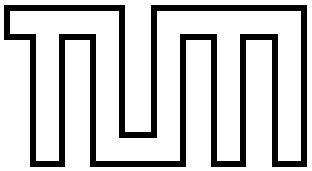


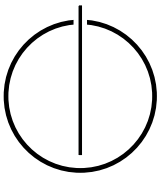
FAKULTÄT FÜR INFORMATIK

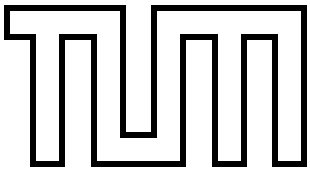
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Using Kokkos to implement a molecular dynamics simulation inspired by AutoPas

Benjamin Decker





FAKULTÄT FÜR INFORMATIK

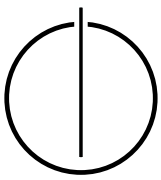
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Using Kokkos to implement a molecular dynamics simulation inspired by AutoPas

Einsatz von Kokkos zur Implementierung einer von AutoPas inspirierten Molekulardynamik-Simulation

Author:	Benjamin Decker
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Fabio Alexander Gratl, M.Sc.
Date:	15.02.2021



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

Munich, 15.02.2021 Benjamin Decker

Abstract

To calculate complex molecular-dynamics simulations efficiently, optimizations in areas like data layout and traversal patterns of the data managed by the simulation have a high significance. Especially, if the simulation is run on a GPU.

This thesis explores how algorithms and data structures that are used in the C++17 molecular-dynamics library AutoPas can be implemented with the Kokkos programming model. The focus is on what has to be considered for the data management in Kokkos, regarding data layout and traversal patterns. The created simulation then enables in-depth performance analyses on different computing platforms.

Zusammenfassung

Um komplexe Molekulardynamiksimulationen effizient berechnen zu können, haben Optimierungen in Bereichen wie Datenlayout und Traversalmuster der von der Simulation verwalteten Daten eine hohe Bedeutung. Besonders, wenn die Simulation auf einer GPU ausgeführt wird.

In dieser Arbeit wird untersucht, wie Algorithmen und Datenstrukturen, die in der C++17-Molekulardynamik-Bibliothek AutoPas verwendet werden, mit dem Kokkos-Programmiermodell implementiert werden können. Der Fokus liegt darauf, was bei der Datenverwaltung in Kokkos, hinsichtlich Datenlayout und Traversalmustern, zu beachten ist. Die erstellte Simulation in ermöglicht dann eingehende Performance-Analysen auf verschiedenen Compute-Plattformen.

Contents

Part I.

Introduction, Background and Related Work

1. Introduction

Many optimization techniques exist to increase the performance of an MD-Simulation. Most of the common optimizing data structures like linked cells complicate a cache efficient implementation. To achieve the best results, configuration items like data layout and traversal patterns have to be tweaked.

Another area of optimization is to delegate the simulation work to a highly parallel compute unit, like a GPU. However, this requires additional considerations due to the way threads are executed by such devices.

The goal of this thesis is to explore how algorithms and data structures that are used in AutoPas, can be implemented with the Kokkos programming model, and to compare the performance of these algorithms on different compute platforms offered by Kokkos.

In order to provide the necessary measurement data, an MD-simulation was written alongside this thesis. The code for this simulation relies heavily on the Kokkos programming Model to provide comparable simulation scenarios, regardless of the computing platform used. In particular, Kokkos enables measuring the performance impact of small changes to the code efficiently, due to only needing one code base for all computing platforms.

2. Molecular-Dynamics-Simulations

MD simulations are used to examine the interaction between particles at the molecular level. For a given set of particles, this is done by calculating the force every particle exerts on every other particle and their resulting change in trajectory in a loop, where each iteration represents a time step of Δt, typically in the order of magnitude of 0.0005 seconds.

2.1. The Lennard-Jones-12-6-Potential

In a simulation, the force that one particle exerts on another particle depends on both particle positions as well as the used potential. The Lennard-Jones 12-6-Potential describes the close-range interaction of electronically neutral atoms or molecules accurately and is therefore used as the particle potential in many MD Simulations. If the distance r between two particles is known, the potential energy of the particle pair, according to the 12-6-Potential, can be calculated[GZK07]:

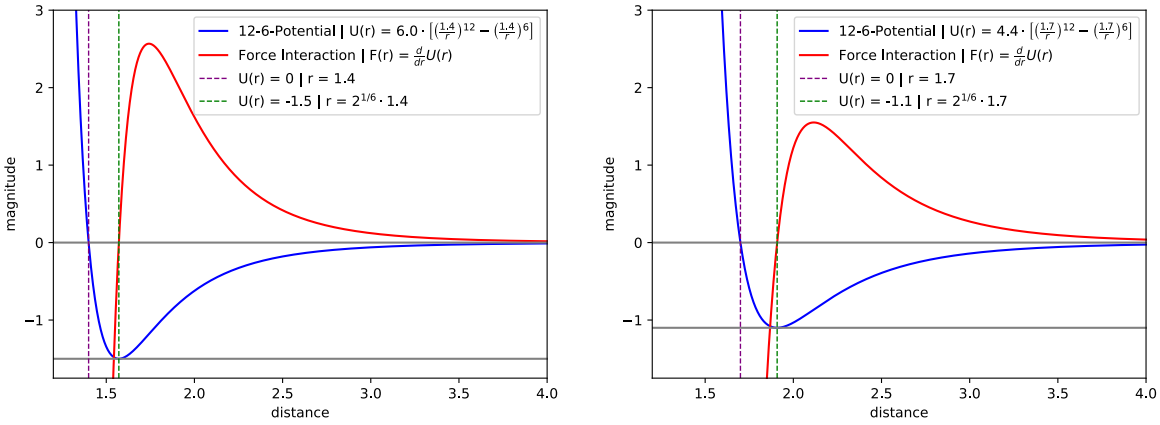
$$U(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \tag{2.1}$$

Figure 2.1 shows two plots of Equation 2.1. In each plot, the blue graph represents the Lennard-Jones-12-6-Potential, and the red graph represents the resulting force on the particles. This force is calculated by taking the derivative of the potential with respect to the distance r. The graphs were plotted with different values for ϵ and σ.

By modifying ϵ and σ, different conditions can be simulated. The lowest potential energy state ϵ, a particle pair can be in is located at a distance of $2^{1/6}\epsilon$. The distance where the potential energy of a particle pair is exactly zero is equal to σ. In Figure 2.1, the lowest potential states are marked with a green line and the zero points of the potentials are marked with a purple line.

By changing the value of the exponents in Equation 2.1, different particle interactions can be modeled. The 12-6-potential represents the most widely used combination. The term with an exponent of 12 describes the Pauli repulsion, the term with an exponent of 6 describes the attractive van der Waals forces. When approaching 0 distance, both terms grow towards infinity. However, the repulsive term grows faster due to its higher exponent, so the total potential grows towards positive infinity when approaching 0 distance. In Figure 2.1, this can be seen as a steep ascend of the potential close to 0. By increasing the distance again, the potential passes σ where the Van-Der-Waals-term outweighs the Pauli-term and the total potential becomes negative. By increasing the distance even further, the potential reaches its global minimum $-\epsilon$, and finally converges towards 0.

The graphs of the force interaction in Figure 2.1, colored in red, illustrate the behavior of two particles based on the 12-6-Lennard-Jones-Potential: Two particles repel each other at very close distance, attract each other at a moderate distance, and only experience a negligible force interaction at large distances.



(a) $\epsilon = 1.5 \quad \sigma = 1.4$ (b) $\epsilon = 1.1 \quad \sigma = 1.7$

Figure 2.1.: The Lennard-Jones-Potential (blue) and resulting force interaction (red) of a particle pair depending on the distance of the two particles. The lowest potential states are marked with a green line and the zero points of the potentials are marked with a purple line.

2.2. Velocity-Störmer-Verlet Algorithm

A force that acts on a particle leads to a change in the particle's position and trajectory. During an interval of time, the behavior of a particle that experiences a constant force can be described with the formulas for the movement of a mass point[Dem18], shown in Figure 2.2. For MD-Simulations, however, the Velocity-Störmer-Verlet-Algorithm formulas, shown in Figure 2.3 are often used instead. Both sets of formulas are mathematically equivalent, but the latter produces numerically fewer rounding errors[GZK07]. In order to use the algorithm, the force each particle in the simulation experienced during its last iteration also has to be saved and updated every iteration, alongside all other particle information.

$$x_i(t^{n+1}) = x_i(t^n) + \Delta t \cdot v_i(t^n)$$
$$v_i(t^{n+1}) = v_i(t^n) + \Delta t \frac{F_i(t^n)}{m_i}$$

Figure 2.2.: The movement of a mass point

$$x_i(t^{n+1}) = x_i(t^n) + \Delta t \cdot v_i(t^n) + (\Delta t)^2 \frac{F_i(t^n)}{2m_i}$$
$$v_i(t^{n+1}) = v_i(t^n) + \Delta t \frac{F_i(t^n) + F_i(t^{n+1})}{2m_i}$$

Figure 2.3.: Velocity-Störmer-Verlet-Algorithm

2.3. The Linked-Cell Algorithm

In each iteration of a simulation, every particle pair has to be considered, and a force has to be calculated for both particles. If the simulation consists of n particles, the running time of the algorithm is in $\mathcal{O}(n^2)$. This means that the time for one iteration grows quickly with the number of particles in the simulation, which makes most simulation scenarios non-feasible.

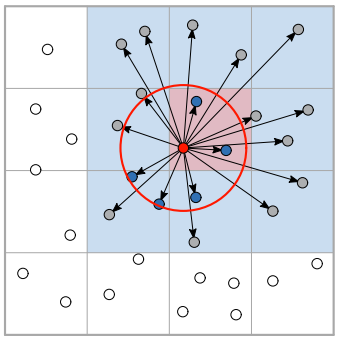


Figure 2.4.: Interaction partners for a single particle using the linked-cell algorithm [GSBN21]

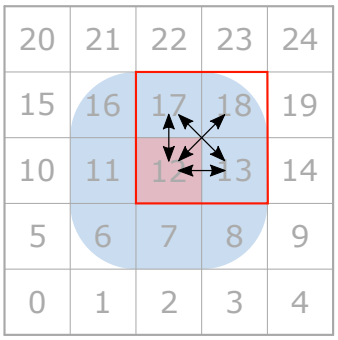


Figure 2.5.: Cells to lock for a c08 base step [GSBN21]

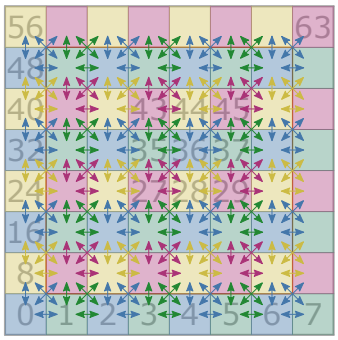


Figure 2.6.: Cell coloring for traversal with c08 base steps [GSBN21]

There are some solutions to this problem, which reduce the simulation time, by skipping calculations that will only have a negligible impact on the simulation. The comparatively small interaction between particles at a large distance from another, based on the Lennard-Jones-Potential (Section 2.1), can be exploited. By only calculating force interactions between particles that are closer together than some large enough cutoff radius, and considering all other interactions negligible, the necessary calculations per time step can be reduced, which increases performance.

The linked-cell algorithm is an efficient implementation of such an optimization. The algorithm subdivides the simulation space into cube-shaped cells with a side length equal to or greater than the cutoff radius. This guarantees that every two particles that are closer together than the cutoff radius will be either inside the same cell or in neighboring cells. In 3-dimensions, each cell has 26 neighbors. By only considering particles inside one of those 26 cells or the own cell, many of the unnecessary force calculations are trivially excluded. If there is an upper bound m on the number of particles inside a cell, the amount of necessary calculations per time step is bounded by $n \cdot 27m$, resulting in a running time in $\mathcal{O}(n)$. This leads to a lower calculation time for a simulation with a large number of particles n and a small upper bound on the cell size m.

A two-dimensional example of the algorithm can be seen in Figure 2.4. The radius of the red circle is equal to the cutoff radius. The side length of the cells is a bit larger than the cutoff radius. The force acting on the red particle is calculated by iterating over every particle in the same cell and all neighbor cells, marked in red and blue, respectively. A force calculation is only done for the blue particles, as they are closer than the cutoff radius. The force calculation for the gray particles is skipped. The white particles are not located in either the red cell or in one of the blue cells and can therefore be ignored.

2.3.1. Newton3 Optimization

Newton's third law of motion states, that every force one particle exerts on another particle leads to an equal and opposite force the other particle exerts on it[Dem18]. For a simulation, this means that, for each particle pair in the simulation, only one force has to be calculated and added to the total forces of both particles respectively.

Implementing Newton3 optimizations halves the necessary calculations per time step, therefore improving performance. When using this optimization, however, the cells have to be traversed in a way that keeps track of already calculated forces while also preventing race conditions by avoiding simultaneous write operations by two threads to the same data.

C08-base-step-algorithm

The c08-base-step-algorithm describes a traversal pattern for Newton3 optimizations. The algorithm works by iterating over every cell and calculating one base step. A base step consists of first calculating the force interactions between all particles inside the current cell with another. Next, force interactions between the current cell and some of the forward neighbors of the current cell are calculated. Finally, interactions between some of the neighboring cells themselves are also calculated. The c08-base-step minimizes the number of cells included in one step and offers good cache efficiency compared to other base steps. A two-dimensional example of this is shown in Figure 2.5. By iterating over every cell and applying the c08-base-step, all particle pairs with a distance closer than the cutoff radius will be considered exactly once across all iterations.

If the c08-base-steps on all cells are calculated in parallel, race conditions can appear when two threads update the information of the same particle at the same time. This can happen for any two c08-base-steps on cells that are directly next to each other, as there will be some cells that have to be written to by both base steps. To prevent this, the cells are divided into groups so that a base step on a cell in one group will not have any overlapping cells with any other base step from the same group. The different groups are commonly referred to as different colors. At least eight colors are needed to correctly divide the cells this way. A two-dimensional example of a correct coloring is depicted in Figure 2.6. In the example, no two neighboring cells have the same color. To extend the example into three dimensions, the next layer of cells on top of the shown cells is colored the same way with, four additional colors. This pattern can be extended to an arbitrary number of cells.

After coloring the cells, each base step of one color can be calculated in parallel without the risk of race conditions appearing. Only after all steps of one color are calculated, the iteration for the next color can be started.

2.3.2. Boundary conditions

Boundary conditions determine what happens to particles at the boundaries of the simulation space. In a simulation that implements the linked-cell algorithm, this is usually handled by a layer of halo cells, which enclose all other cells. Depending on the boundary condition used, the particles will behave differently close to those halo cells.

Outflowing Boundaries Particles are removed from the simulation when they enter a halo cell. No additional calculation is required.

Periodic Boundaries Particles that move into a halo cell are instead moved into a normal cell on the opposite side of the simulation space. Periodic boundaries can be used to approximate an infinite simulation space and an arbitrary particle density with a finite amount of cells and particles.

Reflecting Boundaries Particles that come close to a halo cell experience a repulsive force away from the cell. The halo cells act as walls inside of the simulation and prevent particles from moving into them.

3. Kokkos

Kokkos implements a programming model in C++ for writing performance portable applications targeting all major HPC platforms. For that purpose, it provides abstractions for parallel execution of code and for data management[Tro18]. Code that was written with Kokkos can be compiled to run on any of a set of different platforms, which enables easy and comparable performance measurements of parallel execution between those platforms.

For this purpose, Kokkos offers structures to allocate and keep track of data in different memory spaces, functions for dispatching parallel jobs that can access this data, and methods for tweaking memory layout and access patterns to improve performance on different platforms.

3.1. View

For allocating and accessing data in any available memory-space, Kokkos offers the Kokkos::View class. A view represents an array of zero or more dimensions, which can store any primitive data type or struct of primitive data types.

The data type of a view must be configured via a template parameter. Optional additional parameters may be specified to configure the layout type and memory space of the view's data. If a parameter is not specified, default values are used. When a view is created, the necessary memory allocations are executed from host-space. Afterward, the view's data is filled by the specified device by evoking the default constructor of the specified data type for each element it contains.

```
1 using namespace Kokkos;
2 View<int *, LayoutLeft, CudaSpace> myView("my view", 100);
```

Listing 3.1: A one-dimensional Kokkos::View with coalesced memory layout in CudaSpace

The view in Listing 3.1 represents a one-dimensional int-array of size 100. The string argument acts as a label to distinguishing the view from other views for debugging purposes. The memory layout and memory space of the view are set to Kokkos::LayoutLeft and Kokkos::CudaSpace, respectively. The choice of these parameters can have a severe impact on cache efficiency and therefore performance.

3.1.1. Memory Space

A GPU can not access the system memory directly and has its memory built-in. For a GPU to be able to access any data, this data has to be allocated in its own memory, also called device-memory. A view always exists in CPU-memory, but the allocated memory it represents will be allocated in the memory space it was configured for. If a view is created without specifying a memory space, the data it represents will be allocated in the default

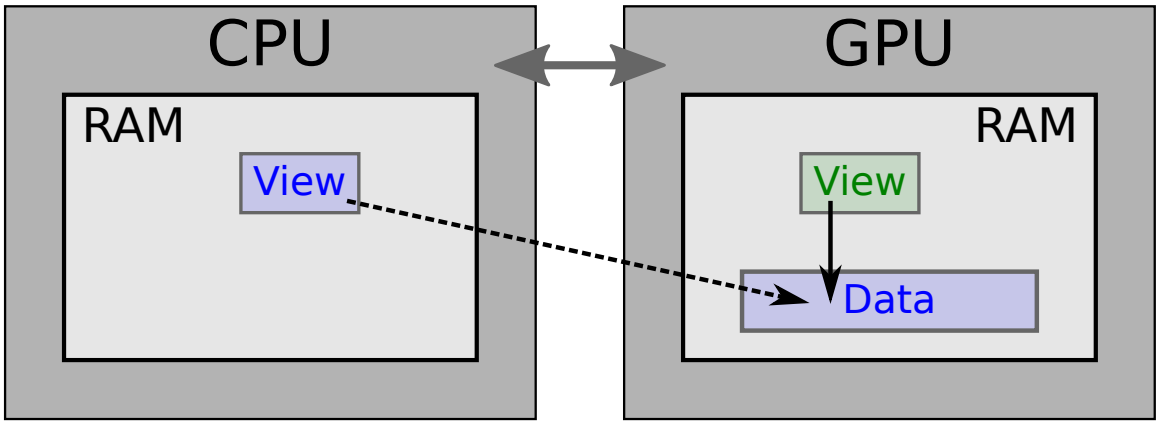


Figure 3.1.: The view in host-space and its data in device space, marked in blue, are both allocated when the view is created and persist in memory until the view is deleted. The view knows the location of the data in device space, but it can't be accessed across the device boundary. In order for the GPU to access the data, a temporary copy of the view, marked in green, has to be created in device space.

memory space. The default memory space can be configured via CMake parameters at compile time.

A view itself is a lightweight object that only contains configuration data and a pointer to its allocated memory. If the view lies in a different memory space than its data, it has to be copied into the same memory space when a parallel execution is dispatched in order to allow the corresponding device to access it, as described in Subsection 3.2.2 and Subsection 3.2.3. This process is illustrated in Figure 3.1.

Some operations on the allocated memory of a view may only be possible from host-space, so in some scenarios, the allocated data may have to be copied from device space into host space and back, which can take a long time. All copy operations between different memory spaces have an increased latency, due to the way a GPU is connected to the rest of a system. Additionally, the allocated memory of a view can be very large, slowing down copy operations even more. During performance-critical sections of a program, it is therefore important to rely on the own memory space as much as possible.

CudaUVMSpace

For rare cases, when access to a view's data must be possible from host-space and CUDA device-space, the memory space of a view can be set to Kokkos::CudaUVMSpace. UVM stands for NVIDIA's unified memory technology and is only available on CUDA capable GPUs[1]. UVM behaves like a memory space that is shared between host and device, while the data is still allocated and stored in device-memory. If data is accessed from host-space, the GPU driver detects it as a page fault and copies the necessary data between the memory spaces automatically. The same happens if the GPU accesses data that was modified by the host.

UVM space simplifies working on the same data from two execution spaces, however, the automatic page migration of the GPU driver comes at the cost of potentially decreased

[1] `https://developer.nvidia.com/blog/unified-memory-cuda-beginners/`

memory latency and throughput[CPM19], and should only be used for non-performance-critical use cases or when it can not be avoided.

3.1.2. Memory Layout

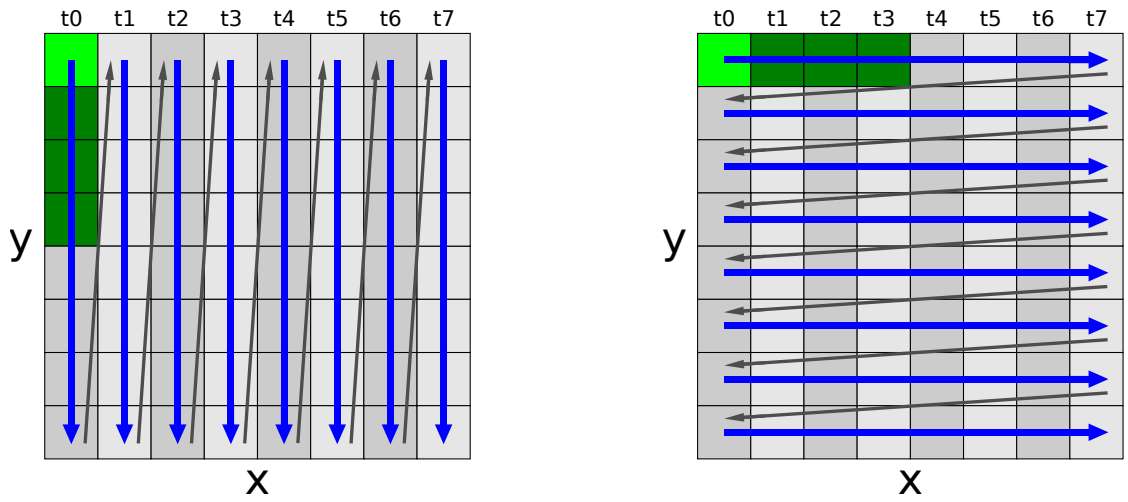
Each memory access to an address that is not already cached leads to a cache miss and a subsequent loading of a cache line with a block of data, containing the specified memory address. As long as the block remains cached, subsequent accesses to addresses inside the block are much faster, compared to the initial one. However, cache memory is small compared to the main memory, so if too much data is cached and the cache becomes full, each new memory access will override a cache block, negating the speed advantage for the overwritten memory addresses. The way a view's data is laid out in memory can have a severe impact on transfer speeds, depending on which device is accessing its data. On a CPU, each thread has its own cache space[Zwi11]. It is therefore important that all required data for one thread is saved close together in memory, which increases the chance of important data to be loaded inside the same cache line. If each thread is assigned an index of a view to work on, then all data corresponding to one index should be saved sequentially and not overlap with any data from the next index.

On a GPU using CUDA, on the other hand, groups of threads are executed together in so-called warps and share a common warp cache. Warp threads are synchronized, which means that they share the same series of instructions and perform them at the same time, with different parameters. Therefore, the ideal way to layout data for cache efficiency on CUDA GPUs is to save all data that is accessed by the warp threads at the same time close together.

In order to take full advantage of caching, the sequence in which memory is accessed by each thread has to be known in advance to configure the memory layout accordingly, depending on the computing platform. Kokkos offers methods to parallelize over a range of indices with a similar syntax to C++ for-loops, which is described in Subsection 3.2.3. Assuming that memory accesses to a multidimensional view are parallelized over the leftmost index of the view, meaning that the value of the leftmost index is dependent on the loop index parameter a thread is given, then this memory access sequence can be predicted and a data layout pattern can be chosen accordingly.

Views in Kokkos choose their layout pattern automatically, depending on their memory space. Kokkos::LayoutRight is the default layout pattern for CPU memory, which applies a row-major mapping, saving all entries with the same leftmost coordinate close together in memory, allowing efficient caching for each thread. On a GPU, the default layout pattern is Kokkos::LayoutLeft, which uses a column-major mapping, where all entries with the same rightmost coordinate are saved close together in memory. Due to the synchronous execution of all warp threads during a memory access, all threads will access entries with the same y-coordinate, which are saved sequentially, improving cache efficiency.

The way both memory layouts are applied to a two-dimensional view is shown in Figure 3.2. The default memory layout in Kokkos is set to optimize performance for the scenario that the threads that access the view are parallelized over the leftmost index of the view. The memory layout of a view can also be set manually, as shown in Listing 3.1.



(a) Kokkos::LayoutRight (b) Kokkos::LayoutLeft

Figure 3.2.: Two different memory layouts are shown, applied to a two-dimensional view in Kokkos, where x is the leftmost index. The path shown by the blue and gray arrows illustrates how the entries are mapped to the underlying one-dimensional array in memory. The entries along a blue arrow are saved sequentially. The first entry of one blue arrow is mapped immediately after the last entry of another, as indicated by the gray arrows. In this example, the cache line size is equal to four memory entries. When thread t0 accesses its first memory entry, colored in light green, its whole cache line is loaded with three subsequent entries, colored in dark green. With Kokkos::LayoutRight on a CPU, t0 has cached some of the entries it will need in the future. With Kokkos::LayoutLeft on a GPU, t0 has loaded some of the entries the other threads need into the shared warp cache.

3.2. Parallel Execution

In Kokkos, a parallel job, representing work that is currently being executed by several threads is called a kernel. Kokkos offers several classes to configure the behavior of such kernels and several methods for launching different types of kernels.

3.2.1. Execution Policies

An execution policy defines the number of jobs that should be parallelized and how to distribute them to the available threads. Kokkos offers three of these policies.

RangePolicy Each iterate is an integer in a contiguous range.

MDRangePolicy Multiple intervals can be specified. Suitable to iterate over a grid or a similar structure.

TeamPolicy For each index in a specified contiguous range, a team of multiple threads is created to work on it.

```
1 using namespace Kokkos;
2 int numCells = 100;
3 RangePolicy<Schedule<Dynamic>> myPolicy(0, numCells);
```

Listing 3.2: Kokkos::RangePolicy

Listing 3.2 shows the construction of a Kokkos::RangePolicy. The runtime arguments define the interval of indices to parallelize over, in this case [0, 99]. Each policy can be further configured via several template parameters. In this case, the use of dynamic scheduling is enabled, to re-balance the workload if some threads are completing their jobs faster than others. This is especially useful if there is a high variance in the workload for each index.

3.2.2. Functor Types

The code that should be executed by a parallel kernel in Kokkos needs to be specified by a functor type.

Functors

A possible way to define this code is by a functor object, which is an instance of a class or struct with an ()-operator.

```
1 class MyFunctor {
2     public:
3         Kokkos::View<int *> numbers;
4         MyFunctor(Kokkos::View<int *> numbers) : numbers(numbers) {}
5         void operator() (int i) const {
6             numbers(i) = i;
7         }
8 }
```

Listing 3.3: A Functor

Listing 3.3 shows a simple example of a functor, writing data to a Kokkos::View. The operator writes the value of i into the view numbers at index i. The code inside the operator function can only access member variables, so numbers has to be copied into the functor object by the constructor. When a kernel with a functor is started, the whole functor object is copied into device memory, to make all variables accessible by the respective device.

Lambda Expressions

Functors are not the only way to define code for a parallel kernel. Lambda expressions are a more compact way to pass code to a kernel, which may improve code readability, if the amount of code to be executed is small.

```
1 Kokkos::View<int *> numbers("numbers", 100);
2 auto myLambda = KOKKOS_LAMBDA (int i) {
3     numbers(i) = i;
4 };
```

Listing 3.4: A Lambda Expression in Kokkos

The macro KOKKOS_LAMBDA in Listing 3.4 expands to [=], configuring the lambda to capture all data that is accessed from inside it by value. If the code is compiled with CUDA enabled, the macro instead expands to [=] __host__ __device__, additionally marking the lambda executable from the GPU. Defining a lambda like this automatically takes care of copying the necessary data, in this case numbers, into device memory, when a parallel kernel is started.

3.2.3. Kokkos::parallel_for

When an execution policy and a functor type is given, parallel execution can be started via a Kokkos::parallel_for().

```
1 using namespace Kokkos;
2 void parallel_for(
3     const std::string& str,
4     const ExecPolicy& policy,
5     const FunctorType& functor
6 );
```

Listing 3.5: Kokkos::parallel_for

The string argument in Listing 3.5 acts as a label for the parallel kernel, distinguishing it from other parallel kernels for debugging purposes.

When the Kokkos::parallel_for() is called, all necessary data for the functor type is copied into the specified memory space and several threads with different index parameters, as defined by the execution policy, are started and begin executing the code specified by the functor type.

4. Related Work

MD-simulations are a well-known area of scientific computing and a large research effort is still being invested into it. As a result, there is a lot of scientific work, discussing topics that are similar to this thesis, which offers a great opportunity to take inspiration from and build on. Two such projects are worth mentioning in particular.

4.1. AutoPas

AutoPas is an open-source C++17 library delivering optimal node-level performance by providing the ideal algorithmic configuration for an arbitrary scenario in a given short-range particle simulation[GSBN21]. The project stands out due to the automatic real-time tuning of its configuration items, which include traversal patterns and data layout, to improve performance.

The automatic tuner regularly decides the optimal configuration by measuring the simulation performance during an iteration and comparing it to previous iterations. The new configuration is then applied by using the factory pattern to prepare containers, traversal patterns, and data layout. In order to be able to iterate over containers with variable data layout in a variable traversal pattern, AutoPas makes use of the facade pattern.

As mentioned in Chapter 1, this thesis explains, amongst other things, how the code from AutoPas could make use of the Kokkos programming model. To achieve this, the code written alongside this thesis takes a lot of inspiration from AutoPas. Optimizations like the linked-cell algorithm with Newton3 optimization, as described in Section 2.3 and Subsection 2.3.1, as well as halo cells to simulate boundary conditions, explained in Subsection 2.3.2 are used in both implementations. However, across both implementations, other features like auto-tuning or optimizations verlet-lists are unique to AutoPas.

For further details, refer to a paper about the project[GSBN21] or its GitHub page[1].

4.2. MiniMD

MiniMD is a C++ parallel molecular dynamics library, using Kokkos. A lot of algorithms similar to AutoPas were implemented in this library, making it a great assistance and inspiration in writing the code for this thesis.

The library's code is hosted on GitHub[2].

[1] `https://github.com/AutoPas/AutoPas`
[2] `https://github.com/Mantevo/miniMD`

Part II.

Implementation

5. Overview

To provide realistic performance measurements, a requirement for the simulation code was to be able to perform real calculations and produce usable results. Therefore, implementation decisions were made to create a working high-performance MD-simulation inspired by real-world examples like AutoPas or MiniMD, mentioned in Chapter 4. Each force calculation is done based on the 12-6-Lennard-Jones-potential, explained in Section 2.1. To improve the overall performance, the linked-cell algorithm, in conjunction with Newton3-optimization, as described in Section 2.3 and Subsection 2.3.1, is used.

Over the course of writing the simulation code, it went through many iterations and changes. This part of the thesis explains the implementation choices that were made and provides an overview of how the simulation is structured.

6. Data Management

Most of the data the simulation manages is saved using the Kokkos::View data structure, introduced in Section 3.1, which represents an array of zero or more dimensions. In order to work with views, custom data types had to be created to store particle information, and special consideration had to be given to the data layout used.

6.1. Coord3D

Each particle has properties that need to be saved, some of which are represented by 3-dimensional vectors. A view can only store primitive data types or structs/classes of primitive data types, so in order to save such vectors inside a view, a class that can hold three double values was implemented.

```
1 class Coord3D {
2  public:
3   double x, y, z;
4 }
```

Listing 6.1: Coord3D

To simplify the use of the Coord3D class, some arithmetic vector operations, such as additions, subtraction, and scalar multiplication were implemented as class operators.

```
1  Coord3D operator+(const Coord3D &rhs) const {
2      return Coord3D{x + rhs.x, y + rhs.y, z + rhs.z};
3  }
4
5  Coord3D operator-(const Coord3D &rhs) const {
6      return Coord3D{x - rhs.x, y - rhs.y, z - rhs.z};
7  }
8
9  Coord3D operator*(double rhs) const {
10     return Coord3D{x * rhs, y * rhs, z * rhs};
11 }
```

Listing 6.2: Coord3D operators

6.2. Particle

Every particle in the simulation has a set of properties that have to be saved, which are shown in Listing 6.3. To distinguish particles from another and to assign them into groups of similar particles, each particle is given a particleID and a typeID. The remaining data fields for each particle are represented as 3-dimensional vectors and are saved as Coord3D objects, introduced in Section 6.1. The fields include position, velocity, and experienced force. In addition to

the position, velocity, and force of each particle, the force that each particle experienced in the previous iteration is also saved, to be able to use the Velocity-Verlet-Störmer algorithm, described in Section 2.2, and reduce rounding errors.

```
1 class Particle {
2  public:
3   int particleID;
4   int typeID;
5   Coord3D position;
6   Coord3D velocity;
7   Coord3D force;
8   Coord3D oldForce;
9 }
```

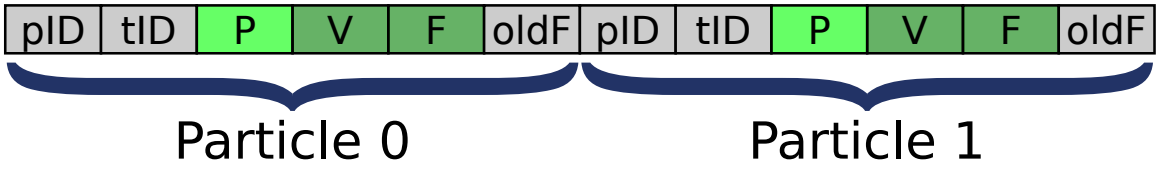
Listing 6.3: A particle

6.3. AoS and SoA

In Subsection 3.1.2, the importance of cache-aware programming was introduced. For optimal performance, data should always be laid out in memory in a way to optimize cache efficiency. Cache efficiency is a measure of how much a thread can make use of its cache. For a given memory access sequence, cache efficiency is high if the amount of cache misses and therefore cache loads is low, resulting in better performance.

To improve cache efficiency, the Particle class from Section 6.2 is only used to work on particle information in host space and is not saved inside of views. Doing so would mean that two data entries of the same type from different particles would always be at least as far apart from another in memory as the memory size of the entire particle. Such a layout, representing an array of structures, is called an AoS layout. A parallel kernel, accessing such a view might only want to access some of the information of each particle, skipping the entries in between, which results in a low cache efficiency.

Instead of saving the particle data this way, each particle property is saved together with data of the same type in a structure of arrays layout, also called SoA layout. The amount of cache misses with an SoA layout is lower than with an AoS layout, which results in a higher cache efficiency. This circumstance is illustrated and explained in Figure 6.1.



(a) AoS

(b) SoA

Figure 6.1.: Both figures show sections of different views. The view in (a) saves Particle objects and contains the information of two particles, the view in (b) only saves the positions of particles in form of Coord3D objects. A thread accesses all position values, marked as P. The cache line size is assumed to be large enough to hold three entries. When a position value is accessed, some memory following that entry is loaded into the thread's cache. The view in (a) contains all information of one particle grouped in an array-of-structures data layout, also called AoS. For each access to a position value, no useful additional information was cached so the cache efficiency is low. The view in (b) shows a single view in a structure-of-arrays data layout, also called SoA. The view only contains the positions of all particles, so for each access to a position, other position values can be cached, increasing cache efficiency.

7. Linked Cells

Implementing the linked-cell algorithm, as described in Section 2.3, has a lot of impact on the data structure of a simulation. Particles from different cells have to be saved in different views. If a particle moves from one cell into another, it also has to be moved in memory. The sizes and capacities of each cell have to be managed and the cells have to be resized, if necessary.

Over the course of implementing a linked cell data structure, two versions with different approaches to this algorithm were implemented. Both versions have their advantages and disadvantages, which will be discussed in this chapter.

7.1. Views inside of Views

The version that was implemented first uses nested views for it's data structure. The first step to this implementation was to create the inner views that contain the particle information for each cell, which are saved in aa SoA data layout, to improve cache efficiency.

```
1 Kokkos::View<int *> particleIDs;
2 Kokkos::View<int *> typeIDs;
3 Kokkos::View<Coord3D *> positions;
4 Kokkos::View<Coord3D *> velocities;
5 Kokkos::View<Coord3D *> forces;
6 Kokkos::View<Coord3D *> oldForces;
```

Listing 7.1: The particle information saved by one cell

Next, the inner views have to be saved in an outer data structure. Saving the views inside a large array or vector is not feasible, as the whole array would have to be copied for every thread in each parallel kernel, so the views have to be saved inside another view. However, as mentioned in Section 3.1, a view can only contain primitive data types or structs/classes of primitive data types.

Saving other types inside of views is not allowed, as this could lead to illegal memory allocations. After creating a view, its data is filled by evoking the default constructor from device-space for each element it contains. These constructors may contain additional memory allocations. It is not possible to execute data allocations from a GPU, so if the memory space of the view is set to a GPU, creating the view will fail.

However, this problem can be avoided, and the restriction can be bypassed. The GitHub page of the Kokkos project describes a way to create nested views with a CUDA capable GPU[1]. The procedure is summarized in Listing 7.2, which shows an outer view outer, that contains 5 inner views, which in turn contain 4 doubles.

[1] https://github.com/kokkos/kokkos/wiki/View#623-can-i-make-a-view-of-views

```
using namespace Kokkos;
using inner_view_type = View<double*, CudaSpace>;
using outer_view_type = View<inner_view_type*, CudaUVMSpace>;
const int numOuter = 5;
const int numInner = 4;

for (int k = 0; k < numOuter; ++k) {
    outer_view_type outer(view_alloc("Outer", WithoutInitializing), numOuter);
    new (&outer[k]) inner_view_type(view_alloc("Inner"), numInner);
}
```

Listing 7.2: A View inside of another View

The outer view must be accessible from device-space in order to access the inner views from the device, but also be accessible from host-space to create the inner views and perform the necessary memory allocations. On a CUDA capable GPU, this is possible by allocating the outer view in CudaUVMSpace, mentioned in Section 3.1.1. To prevent allocating memory from device space, the WithoutInitializing parameter is specified when creating the outer view, which disables the automatic filling of the views data, and therefore the automatic evocation of the inner views constructors. The inner views can then be created from host-space at the correct memory location, pointed to by the outer view.

Listing 7.3 shows the resulting data structure. For each particle, an outer view was created, containing a number of views equal to the number of cells. Additionally, the cellSizes and cellCapacities views are created to save the number of particles and capacities per cell. In order for the simulation to also work if no device-space is configured, the memory space of the outer view is set by preprocessor directives depending on which memory space was specified at compile time.

```
#ifdef KOKKOS_ENABLE_CUDA
#define SharedSpace Kokkos::CudaUVMSpace
#else
#define SharedSpace Kokkos::DefaultExecutionSpace

typedef Kokkos::View<Coord3D *> InnerCoordViewType;
typedef Kokkos::View<int *> InnerIntViewType;
typedef Kokkos::View<InnerCoordViewType *, SharedSpace> CoordViewType;
typedef Kokkos::View<InnerIntViewType *, SharedSpace> IntViewType;

CoordViewType positions;
CoordViewType forces;
CoordViewType oldForces;
CoordViewType velocities;
IntViewType typeIDs;
IntViewType particleIDs;
Kokkos::View<int *> cellSizes;
Kokkos::View<int *> cellCapacities;
```

Listing 7.3: Views inside of Views representing a linked-cell memory layout

This implementation offers flexibility on how to manage different cells. The cells can be resized independently from another and different data layout patterns can be used. A problem with this implementation is that saving views inside of views with a GPU device-space is only possible with CUDA-capable GPUs. There is no equivalent to CudaUVMSpace

for other GPU platforms offered by Kokkos. Another issue is the long initialization time. For each cell in the simulation, six views have to be created, allocated, and filled with particles, which takes a long time for simulations with a realistic size. To overcome these issues, another version of a linked-cell algorithm was implemented.

7.2. Two-dimensional Views

The intention of writing this version was to stop using CudaUVMSpace, as it was presumed to be the reason for the weak CUDA-performance of the first implementation, and to reduce the number of views used in the simulation, in order to reduce the initialization time.

```
1 Kokkos::View<Coord3D **> positions;
2 Kokkos::View<Coord3D **> forces;
3 Kokkos::View<Coord3D **> oldForces;
4 Kokkos::View<Coord3D **> velocities;
5 Kokkos::View<int **> typeIDs;
6 Kokkos::View<int **> particleIDs;
7 Kokkos::View<int *> cellSizes;
8 Kokkos::View<int> capacity;
```

Listing 7.4: Two-dimensional views saving particle information

Listing 7.4 shows the implemented data structure, which uses two-dimensional views to save each of the particle properties in the simulation. For each of the views, the first index corresponds to the cell number and the second corresponds to the particle number in the specified cell. The cellSizes view saves the amount of particles per cell. capacity saves the current cell capacity.

For a view in Kokkos, it is not possible to allocate a different number of entries for different rows or columns of the view. For this implementation, this means that the capacity of each cell is equal to the highest required capacity, across all cells, resulting in high memory usage. The amount of unused allocated memory is proportional to the homogeneity of the particle distribution. If most of the particles are located in one or a few cells, this effect is the greatest.

8. The Simulation Loop

The simulation calculates the movement of each particle for each time step in a loop. During each time step, a particle is assumed to follow a uniform rectilinear movement and to only experience an instantaneous acceleration in between the time steps. To improve performance, the linked-cell algorithm in combination with Newton3-c08 base steps, as described in Section 2.3 and Subsection 2.3.1, was implemented.

Algorithm 1: Simulation Loop

Input: numIterations

```
1 Function doSimulation(numIterations):
2     for iteration ← 0 to numIterations do
3         calculateForces()
4         calculateVelocitiesAndPositions()
5         moveParticles()
```

Algorithm 1 describes the simulation loop. In each time step, the force exerted on every particle and their resulting change in trajectory is calculated. If a particle moves from one cell to another during one iteration, the particle data also has to be moved in memory, which is done at the end of each iteration.

8.1. calculateForces()

Implementing a linked-cell algorithm with newton3-c08-optimization has a severe impact on cell traversal patterns. To avoid race conditions, all cells have to be assigned one of eight colors, following the c08-coloring scheme, as described in Subsection 2.3.1.

Algorithm 2: Force Calculation

```
1 Function calculateForces():
2     saveOldForces()
3     for iteration ← 0 to 7 do
4         calculateBaseStepsInParallel(color)
```

Before any new forces are calculated, the old forces have to be saved, to be able to use the Velocity-Verlet-Störmer algorithm for calculating positions and velocities, as described in Section 2.2. Afterward, all base steps corresponding to cells with the same color can be executed in parallel. Only if all base steps for one color are completed, the calculation for the next color can be started.

For each color, a Kokkos:: parallel_for , as described in Subsection 3.2.3, is executed to run the necessary calculations in parallel.

8.2. calculateVelocitiesAndPositions()

After the new forces are known, the resulting velocities and positions can be calculated. A Kokkos:: parallel_for is used to iterate over all cells and one thread per cell calculates the new particle positions and trajectories based on the Velocity-Störmer-Verlet formulas, introduced in Section 2.2.

8.3. moveParticles()

Each particle that moved from one cell into another during one iteration also has to be moved to the new cell in memory. This is done by iterating over every particle and checking if the cell it should be in, based on its position, matches the cell it is saved in. If necessary the particle is removed from one cell and inserted into the other.

Part III.

Results

9. Overview

All performance measurements were run on a system with an AMD Ryzen 7 3800X 8-Core Processor, with 32GB DDR4 memory and an NVIDIA GTX 1060 GPU with 6GB GDDR5 memory. The theoretical single precision performance for both devices is approximately 650 GFLOPS[1] for the CPU and 4.375 GFLOPS[2] for the GPU.

For measurements on different platforms, the Kokkos::DefaultExecutionSpace has to be set correctly when compiling the code. For GPU measurements, it was set to Kokkos::Cuda, for CPU measurements it was set to Kokkos::OpenMP, with the number of maximum OMP threads set to 16.

In each of the tests, the performance of OpenMP and CUDA is compared by running simulations with identical configurations on both platforms. Configuration items include data structure, data layout, traversal patterns, and the number and distribution of particles in the simulation.

The implemented versions of the linked-cell algorithm are described in Section 2.3. The time step size is 0 for all calculations to prevent particles from moving, which would otherwise lead to particles escaping the simulation space. All calculations were done using single-precision floating-point numbers.

The performance results for all comparisons were created by running the configured simulation for a fixed number of iterations and measuring the time it takes for the calculation to complete. The average time per iteration for each platform can then be compared. A lower time per iteration means better performance.

[1] https://gadgetversus.com/processor/amd-ryzen-7-3800x-gflops-performance/
[2] https://www.techpowerup.com/gpu-specs/geforce-gtx-1060-6-gb.c2862

10. Measurements

10.1. No Cells

The first measurement, plotted in Figure 10.1, does not use any linked-cell implementation. All particle information is saved in an SoA layout. Each thread has to access all elements of a few large views sequentially in each iteration, which means that the entries can be cached very efficiently. Additionally, the partner particles for the force calculation are the same for each particle, so the GPU threads can work together efficiently, without having to mask out some of their instructions. The efficient use of the GPU's resources is reflected in a lower running time on CUDA, compared to OMP.

Due to the fact that all particle pairs have to be considered in this measurement, the running time on both platforms is in $\mathcal{O}(n^2)$, where n is the number of particles. The quadratic running time is illustrated by the gray parabola.

10.2. Views inside of Views vs. Two-dimensional Views

The two implemented versions of the linked-cell algorithm show significantly different initialization times and iteration times, depending on the platform. The speedup of using two-dimensional views over nested views is plotted in Figure 10.2. The consequences of swapping nested views with two-dimensional views are explained in the following.

The initialization time is much faster on both platforms, with a greater difference on CUDA. The running times don't give such a conclusive image. With CUDA, performance is consistently better with a significant difference for a high number of particles. With OpenMP, performance for a small number of particles, up to ca. $2 * 10^4$, are slightly worse. For larger numbers of particles, the ratio converges to 1.

The inner views in the nested-view implementation are allocated independently from another. This leads to many small data allocations, scattered across different, non-sequential, memory addresses. In order to use efficient caching on a GPU, the accessed memory has to be laid out in a specific way, described in Subsection 3.1.2. The scattered data layout leads to many cache misses and performance penalties on a GPU, and most likely explain the high iteration times for nested views.

A CPU thread, on the other hand, only needs to load the data of the cell it is working on into cache, which is still saved sequentially in both implementations, which leads to an overall high cache efficiency on OpenMP. However, in two-dimensional views, every cell has the same capacity as the largest cell, so a lot of unused memory space is allocated in each view. This makes caching useful information beyond the current cell unlikely. With nested views, the cell views are a lot smaller and if another cell is located next to the current cell in memory, some of its information may also be cached by chance, which could explain the slightly higher performance with nested views on OpenMP.

10.3. CUDA vs. OpenMP

For all of the following measurements, the two-dimensional-views-implementation was used, due to its lower initialization time

10.3.1. One Particle per Cell

Just one particle is placed in each cell of the simulation, the running times are plotted in Figure 10.3. The workload per cell is the same for every cell in the simulation. The capacity for each cell in a two-dimensional view is exactly one. Similar to the situation in Section 10.1, this leads to a very cache efficient memory layout, which is reflected in the graphs. In general, performance is better on CUDA, with an approximately constant ratio between the running times of CUDA and OMP. Because of the linked-cell algorithm, both running times grow linearly with the number of particles, represented as two straight lines.

10.3.2. Variable Homogeneity

Particles were distributed according to a three-dimensional Gaussian distribution with a variable standard deviation. The data is plotted in Figure 10.5. OMP shows an overall better performance than CUDA. Performance sinks on both platforms when the standard deviation is increased. For small values of the standard deviation, performance on OMP is still up to 17 times better than on CUDA, however, the running time ratio quickly approaches 1, when the standard deviation is increased.

The lower the standard deviation, the more particles are located closer to the center of the simulation space, resulting in a low homogeneity. In such scenarios, the difference in size between the smallest and largest cell also rises. The workload of the simulation is only parallelized over the c08-base steps for each cell, and each base step is calculated by a single thread. This means that the running time to calculate all base steps of one color is at least as long as the time to calculate just the base step with the most particles. Additionally, if this largest base step is only scheduled when almost all other base steps have already been calculated, this effect is aggravated even more.

The high difference between CUDA and OMP for low standard deviations can be explained by the higher clock frequency of a CPU, compared to a GPU. A CPU thread will finish a given base step quicker than a GPU thread, so the time to finish the largest base step will be as well. This also lowers the mentioned lower bound for the running time of one color, and therefore the running time of the whole iteration.

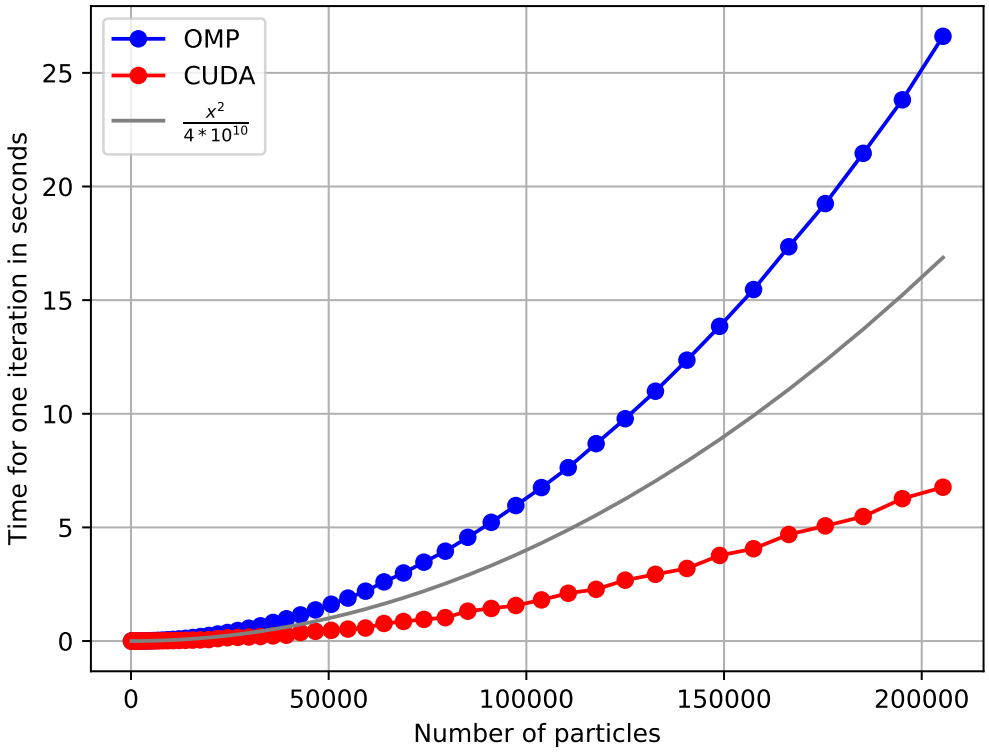
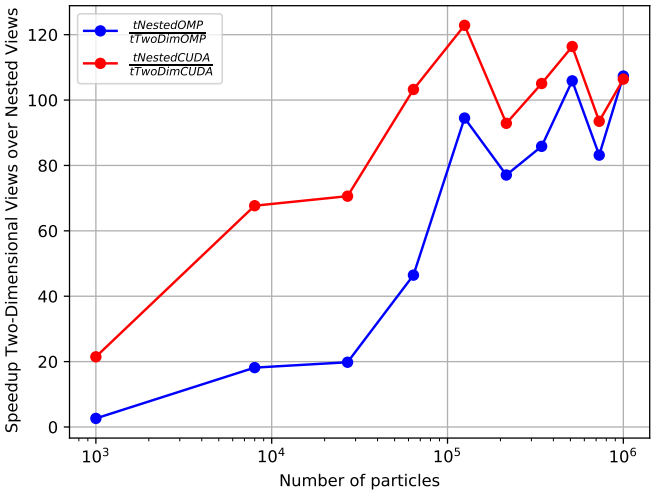
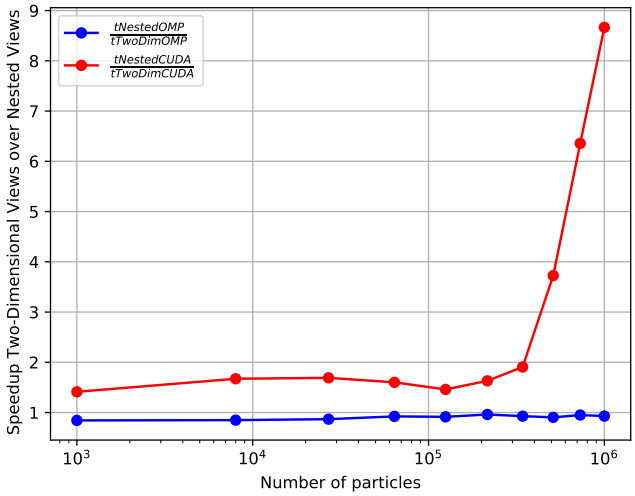


Figure 10.1.: One-dimensional views were used to save particle information in an SoA layout, as described in Section 6.3. No linked-cell or cutoff-radius optimization was used, so the force interaction for every particle pair in the simulation was calculated in each iteration. The number of particles was increased exponentially for each measurement, ranging from 125 to 216.000. The gray parabola was plotted to illustrate the quadratic running time.



(a) Initialization time



(b) Running time

Figure 10.2.: The particles were placed in a cuboid grid structure to represent a homogeneous distribution. The number of particles and cells was increased exponentially for each measurement, ranging from 1.000 to 1.000.000, while keeping the particle density constant. The measured data includes initialization times (a) and running times (b) for both implementations and both platforms. $tNestedOMP$ and $tNestedCUDA$ represent times achieved on OMP and CUDA with the nested-views-implementation, as described Section 7.1, while $tTwoDimOMP$ and $tTwoDimCUDA$ represent times achieved on on OMP and CUDA with the two-dimensional-views-implementation, as described in Section 7.2. The graphs represent the ratio of the times achieved with different implementations, but on the same platform.

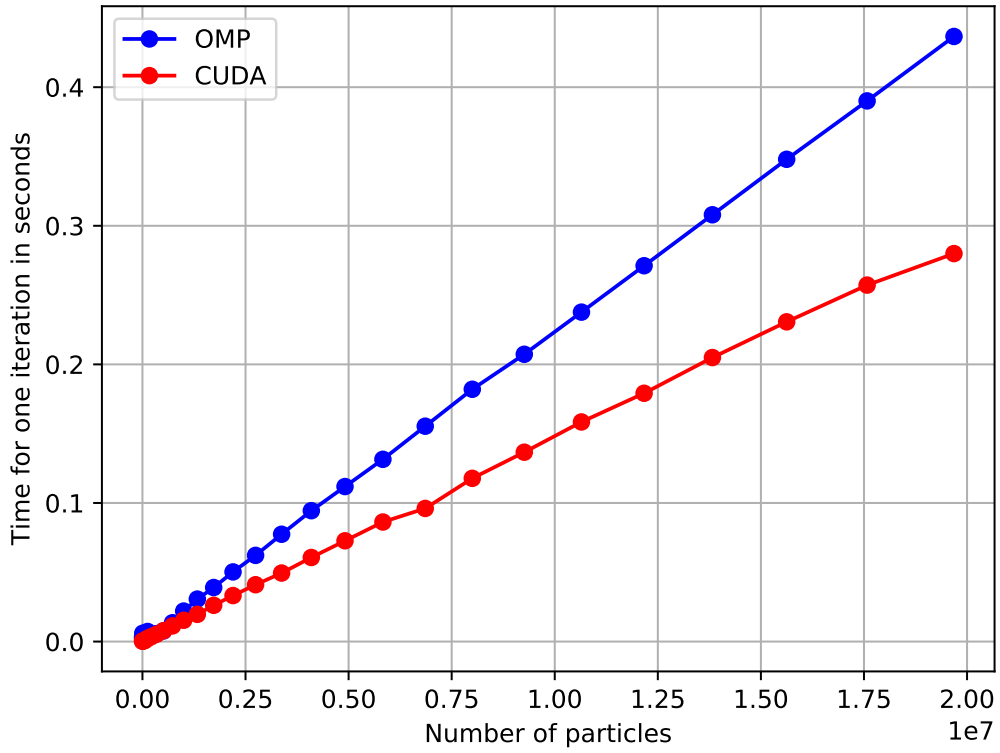


Figure 10.3.: The Particles were created in a grid pattern. For each cell in the simulation, one particle was placed in its center, so the number of particles is always equal to the number of cells. The number of cells was increased exponentially for each measurement, ranging from 1.000 to ca. 20.000.000.

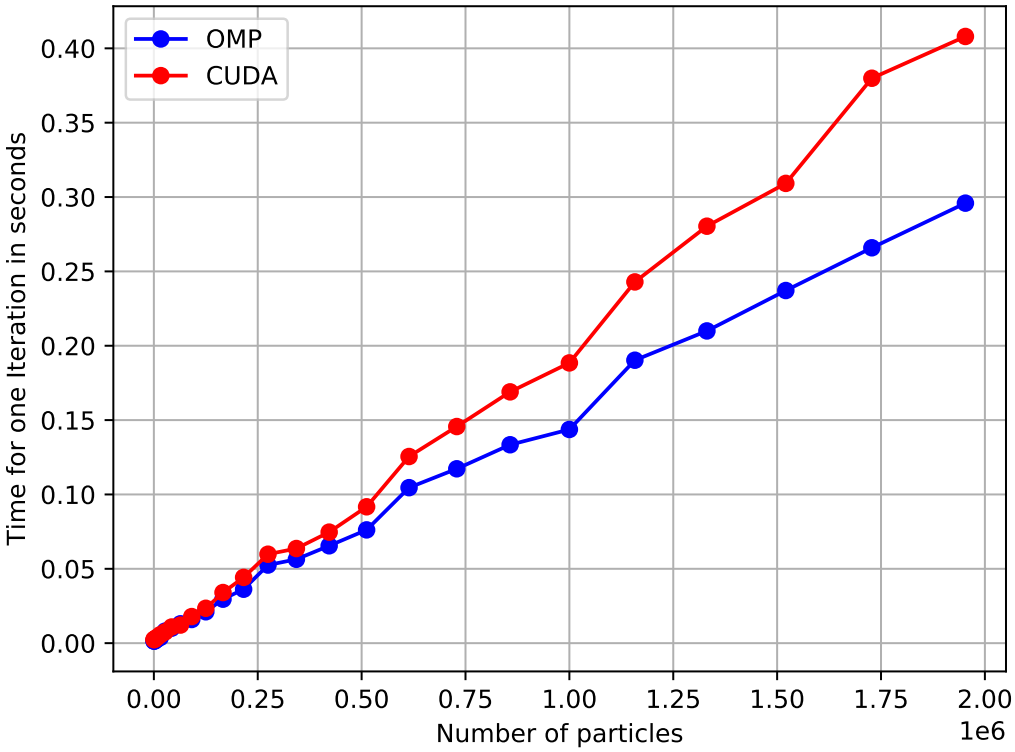
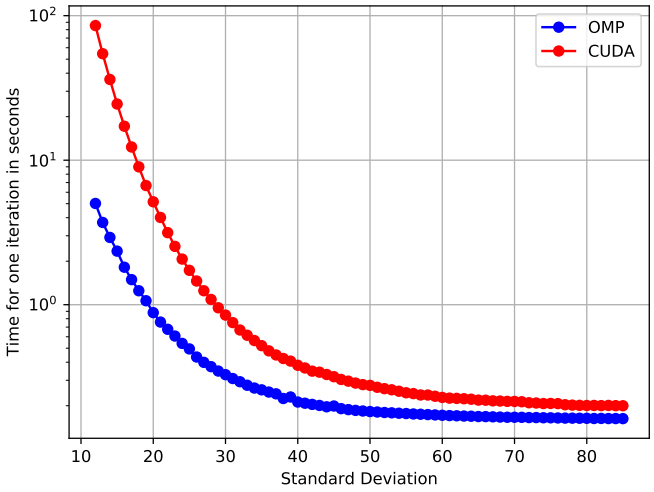
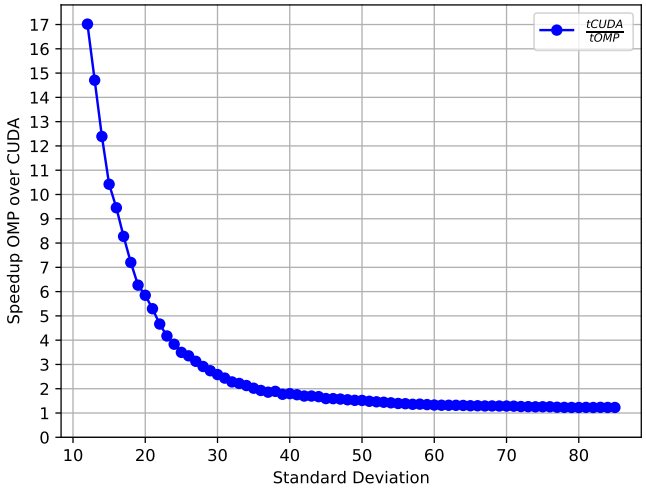


Figure 10.4.: The particles were places in a cuboid grid structure to represent a homogeneous distribution. The number of particles and cells was increased exponentially for each measurement, ranging from 125 to ca. 2.200.000, while keeping the particle density constant, at 9 particles per cell.



(a) Running time



(b) Ratio between running times

Figure 10.5.: The Particles were distributed according to a three-dimensional Gaussian distribution with a variable standard deviation. For every measurement, the amount of particles was fixed at 1.000.000, the amount of cells was fixed at 64.000, and the mean of the Gaussian distribution was fixed at (0,0,0). The cells form a cube with a side length of 120 and it's center at (0,0,0), the mean of the distribution. The standard deviation was increased by 1.0 for each measurement, ranging from 12,0 to 85,0. The running times are plotted in (a), the ratio between the running times of the two platforms is plotted in (b).

Part IV.

Conclusion and Future Work

The Kokkos programming model is a powerful tool to create portable, parallel, high-performance MD-simulations. A lot of functionality, that is included in AutoPas, could be taken over and rewritten with Kokkos. This code served as a comparison basis for parallel execution on multiple platforms, specifically for comparisons between CPUs and GPUs.

It could be shown, that the increased FLOP count of a GPU in comparison to a CPU, can lead to a much-increased performance in some simulation use cases. However, in a linked-cell implementation, the GPU performance is very sensitive to the homogeneity of a simulation distribution and gets significantly worse, if particles are not distributed evenly enough.

Possible future research work could be put into improving the cache efficiency of a nested-view implementation of the linked-cell algorithm.

Part V.

Appendix

List of Figures

Bibliography

[CPM19] Steven WD Chien, Ivy B Peng, and Stefano Markidis. Performance evaluation of advanced features in cuda unified memory. *arXiv preprint arXiv:1910.09598*, 2019.

[Dem18] Wolfgang Demtröder. *Experimentalphysik 1*, volume 8. Springer, 2018.

[GSBN21] Fabio Alexander Gratl, Steffen Seckler, Hans-Joachim Bungartz, and Philipp Neumann. N Ways to Simulate Short-Range Particle Systems: Automated Algorithm Selection with the Node-Level Library AutoPas. 2021. submitted.

[GZK07] Michael Griebel, Gerhard Zumbusch, and Stephan Knapek. *Numerical Simulation in Molecular Dynamics*, volume 5. Springer, 2007.

[Tro18] Christian Trott. Github:kokkos/kokkos wiki. Internet, 2018.

[Zwi11] Michael Zwick. *Predicting Cache Contention in Multicore Processor Systems*. Dissertation, Technische Universität München, München, 2011.