of motion essentially states that for every action there is an equal and opposite reaction. In the context of particle simulations, this law means that if there is a force F between two particles, let's call them i and j, then the force F exerted by i on j is equal in magnitude but opposite in direction to the force F exerted by j on i. So mathematically it's expressed as

$$F_{ij} = -F_{ij} \tag{2.5}$$

Therefore, it becomes possible to reduce by half the computations involved in calculating the pairwise force by using this equation. However, this adjustment requires direct updating of both variables that hold the accumulated force for each particle when calculating F_{ij}, which has significant implications for parallelization strategies.

2.5 Data Layouts

The data layout determines how the particle data is organized in memory. AutoPas supports two primary methods of storing structured data: Array of Structures (AoS) and Structure of Arrays (SoA).

In the AoS layout, particles are represented as C++ objects containing properties such as positions in x, y, and z dimensions, forces, etc. These objects are then stored consecutively in a

```
std::vector<Particle>.
```

In the SoA layout, each property, such as the x position, is stored in a separate

```
std::vector<double>
```

with an entry for each particle. SoAViews can also be created, which are references to a start and end point within an actual SoA.

AutoPas stores particle data in AoS format. When the SoA layout is selected, the data needed by the functor is copied to SoAs before the force calculation in each iteration, and then transferred back to AoS afterward. [Gra+22]

2.6 SortingThreshold

As described in Section 2.3.2, the Linked Cells algorithm segments the domain into cells. These cells are then traversed by the methods `CellFunctor::ProcessCell` and `CellFunctor::ProcessCellPair`. During this traversal, particle distances can be computed directly, or alternatively, the `CellFunctor` generates a pre-sorted view using the x-coordinates of the particles. While this sorted view eliminates unnecessary distance computations, it does incur a time overhead. Therefore, the optimal `sortingThreshold` parameter determines the particle count at which it becomes advantageous to use this approach.

In the `ProcessCell` method, the number of particle in a cell is compared to the sorting threshold. If it falls below the threshold, the sorted view is constructed:

```
if (cell.size() > _sortingThreshold) {
  SortedCellView<ParticleCell> cellSorted(cell,
  utils::ArrayMath::normalize(std::array<double, 3>{1.0, 1.0, 1.0}));
  ...
}
```

Figure 2.2: The sortingThreshold in ProcessCell.

Subsequently, in the `ProcessCellPair` methods, the numbers of particles of neighboring cells are aggregated:

```
if ((cell1.size() + cell2.size() > _sortingThreshold)
    and (sortingDirection != std::array<double, 3>{0., 0., 0.})) {
  SortedCellView<ParticleCell> cell1Sorted(cell1, sortingDirection);
  SortedCellView<ParticleCell> cell2Sorted(cell2, sortingDirection);
  ...
}
```

Figure 2.3: The sortingThreshold in ProcessCellPair.

2.7 md-flexible

To give users a quick overview of AutoPas, the library includes several example codes, one of which is md-flexible. This simple molecular dynamics simulation framework is built around AutoPas and supports two types of particle simulations: single-site and multisite molecular dynamics (MD) simulations. [New+22]. In single-site MD, md-flexible simulates the Lennard-Jones 12-6 potential on single-site particles using Störmer-Verlet time integration. On the other hand, multi-site MD supports molecules composed of fixed rigid bodies of LJ sites. Users can switch between these modes at compile time, with each mode requiring slightly different input files. md-flexible has several features, including a thermostat, periodic boundary conditions, support for multiple particle types, checkpointing functionality, and VTK output for visualization. While its primary purpose is to introduce AutoPas to first-time users, it also serves as a tool for developers to test AutoPas in a real simulation code. Exploring md-flexible is made easy by its user-friendly interface, which exposes every set() function of AutoPas to user input. In addition, md-flexible provides reasonable defaults for each option, allowing users to configure only the parameters they are interested in. Combined with simulation scenario generators, md-flexible facilitates rapid prototyping, benchmarking, and exploration of AutoPas capabilities. [Gra+22]

3 Related Work

To fine-tune specific parameters within an application, several programs are already available. Two notable examples are GPTuner and Kernel Tuner, both adept at optimizing continuous as well as multi-dimensional tunable parameters. However, utilizing these tuners requires some effort, including the incorporation of libraries and supplying values to them. Given that only a discrete parameter—the sorting threshold—is optimized within the executable for this project, a simpler, custom implementation may be more suitable. Moreover, adjusting the surroundings is necessary to facilitate the tuning of this parameter. Nevertheless, there is potential for the GPTuner or KernelTuner to be applied to AutoPas in the future (see Chapter 6). Hence, they will be further elaborated on below.

3.1 GPTuner

GPTuner is a manual-reading database tuning system [Lao+23]. It uses the power of large language models to unify the available tuning knowledge to select the most relevant knobs to tune, the range of values to consider, and also to suggest promising starting points for the optimization process. DB-Bert [Tru22] is a similar system that uses a transformer-based model to predict the performance of a given knob configuration.

In this section we will first give some background on large language models and Bayesian optimization. Then we will discuss GPTuner in more detail.

3.1.1 Large Language Models

Language models are a type of artificial intelligence that can generate human-like text. Large language models (LLMs) are a language models that have a large number of parameters (e.g., the largest version of GPT-3 has 175 billion parameters [Bro+20]). Examples of large language models are the GPT Series from OpenAI, with models such as GPT-3 [FC20] and GPT-4 [Ope23], and the Llama model family [Tou+23]. The field of large language models has grown drastically in popularity since the advent of chatGPT [Ope22], for instance the number of papers on arXiv mentioning large language models has increased exponentially since the release of chatGPT [Cha+24].

A crucial component of large language models is the self-attention mechanism [Vas+17] which comes from the Transformer architecture [Vas+17]. The self-attention mechanism allows the model to weigh the importance of different words in a sentence, allowing it to understand the context and relationships between words. Essentially, self-attention works by Computing a weighted sum of values based on similarities (dot products) between a query and key vectors, to produce attention scores that indicate the relevance of each word to the others in the sequence. These scores are then used to compute weighted sums of the value vectors, resulting in the final output [Vas+17].

Large language models work by using large amounts of data to learn patterns and relationships within language [FC20]. During training, the model is exposed to a large corpus of text and learns to predict the next word in a sequence given its context. This process involves adjusting the parameters of the model to minimize the difference between its predictions and the actual next words in the training data. Through this iterative process, the model gradually learns to generate coherent and contextually relevant text. Reinforcement Learning from Human Feedback (RLHF) is another important component of LLMs, allowing them to improve their performance through performance through interaction with human users [Ope23].

Large language models have been applied to a wide range of tasks, including text generation, translation, summarization, and question answering [FC20]. The application of large language models to database tuning, as in GPTuner [Lao+23], is discussed in Section 3.1.3 below.

3.1.2 Bayesian Optimization

Determining the best knob configuration of a DBMS can be described mathematically as

$$\theta^* = \arg\min_{\theta \in \Theta} f(\theta),$$

where $\theta = (\theta_1, \ldots, \theta_n)$ describes the knob configuration, $\Theta = \Theta_1 \times \cdots \times \Theta_n$ is the parameter space, and f is the objective function, for example, throughput or latency (for the throughput the $\arg\max$ is used). Classical optimization methods, such as grid search, random search, or gradient-based methods are not suitable for this problem due to the high dimensionality and expensive evaluation of the objective function. Furthermore, the objective function is often noisy, non-convex, and lacks gradient information. To address these problems, Bayesian Optimization builds a surrogate model of the objective function and uses an acquisition function to decide which configuration to evaluate in the next iteration. Typical choices for the surrogate model are Gaussian Processes or random forests [Hut18]. The purpose of the surrogate model is to approximate the objective function and optionally add uncertainty estimates.

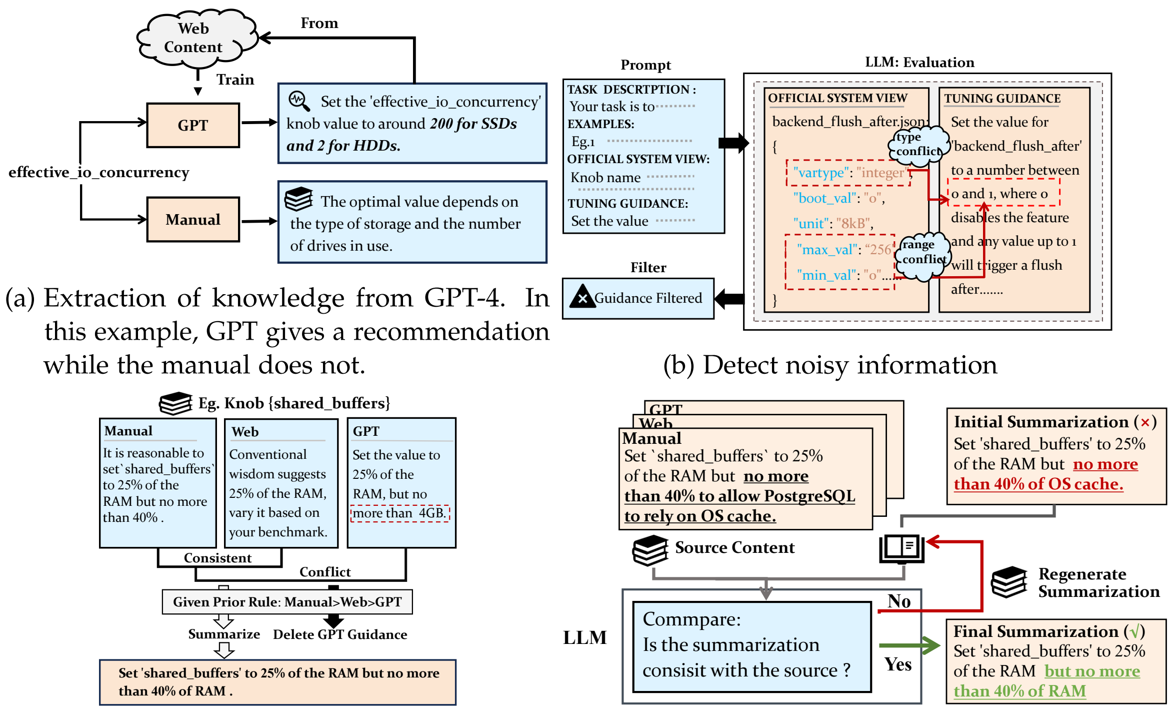
The acquisition function is used to balance exploration and exploitation and can be selected from a variety of functions, such as expected improvement, probability of improvement, or upper confidence bound [Hut18]. After each new function evaluation the surrogate model is updated and the acquisition function is minimized to find the next configuration to evaluate. To build the initial surrogate model, a set of initial configurations is evaluated. A classic choice is to use a Latin hypercube design [McK92]. A disadvantage of Bayesian Optimization is that a large number of evaluations is needed to find high performing configurations [FSH15]. This problem can be mitigated by using domain-specific knowledge to guide the initialization [FSH15], as is done in GPTuner [Lao+23] with a novel coarse-to-fine Bayesian Optimization framework.

3.1.3 GPTuner in a Nutshell

First GPTuner has to structure the available tuning knowledge. The structured tuning knowledge is then used to guide the Bayesian Optimization process by identifying the most relevant knobs to tune (and thus reducing the dimension of the search space), suggesting promising points, and limiting the ranges of the knobs to the relevant intervals. After its initialization, GPTuner's Bayesian Optimization process starts with a coarse search on a discretized grid of the parameter space. After a certain number of evaluations, the fine search is started using the surrogate model obtained from the coarse search. The fine search continues until a stopping criterion is met.

GPTuner combines the knowledge from the DBMS documentation, web forums, and large language models. Using the knowledge from GPT-4 can sometimes give useful suggestions that are not present in the DBMS documentation, for instance, when a suggestion is made on a blog that was used in the training data of GPT-4, as in the example in Fig. 3.1a. The knowledge obtained by the mentioned sources might be noisy or conflicting. To deal with this, GPTuner uses GPT-4 to compare an offical system view of a knob with the suggested tuning knowledge and classifies the information as noisy if these views differ. This process is illustrated in Fig. 3.1b. The different sources of tuning knowledge can also be inconsistent. To deal with this, GPTuner uses GPT-4 to check the tuning knowledge for consistency. In case of inconsistencies, the manual has the highest priority, followed by the DBMS documentation, and large language models have the lowest priority, as shown in Fig. 3.1c. Finally the tuning knowledge needs to be summarized. As the summarization can be inconsistent, the summary is checked for consistency using GPT-4, and repeated until a consistent summary is obtained, as illustrated in Fig. 3.1d.

The cleaned summary obtained by the above process then needs to be transformed into a structured format. To avoid the dependency of the output of the large language model on the input, GPTuner uses several different prompts, each of which produces a


Web Content
From
Train
GPT
Manual
effective_io_concurrency
Set the 'effective_io_concurrency' knob value to around 200 for SSDs and 2 for HDDs.
The optimal value depends on the type of storage and the number of drives in use.
Prompt
TASK DESCRTPTION : Your task is to
EXAMPLES: Eg.1
OFFICIAL SYSTEM VIEW: Knob name
TUNING GUIDANCE: Set the value
Filter
Guidance Filtered
LLM: Evaluation
OFFICIAL SYSTEM VIEW
backend_flush_after.json
{ "vartype": "integer", "boot_val": "0", "unit": "8kB", "max_val": "256", "min_val": "0"... }
type conflict
range conflict
TUNING GUIDANCE
Set the value for 'backend_flush_after' to a number between 0 and 1, where 0 disables the feature and any value up to 1 will trigger a flush after.......
Eg. Knob {shared_buffers}
Manual
It is reasonable to set`shared_buffers` to 25% of the RAM but no more than 40% .
Web
Conventional wisdom suggests 25% of the RAM, vary it based on your benchmark.
GPT
Set the value to 25% of the RAM, but no more than 4GB.
Consistent
Conflict
Given Prior Rule: Manual>Web>GPT
Summarize
Delete GPT Guidance
Set 'shared_buffers' to 25% of the RAM but no more than 40% of RAM .
GPT
Web
Manual
Set `shared_buffers` to 25% of the RAM but no more than 40% to allow PostgreSQL to rely on OS cache.
Source Content
Initial Summarization (×)
Set 'shared_buffers' to 25% of the RAM but no more than 40% of OS cache.
Regenerate Summarization
LLM
Commpare: Is the summarization consisit with the source ?
No
Yes
Final Summarization (√)
Set 'shared_buffers' to 25% of the RAM but no more than 40% of RAM

(a) Extraction of knowledge from GPT-4. In this example, GPT gives a recommendation while the manual does not.

(b) Detect noisy information

(c) Check the knowledge for consistency

(d) Check if the summary is consistent

Figure 3.1: The preparation of the tuning knowledge in GPTuner. Figures are taken from [Lao+23].

JSON object. The results are then aggregated using element-wise majority voting.

The structured tuning knowledge is then used to optimize the search space for the Bayesian Optimization process. This is done by identifying the most relevant knobs to be tuned and by restricting the ranges of these knobs. However, before starting the optimization, the entire search space is first optimized, GPTuner first creates a discrete grid based on the recommended values. by some factors, since the recommended values may not be optimal. After initializing the Bayesian Optimization process with Latin Hypercube Sampling, the surrogate model is updated, by evaluating the acquisition function over all the grid points of the smaller search space. This is the coarse search phase. In the following fine search phase, the surrogate model from the is used to evaluate the acquisition function over the entire search space until a stopping criterion is met.

GPTuner performs better than DB-BERT and vanilla Bayesian Optimization on the TPC-H and TCP-C benchmarks. GPTuner also outperforms DB-BERT when BERT is replaced by GPT-4 in DB-BERT. The individual components of GPTuner were also evaluated, showing that that space optimization, the use of GPT-4, and the coarse-to-fine Bayesian Optimization framework all contribute to the performance of GPTuner.

3.2 Kernel Tuner

Kernel Tuner [Wer19] is an auto-tuning system for GPU kernels. A kernel is the code that is executed on the GPU. Similar to database tuning, kernel tuning is expensive to evaluate the objective function. Kernel Tuner implements several methods to find a good configuration for a given kernel. These methods are described in Section 3.2.1.

3.2.1 Configuration selection strategies

The selection strategies are used to find the next configuration to evaluate. Kernel Tuner's default strategy is *brute force*. While this strategy is simple and guarantees to find the best configuration, it is not feasible for more complex kernels with larger search spaces. The other strategies are random sample, minimize, basin hopping, differential evolution, simulated annealing, particle swarm optimization, genetic algorithm, and firefly algorithm. These strategies are described in more detail in the rest of this section.

Random search just takes a random sample from the search space. The user has to specify how many configurations to evaluate.

The minize strategy uses a classical optimization methods to finde the next configuration to evaluate and also to check for convergence. A variety of optimization methods are available, namely Nelder-Mead, Powell, Conjugate Gradient, BFGS, L-BFGS-B, TNC [DS83], COBYLA, and SLSQP.

The basin hopping strategy [WD97] pertubes the current configuration randomly and then uses a local optimizer (any solver from the minimize strategy) to find the next candidate configuration. This candidate configuration is then accepted or rejected as the new configuration based on the objective function value.

The other strategies are not discussed in more detail here as they are not of direct relevance to this thesis.

Most of the above methods are restricted to continuous search spaces. To tackle this problem, Kernel Tuner computes the nearest valid configuration to the result of the optimization method. Additionally, for each selection strategies except differential evolution, genetic algorithm, and simulated annealing, each variable is scaled to the interval $[0, 1]$, or to an even smaller interval for some parameters. To deal with the problem of measurement errors, previously measured configurations are cached for consistency.

3.2.2 Comparison of selection strategies

The selection strategies are compared using kernel for 2D convolution, matrix multiplication (GEMM), and Point-in-Polygon.

For 2D Convolution, the evaluation reveals distinct performance distributions among the strategies. The brute force approach demonstrates a narrow peak, indicating a limited number of configurations yielding high performance. As the minimize strategy does not use a strategy to avoid local minima, it is not surprising that no solver is able reach near-optimal performance. Basin hopping reaches near-optimal performance for five of the eight solvers. Besides the perfomance of the found configuration, it is also important to consider the tuning time. For example particle swarm optimization yields better results than basin hopping (3832.81 GFLOP/s vs. 3705.42 GFLOP/s).

In the case of Matrix Multiplication (GEMM), the comparison portrays a less volatile distribution landscape compared to 2D Convolution. Differential evolution is capable of achieving near-optimal performance, but is only slightly better than random search. Basin hopping is efficiently achieving near-optimal configurations with significantly reduced tuning times compared to brute force search. The global optimization methods all outperform random search.

In the Point-in-Polygon application, the evaluation presents a challenging search space. Besides basin hopping, no selection strategy is able to outperform random search.

While certain strategies excel in specific applications, no single approach universally outperforms others across all scenarios. The authors of [WNW21] created an implementation of Bayesian Optimization specifically for GPU kernel tuning, that outperforms all existing strategies in the Kernel Tuner framework and also general Bayesian Optimization implementations that are not tailored to GPU kernel tuning. This implementation of Bayesian Optimization for GPU kernel tuning was added to Kernel Tuner.

4 Implementation

Below is an overview of the development of an executable that aims to determine the optimal sorting threshold for the ProcessCell and ProcessCellPair functions, as discussed in Section 2.6.

4.1 The md-flexible-tuner executable

4.1.1 Extension of the Simulation Class

First, an extension to the `Simulation.cpp` file from md-flexible is required. To find out how to do this, code analysis was done using vscode as a C++ debugger. Using CMake and the Cmake Tools Extensions, a debugging configuration can be set up. Setting breakpoints on relevant methods such as processCellPair and inspecting the call stack helps to understand how parameters and context are passed to these methods.

The new class `SimulationExtForTuning` inherits from `Simulation` which is the main class in md-flexible. `SimulationExtForTuning` thus has access to the configuration and the `CellFunctor`. The class is able to construct the parameters needed to call the methods to be tuned.

The methods `SimulationExtForTuning::processCell` and `SimulationExtForTuning::processCellPair` create suitable `ParticleCells` by randomly distributing particles within the given volume. `ProcessCell` distributes the particles within one cell, while `ProcessCellPair` distributes them across two cells.

This is because the `CellFunctor::ProcessCellPair` methods compare the combined size of two cells against the `sortingThreshold`, so it is not advisable to fill both cells with the given number of particles. You then call the `processCell` or `processCellPair` method using the `CellFunctor`.

Runtime metrics are taken outside of the `SimulationExtForTuning` class.

`SimulationExtForTuning::applyWithChosenFunctor` is an exact copy of the private method `Simulation::applyWithChosenFunctor`.

It is a possible optimization to make `Simulation::applyWithChosenFunctor` protected.

Below are the functions that call `CellFunctor::ProcessCell` as well as `CellFunctor::ProcessCellPair`.

```
bool callProcessCell(FunctorType *f,
                     autopas::FullParticleCell<ParticleType> &cell,
                     double interactionLength,
                     autopas::DataLayoutOption dataLayout,
                     bool useNewton3, size_t sortingThreshold) {
  autopas::internal::CellFunctor<autopas::FullParticleCell<ParticleType>,
  FunctorType, false> cf(f, interactionLength, dataLayout, useNewton3);
  cf.setSortingThreshold(sortingThreshold);
  cf.processCell(cell);
  return true;
}

bool callProcessCellPair(FunctorType *f,
                       autopas::FullParticleCell<ParticleType> &cell1,
                       autopas::FullParticleCell<ParticleType> &cell2,
                       const std::array<double, 3> &sortingDirection,
                       double interactionLength,
                       autopas::DataLayoutOption dataLayout,
                       bool useNewton3, size_t sortingThreshold) {
  autopas::internal::CellFunctor<autopas::FullParticleCell<ParticleType>,
  FunctorType, false> cf(f, interactionLength, dataLayout, useNewton3);
  cf.setSortingThreshold(sortingThreshold);
  cf.processCellPair(cell1, cell2, sortingDirection);
  return true;
}
```

4.1.2 Main.cpp

The main function of the executable is to compute the optimal SortingThreshold for a simulation. To achieve this, one could iterate over different threshold values and run the simulation with reduced iteration counts for each value. The iteration with the shortest runtime would theoretically indicate the best threshold. However, this approach is inefficient in terms of overall runtime and may produce inaccurate results due to unintended side effects. Instead, only the relevant parts of the simulation are considered-specifically, the ProcessCell and ProcessCellPair methods where the SortingThreshold is applied. As described in Chapter 5, these methods have different optimal thresholds and are therefore analyzed separately.

For each method, a loop iterates over the number of particles starting at 2. Within this loop, the `ProcessCell/ProcessCellPair` method of the Extended Simulation class is called several times: once with a `sortingThreshold` of 0 and once with a `sortingThreshold` of `INT_MAX`. The execution time in milliseconds is recorded for each call. (A `sortingThreshold` of 0 implies that sorting is performed for any number of particles, since the cell size always exceeds the threshold, as described in the Background chapter. Conversely, a `sortingThreshold` of `INT_MAX` will not sort for any number of particles, since the cell size will always be less than the threshold). Since calls to these methods are computationally fast, typically taking between 0 and 1 milliseconds, each is called 100000 times to generate reliable comparison data. Then the difference between the recorded times is calculated:

```
Difference = DurationThreshold0.count() - DurationThresholdMax.count();
```

For small particle counts, the difference is consistently positive, indicating that direct distance calculation without prior sorting is faster. However, as the particle count increases beyond a certain threshold, this trend reverses and the difference becomes negative. This threshold represents the optimal `sortingThreshold` for the `ProcessCell/ProcessCellPair` method.

To ensure more stable results, this calculation is repeated three times for each method and an average value is calculated. The resulting optimal thresholds are then displayed on the console.

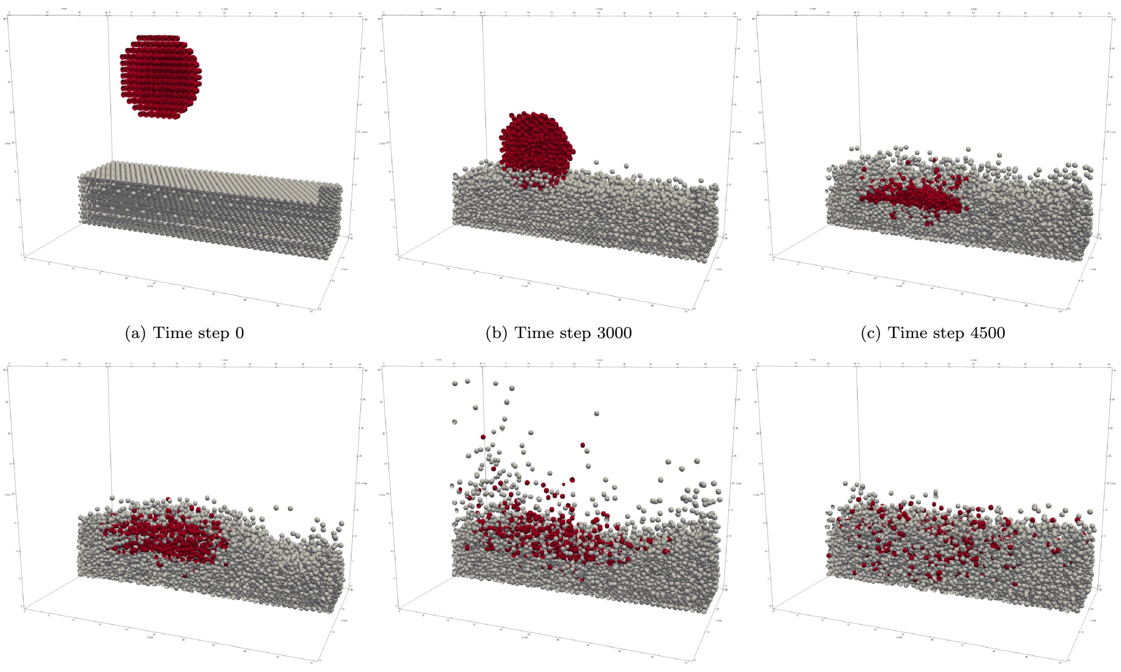
GitHub AutoPas Repository:
The executable for tuning the `sortingThresholds` has been added to the AutoPas repository on GitHub under commit ID a1812099c77b2022ff1858f9c98b58f25c103d42. The results obtained from running this version are detailed in the subsequent chapter. [1]

[1] https://github.com/AutoPas/AutoPas/commit/a1812099c77b2022ff1858f9c98b58f25c103d42

5 Results

5.1 Examples used

5.1.1 FallingDrop



(a) Time step 0 (b) Time step 3000 (c) Time step 4500

(d) Time step 5500 (e) Time step 9500 (f) Time step 35000

Figure 5.1: Cross-section of the falling drop scenario colored by particle type.

The primary example used for simulations and testing of the md-flexible tuner is the Falling Drop scenario, which involves about 18,000 particles. It entails modeling a cluster of particles forming a sphere that is accelerated by gravity and then falls into a layer of particles, as shown in Fig. 5.1. The boundaries of the domain constrain the motion: the top and bottom sides reflect particles through a particle wall of infinite mass, while the remaining sides have periodic boundary conditions. The impact shock (Fig. 5.1d) is reflected at the bottom of the domain, causing particles from both the initial drop and the bed to scatter back (Fig. 5.1e). Over time, the particles begin to stabilize at the bottom of the domain due to gravity (Fig. 5.1f). [Gra+22]

5.1.2 Exploding Liquid

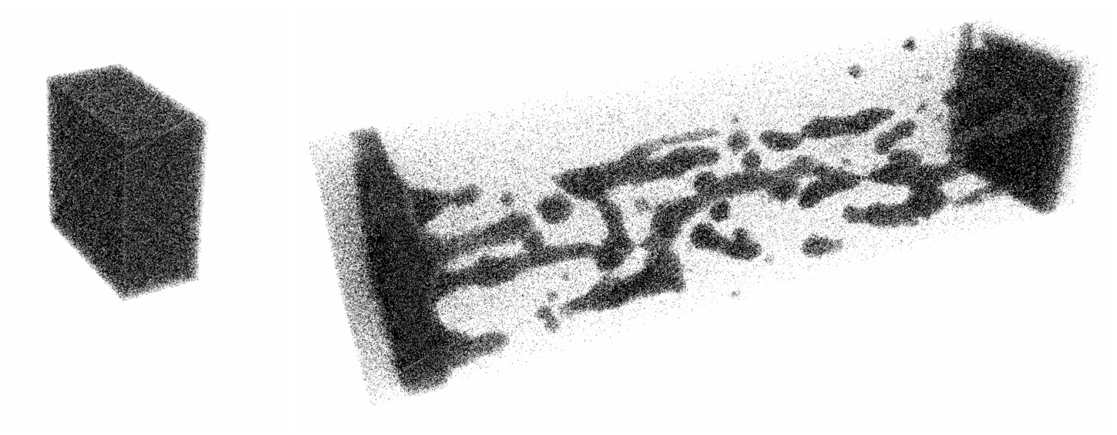


Figure 5.2: Exploding liquid scenario at the start (l.) and the end (r.) of the simulation.

Another example scenario involves an exploding liquid, featuring a highly compressed and heated liquid film suddenly exposed to vacuum. This results in the rapid expansion of the film, leading to its disintegration into filaments and droplets. [Sec+21]

5.2 Computation of optimal SortingThresholds

	Clang-14	GCC-12	GCC-11
ProcessCellPair	15	9	7
ProcessCell	8	5	5

Table 5.1: Different optimal SortingThresholds.

The calculation of optimal sorting thresholds for the ProcessCell and ProcessCellPair methods was performed on different systems using different compilers. The results show that both the compiler and the system used can affect the results. However, for a given system and compiler, the results remain relatively consistent over multiple runs.

The following results were obtained using the FallingDrop simulation described earlier, and are also presented in Section 5.2. Using the Windows Subsystem for Linux (GNU/Linux 5.15.146.1-microsoft-standard-WSL2 x86_64) on a Windows 11 Pro System with 4 cores and Ubuntu 22.04.4 LTS, the two compilers Clang and GCC gave different results:

With Ubuntu clang version 14.0.0-1ubuntu1.1, the optimal sorting threshold for ProcessCellPair was consistently found to be 15 across multiple runs. This means that sorting should ideally be performed for particle counts greater than 15. For ProcessCell, the optimal threshold was found to be 8. Using gcc version 12.3.0 (Ubuntu 12.3.0-1ubuntu1 22.04), the optimal sorting threshold for ProcessCellPair was calculated to be 9, while for ProcessCell it was 5.

On a server system with 8 cores and gcc version 11.4.0 (Ubuntu 11.4.0-1ubuntu1 22.04), the optimal threshold was found to be 7 for ProcessCellPair and 5 for ProcessCell.

Different optimal sorting thresholds for ProcessCell and ProcessCellPair:
The differences in optimal sorting thresholds for ProcessCell and ProcessCellPair can be attributed to the fact that ProcessCellPair requires the creation of two sortedViews when the combined size of the two cells exceeds the threshold. Consequently, the point at which creating these sortedViews becomes advantageous in terms of computation time is reached later than in ProcessCell, which only requires a single sortedView for one cell.

Using a different example:
When the exploding liquid scenario is used instead of FallingDrop, the results remain consistent. This indicates that the sortingThreshold is influenced by the system or the compiler and not by the specific simulation. The reason may be that the interaction computation does not differ a lot between the two scenarios.

Intersection:
As discussed in Chapter 4, the optimal sortingThreshold is identified when the difference in duration between calling processCell/processCellPair with a sortingThreshold of 0 and calling it with a threshold of `INT_MAX` becomes negative. By plotting these durations on an xy-graph, the intersection point reveals the optimal threshold. Fig. 5.3 illustrates this process by showing the calculation of the optimal threshold for ProcessCellPair using GCC-11 on the server system described earlier.

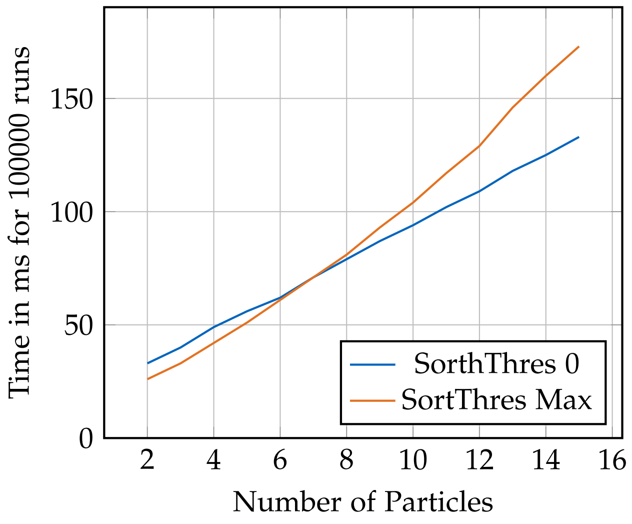


Figure 5.3: Intersection at 7 shows the optimal SortingThreshold calculated for ProcessCellPair using GCC-11.

5.3 Tests with different sortingThresholds

To evaluate whether using the optimal sortingThreshold really improves runtime, complete FallingDrops simulations were run with different thresholds. The results are shown in Fig. 5.4 - Fig. 5.8.

For Fig. 5.4 - Fig. 5.7 violin plots were used. Bootstrapping [Efr79] is used to estimate the mean performance. Bootstrapping works by repeatedly sampling with replacement from the original data set and calculating the mean of each sample. To make the comparison between different thresholds, the sampling is done on the number of runs and then the corresponding performance values are averaged for each threshold. For these plots, 1000 runs were performed for each.

First, the FallingDrop simulation was run five times for sortingThresholds ranging from 0 to 20 using the clang compiler, with time measured in milliseconds. The same sortingThreshold was used for both processCell and processCellPair. Ideally, a common optimal sortingThreshold would fall between their respective optimal thresholds. As shown in Fig. 5.4, simulations with a sortingThreshold around 12 yielded the fastest results, which actually falls in the middle. However, this point also had the largest variance, as shown by the distribution of the blue area along the y-axis.

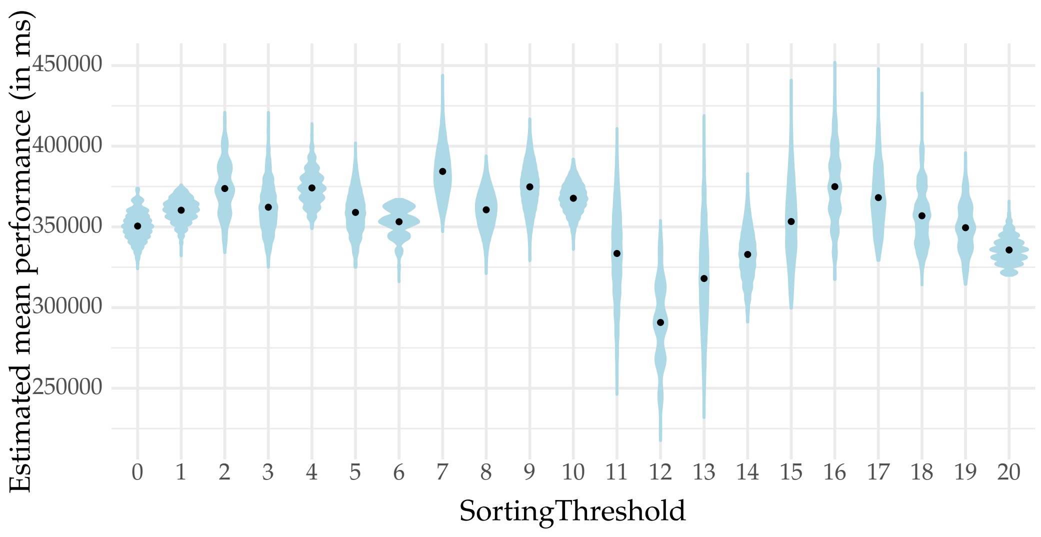


Figure 5.4: Performance by threshold using the clang compiler

To get more reliable results, additional simulations were performed with GCC-11 on a server system, where all simulation runs were significantly faster. First, the simulations were run with the same sortingThreshold for both ProcessCell and ProcessCellPair. As mentioned above, the optimal sortingThreshold was found to be 5 for ProcessCell and 7 for ProcessCellPair. In Fig. 5.5 the runtimes are shown for thresholds ranging from 0 to 10, with each threshold measurement repeated 20 times. As can be seen, the results are very close, with no significant improvement from the sortingThreshold.

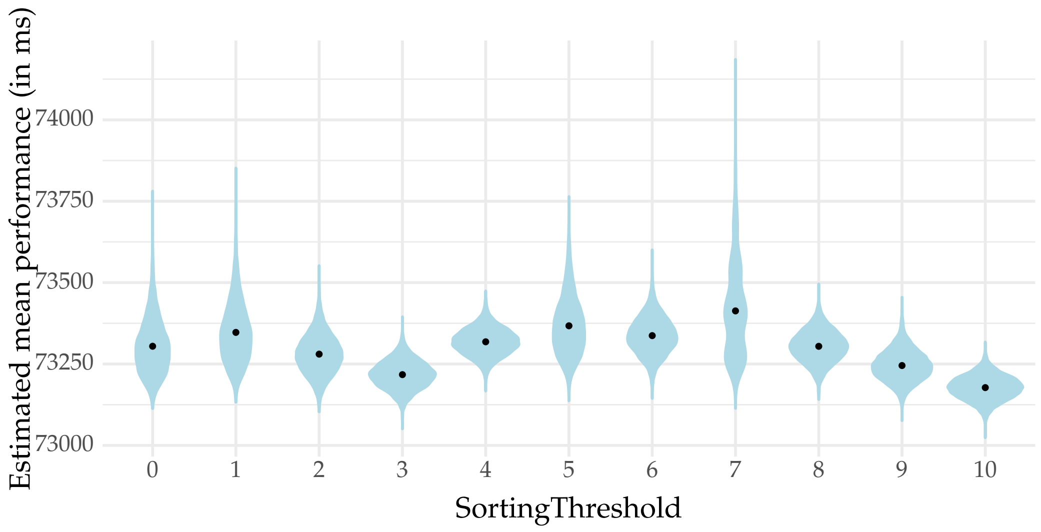


Figure 5.5: Performance by threshold using the GCC-11 compiler

Since there are two different optimal thresholds, two additional measurements were performed, each with one of the sortingThresholds set in processCell/processCellPair. Thirty measurements were taken for each threshold. Fig. 5.6 illustrates the result when the sortingThreshold is fixed to 5 within the implementation of the processCell method. Thus, the simulations with thresholds ranging from 0 to 10 show when ProcessCellPair performs best. Again, no significant differences are observed, especially since the runs for thresholds 5 to 10 show very similar performance.

In Fig. 5.7, the sortingThreshold in processCellPair was concurrently fixed at 7. Indeed, there is an improvement in runtime at 5, although once more, with little deviation from the other measurements.

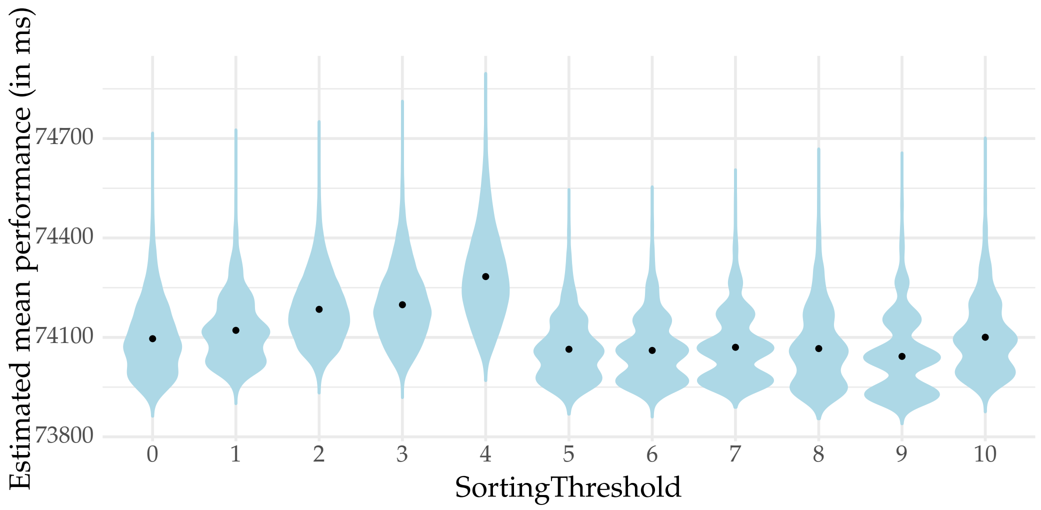


Figure 5.6: Performance by sortingThreshold for ProcessCellPair with a fixed threshold of 5 in ProcessCell using the GCC-11 compiler

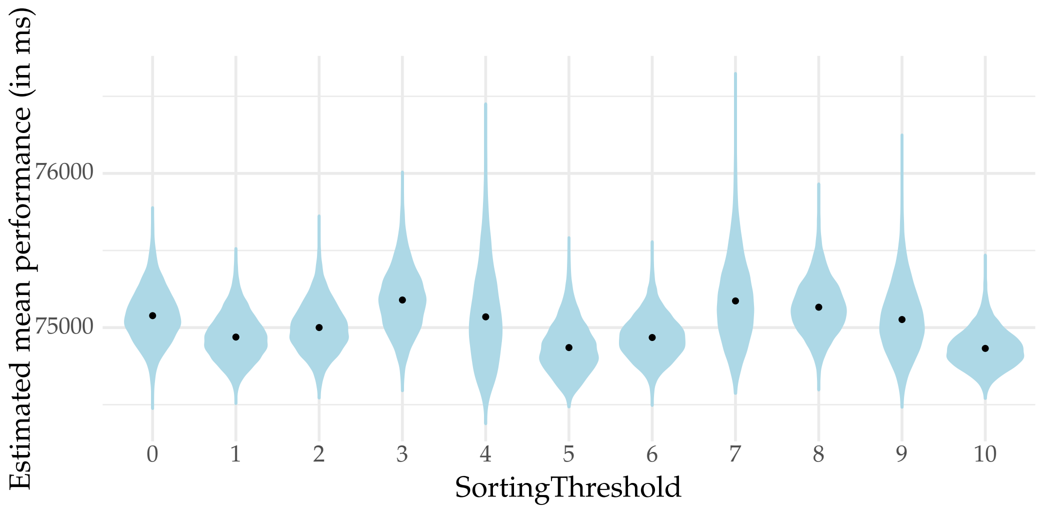


Figure 5.7: Performance by sortingThreshold for ProcessCell with a fixed threshold of 7 in ProcessCellPair using the GCC-11 compiler

Fig. 5.8 shows the previously described measurement from another perspective. This shows the variation per run. Except for runs 22 and 23 there are no major deviations. So the simulations are quite stable.

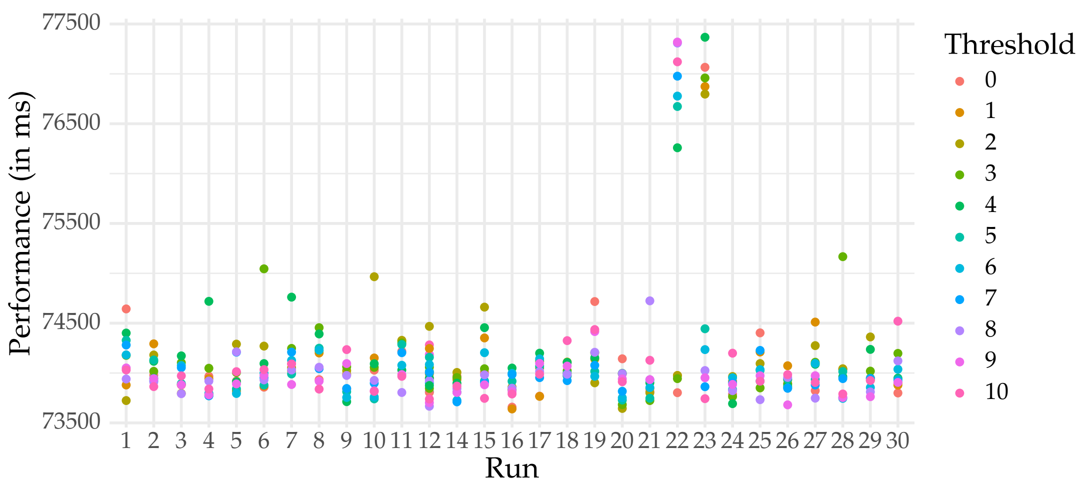


Figure 5.8: Performance by run using the gcc compiler, fixed sortingThreshold of 7 in ProcessCellPair

In conclusion, the sorting threshold does not make a significant difference at this low number of iterations. For larger simulations, it might be useful to assign a separate threshold to each of the methods, ProcessCell and ProcessCellPair, since their optimal thresholds are different.

6 Future Work

There are various ways to further improve AutoPas in future work. For example, if additional discrete tunable parameters are identified, they could be added to the executable described in this thesis. This provides a simpler approach than using one of the tuners mentioned here.
However, the GPTuner or KernelTuner could also be beneficial if continuous or multi-dimensional tunable parameters are identified. In that case, AutoPas would need to be made usable as a kernel for the KernelTuner in the form of an executable. Furthermore, it would also be an option to directly implement optimization methods such as Bayesian optimization or basin hopping for AutoPas.
Apart from additional tunable parameters, AutoPas could also be optimized for metrics other than runtime. For example memory usage, energy consumption, precision, complexity.
Understanding compiler optimisations: Since the runtimes of simulations vary significantly across different compilers, it might also be advantageous to examine the differences between Clang and GCC in this regard.
Providing a detailed answer to the following question could also lead to an improvement of AutoPas: What roles do parallelism and or execution on an GPU play?

7 Summary and Clonclusion

The optimal sortingThreshold indicates the number of particles from which it is advantageous to use a sortedView for the cells in the ProcessCell and ProcessCellPair methods which are particularly relevant when using the Linked Cells algorithm. To compute these thresholds, an executable was developed, consisting of an extension of the Simulation class and a Main function. This extension class aims to populate cells with a specific number of particles and then invoke ProcessCell(Pair). In the Main function, timing measurements and calculations are conducted.

The optimal sortingThreshold is determined when the difference in duration between calling processCell/processCellPair with a sortingThreshold of 0 and calling it with a threshold of `INT_MAX` becomes negative. Calculating the best thresholds on various systems with different compilers yielded varying results. Additionally, it was observed that processCell and processCellPair have different optimal thresholds.

In conclusion, it would be beneficial to assign different thresholds to the two methods in the implementation. However, the sortingThreshold has no significant effect on simulations with a smaller number of iterations. Therefore, in future work, it might be worthwhile to tune other parameters and employ optimization techniques.

List of Figures

List of Tables

Bibliography

[All04] M. P. Allen. "Introduction to Molecular Dynamics Simulation". In: *Computational Soft Matter: From Synthetic Polymers to Proteins* 23 (2004), pp. 1–28.

[Bro+20] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. "Language models are few-shot learners". In: *Advances in neural information processing systems* 33 (2020).

[Cha+24] Y. Chang, X. Wang, J. Wang, Y. Wu, L. Yang, K. Zhu, H. Chen, X. Yi, C. Wang, Y. Wang, et al. "A survey on evaluation of large language models". In: *ACM Transactions on Intelligent Systems and Technology* 15.3 (2024), pp. 1–45.

[DS83] R. S. Dembo and T. Steihaug. "Truncated-Newton algorithms for large-scale unconstrained optimization". In: *Mathematical Programming* 26.2 (1983), pp. 190–212.

[Efr79] B. Efron. "Bootstrap Methods: Another Look at the Jackknife". In: *The Annals of Statistics* (1979), pp. 1–26.

[FC20] L. Floridi and M. Chiriatti. "GPT-3: Its nature, scope, limits, and consequences". In: *Minds and Machines* 30 (2020), pp. 681–694.

[FSH15] M. Feurer, J. Springenberg, and F. Hutter. "Initializing bayesian hyperparameter optimization via meta-learning". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 29. 1. 2015.

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

[Gra+22] F. A. Gratl, S. Seckler, H.-J. Bungartz, and P. Neumann. "N ways to simulate short-range particle systems: Automated algorithm selection with the node-level library AutoPas". In: *Computer Physics Communications* 273 (2022), p. 108262. ISSN: 0010-4655. DOI: `https://doi.org/10.1016/j.cpc.2021.108262`.

[HD18] S. A. Hollingsworth and R. O. Dror. “Molecular Dynamics Simulation for All”. In: *Neuron* 99(6) (2018), pp. 1129–1143. DOI: `https://doi.org/10.1016/j.neuron.2018.08.011`.

[Hut18] F. Hutter. *Automated Machine Learning: Methods, Systems, Challenges*. Ed. by L. Kotthoff and J. Vanschoren. Cham: Springer, 2018. ISBN: 9783030053184.

[Lao+23] J. Lao, Y. Wang, Y. Li, J. Wang, Y. Zhang, Z. Cheng, W. Chen, M. Tang, and J. Wang. *GPTuner: A Manual-Reading Database Tuning System via GPT-Guided Bayesian Optimization*. 2023. arXiv: `2311.03157 [cs.DB]`.

[McK92] M. D. McKay. “Latin hypercube sampling as a tool in uncertainty analysis of computer models”. In: *Proceedings of the 24th conference on Winter simulation*. 1992, pp. 557–564.

[Mic07] G. Z. Michael Griebel Stephan Knapek. *Numerical Simulation in Molecular Dynamics: Numerics, Algorithms, Parallelization, Applications*. 5th ed. Berlin: Springer, 2007. ISBN: 978-3-540-68094-9.

[New+22] S. J. Newcome, F. A. Gratl, P. Neumann, and H.-J. Bungartz. “Towards auto-tuning Multi-Site Molecular Dynamics simulations with AutoPas”. In: *Journal of Computational and Applied Mathematics* 433 (2022), p. 115278. ISSN: 0377-0427. DOI: `https://doi.org/10.1016/j.cam.2023.115278`.

[Ope22] OpenAI. *OpenAI: Introducing ChatGPT*. 2022. URL: `https://openai.com/blog/chatgpt`.

[Ope23] OpenAI. *GPT-4 Technical Report*. 2023. arXiv: `2303.08774 [cs.CL]`.

[Sec+21] S. Seckler, F. Gratl, M. Heinen, J. Vrabec, H.-J. Bungartz, and P. Neumann. “AutoPas in ls1 mardyn: Massively Parallel Particle Simulations with Node-Level Auto-Tuning”. In: *Journal of Computational Science* 50 (2021), p. 101296. DOI: `.https://doi.org/10.1016/j.jocs.2020.101296`.

[Tou+23] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, et al. “Llama: Open and efficient foundation language models (2023)”. In: *arXiv preprint arXiv:2302.13971* (2023).

[Tru22] I. Trummer. “DB-BERT: a Database Tuning Tool that" Reads the Manual"”. In: *Proceedings of the 2022 international conference on management of data*. 2022, pp. 190–203.

[Vas+17] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. “Attention is all you need”. In: *Advances in neural information processing systems* 30 (2017).

[WD97] D. J. Wales and J. P. Doye. “Global Optimization by Basin-Hopping and the Lowest Energy Structures of Lennard-Jones Clusters Containing up to 110 Atoms”. In: *J. Phys. Chem. A* 101 (1997), pp. 5111–5116.

[Wer19] B. van Werkhoven. “Kernel Tuner: A search-optimizing GPU code auto-tuner”. In: *Future Generation Computer Systems* 90 (2019), pp. 347–358.

[WNW21] F.-J. Willemsen, R. van Nieuwpoort, and B. van Werkhoven. “Bayesian Optimization for auto-tuning GPU kernels”. In: *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE. 2021, pp. 106–117.