Karlsruhe Institute of Technology

Examination Project
Numerical Simulations in Molecular Dynamics I & II

Modeling Benzene using Molecular Dynamics Simulation on GPUs

Adrian Kummerländer

March 31, 2021

PD Dr. Volker Grimm

M.Sc. Tobias Kliesch

Department of Mathematics

Karlsruhe Institute of Technology

Contents

1 Basics

Due to widespread availability of powerful electronic computers numerical simulations have been established as a *third pillar of science* alongside theoretical investigation and experimentation. One important application of such simulations are problems in the field of fluid dynamics or more generally of transport problems. Approaches for simulating such systems may be coarsely grouped into macroscopic, mesoscopic and microscopic models. Where macroscopic and mesoscopic approaches model fluid-like behaviours on a higher level, either fully abstracting or at least statistically approximating the underlying physical reality of particles moving in space, microscopic approaches aim to directly model the dynamics of the very particles of which e.g. a fluid as described by Navier Stokes or transport as described by an advection diffusion equation is but an emergent property. As such microscopic models can also capture phenomena in areas where both other approaches break down i.e. on the level of individual molecules or atoms. Notably this viewpoint of particle interactions is not restricted to very small scales but can also be used to model other systems up to whole galaxies.

The specific microscopic model employed by both this examination project and the underlying lecture on *Numerical Simulations in Molecular Dynamics* [1] considers a set of particles with distinct spatial positions and evaluates short- and long range interactions between them.

1.1 Relevance of Benzene

Bezene C_6H_6 is an organic molecule consisting of a ring of six carbon atoms where the carbon atoms are additionally connected to one hydrogen atom each. Configurations of two Benzene molecules $(C_6H_6)_2$ are called dimers. Benzene dimers are the simplest prototype of π-π stacking which is a molecular interaction that is an important aspect of e.g. nucleobase stacking, protein interactions or drug binding in DNA and RNA [2]. As such investigations of benzene interactions provide a foundation for research into these essential areas.

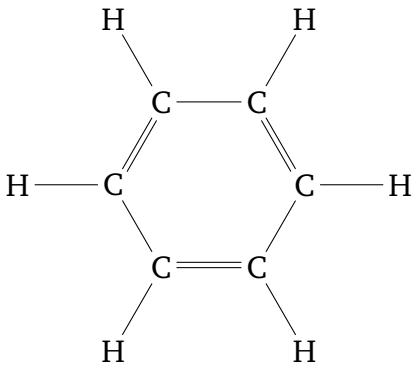


Figure 1: Chemical structure of benzene

1.2 Potentials for Molecular Dynamics

Molecular dynamics simulation as discussed in the lecture are based on the laws of classical mechanics, avoiding e.g. the complexity of the actual quantum mechanical phenomena governing molecular behavior. That is they consider atoms as point masses influenced by some set of forces under Newton's second law. A molecular dynamics simulation thus consists of two essential steps: Computing the forces between particles and updating their positions by numerically integrating the Hamiltonian system.

For the integration step one can in principle apply any numerical integration method such as e.g. Euler's method. However in order to avoid the numerical instability of the standard explicit Euler method and to ensure energy convervation it is better to use a symplectic method such as Velocity-Störmer-Verlet.

Definition 1.1 (Velocity-Störmer-Verlet)**.** *Let* $x_i^n, v_i^n, F_i^n \in \mathbb{R}^d$ *be the spatial position, velocity and total force on a particle* i *with mass* $m_i \in \mathbb{R}_0^+$ *at time* $n \in \mathbb{Z}$*. The successive computation of expressions*

$$x_i^{n+1} = x_i^n + \tau v_i^n + \frac{\tau^2}{2m_i} F_i^n$$
$$v_i^{n+1} = v_i^n + \frac{\tau}{2m_i}\left(F_i^n + F_i^{n+1}\right)$$

for position and velocity is called Velocity-Störmer-Verlet integration with step size $\tau \in \mathbb{R}^+$*.*

Individual forces F_{ij} between particles adding up to the total per-particle force using

$$F_i = \sum_{j=1, j\neq i}^{N} F_{ij}$$

are commonly formulated using potentials from which the forces are derived as the negative gradient. There exists a large variety of different potentials to model different physics — a common case are pairwise potentials depending only on the distance between two particles.

Definition 1.2 (Lennard-Jones potential)**.** *Let* $x_i, x_j \in \mathbb{R}^d$ *be the spatial locations of two particles and* $r_{ij} := \|x_j - x_i\|$ *the distance between them.*

$$U(r_{ij}) := 4\epsilon\left(\left(\frac{\sigma}{r_{ij}}\right)^n - \left(\frac{\sigma}{r_{ij}}\right)^m\right)$$

is called the Lennard-Jones potential for parameters $m < n, \epsilon, \sigma \in \mathbb{R}^+$*.*

Instances of the Lennard-Jones potential are amongst the most-used models for interactions in MD simulations. A common parameter choice is $n = 12$ and $m = 6$ resulting in the 12-6 potential where the remaining parameters ϵ and σ can be fitted to model a variety of physical phenomena. In general σ describes the distance at which attraction and repulsion forces are balanced and ϵ describes the strength of the interaction.

Definition 1.3 (Coulomb potential)**.** *Let* $x_i, x_j \in \mathbb{R}^d$ *be the spatial locations of two particles and* $r_{ij} := \|x_j - x_i\|$ *the distance between them.*

$$C(r_{ij}) := \frac{1}{4\pi\epsilon_0}\frac{q_i q_j}{r_{ij}}$$

is the Coulomb or electrostatic potential which models the interaction between two point charges $q_i, q_j \in \mathbb{R}$ *for vacuum permittivity* ϵ_0*.*

The force on x_i due to the Coulomb potential between x_i and x_j with attached point charges q_i respectively q_j can be derived as

$$\begin{aligned} F_{\text{coulomb},ij} &= -\nabla_{x_i} C(r_{ij}) \\ &= -\frac{1}{4\pi\epsilon_0}\frac{q_i q_j}{r_{ij}^3}(x_j - x_i). \end{aligned}$$

These Lennard-Jones and Coulomb potentials can be used to model both Van der Waals and electrostatic interactions between atoms. Databases such as MolMod [3] collect the necessary parameters for a wide variety of atoms and ions. Connection of such particles into fixed molecular structures where e.g. the distance or angle between atoms behaves like a harmonic oscillator can be modeled using fixed bond potentials.

Definition 1.4 (Bond potential)**.** *Let* $x_i, x_j \in \mathbb{R}^d$ *be the spatial locations of two particles and* $r_{ij} := \|x_j - x_i\|$ *the distance between them.*

$$U_b(r_{ij} := \frac{1}{2}k_b(r - r_0)^2$$

is the bond potential between x_i *and* x_j *harmonically oscillating at distance* $r_0 \in \mathbb{R}^+$ *with spring constant* $k_b \in \mathbb{R}^+$*.*

Definition 1.5 (Angle potential)**.** *Let* $x_i, x_j, x_k \in \mathbb{R}^d$ *be the cartesian spatial locations of three particles spanning an angle*

$$\Theta_{ijk} = \arccos\left(\frac{\langle x_j - x_i, x_j - x_k\rangle}{\|x_j - x_i\|\|x_j - x_k\|}\right)$$

between them. One possible potential for describing oscillation around an equilibrium angle Θ_0 *with strength* $k_0 \in \mathbb{R}^+$ *is*

$$U_a(\Theta) := -k_0(\cos(\Theta - \Theta_0) - 1).$$

Definition 1.6 (Torsion potential)**.** *Let* $x_i, x_j, x_k, x_l \in \mathbb{R}^3$ *with* $\mathbf{r}_{ab} = x_b - x_a$ *be the cartesian spatial locations of four particles spanning between them two planes intersecting at the dihedral angle*

$$\phi_{ijkl} = \pi + sign(\langle \mathbf{r}_{ij}, \mathbf{r}_{jk} \times \mathbf{r}_{kl}\rangle)\arccos\left(\frac{\langle \mathbf{r}_{ij} \times \mathbf{r}_{jk}, \mathbf{r}_{jk} \times \mathbf{r}_{kl}\rangle}{\|\mathbf{r}_{ij} \times \mathbf{r}_{jk}\|\|\mathbf{r}_{jk} \times \mathbf{r}_{kl}\|}\right)$$

using the polymer convention.

$$U_t(\phi) := \sum_i k_i \cos^i \phi$$

is one possibilty for a torsion potential based on this dihedral angle.

1.3 GPU Programming

The highly parallel and bandwidth-optimized nature of graphics processing units (GPUs) renders them a convenient target platform for many scientific applications. The main questions for determining whether a given computation can benefit from utilizing a GPU are the degree of parallelism and data interdependencies as well as the amount of instruction stream branching.

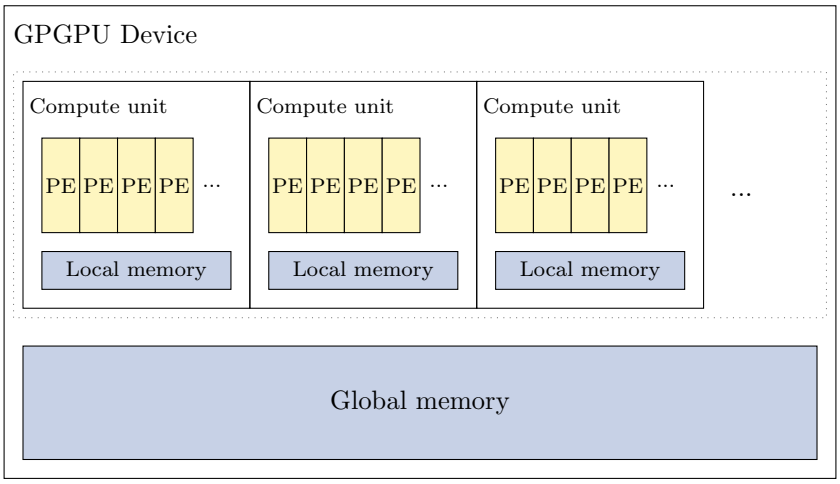


Figure 2: Basic hardware structure of a generic GPU

Also of importance is the required floating point precision — while modern GPUs support both single and double precision computations, performance parity w.r.t. to the available memory bandwidth is commonly only available on high end GPUs specifically intended for HPC applications.

General purpose programs for GPUs may be developed in a variety of frameworks and languages depending on the specific hardware. Common choices are the proprietary options CUDA and ROC of Nvidia respectively AMD, GLSL compute shaders or the heterogenous OpenCL framework which can target GPUs of both manufacturers as well as CPUs and even FPGAs [4] using a dialect of the C programming language with convenient extensions for e.g. vector arithmetic. The simulation code developed for this examination utilizes OpenCL embedded in a Python runtime [5] for quick and expressive prototyping. This is a common approach for scientific computing, e.g.Python is a major language in machine learning but the actual computationally demanding parts are mostly delegated to efficient C/C++ codes.

Considering the specific requirements of MD simulations: The computation of pairwise per-particle forces as discussed in the lecture is a *embarrasingly* parallel problem if one is willing to compute each pair of forces twice — once for each particle. Then each computation of the force on a single particle may read the positions of any other particle in the system but writes only its own force to memory. In an ideal situation a system of N processing units may compute the forces on N particles in a constant time unit (ignoring for now the N^2 term due to accessing potentially all other particles).

More problematic is the pointer-based structure that is used for encoding the structure of individual molecules. Or more specifically not the pointer structure by itself but rather the way it implicitly encodes the different bond types which requires branching. It should be noted here that branching is not impossible on GPUs but as each thread block executes its instruction stream in lockstep any branching causes *thread divergence*, potentially slowing down performance significantly. For this reason section 2.2 will describe an alternative approach for managing intramolecular interactions.

The lecture introduced the linked cell approach for reducing the $\mathcal{O}(N^2)$ complexity of computing the pairwise potentials to $\mathcal{O}(N)$. While this method could in principle also be employed on GPUs, a Verlet list based approach was deemed more convenient for the code developed during the course of this examination project. This approach will be detailed in section 2.1.

2 Molecular Dynamics Simulation Code

Following the GPU considerations of the previous section the MD code [6] developed by the author utilizes Python as a high-level interface language for declaring parameters, managing the data structures and post-processing for visualization (including e.g. the automatic generation of velocity histograms and temperature plots) while the actual simulation is performed on the GPU using OpenCL.

```
if self.step % self.neighborhood_step == 0:
    self.update_neighborhoods()
self.evolve_x()
self.compute_intramolecular_forces()
self.compute_intermolecular_forces()
self.evolve_v()
```

Listing 1: OpenCL kernel calls for a single simulation step in Python

All data is stored as single precision floating point values in order to utilize the full potential of the available hardware as double precision performance is often comparably limited on desktop-grade GPUs. While the code also supports double precision this was not used for the simulations as no precision issues were observed w.r.t. dimer formation.

Ignoring potential-specific data the basic data structure of the code consists of individual arrays storing the particle positions, velocities and forces. Two force arrays `force_curr` and `force_prev` are maintained to support the Velocity-Störmer-Verlet integration kernel functions `update_x` and `update_v` between which the forces are accumulated into `force_curr`. This fully seperates the generic integration part of the code from any potential-specific considerations.

To clearly separate host-side data structures maintained as configuration by the user and the actual device-side data structures used for the simulation, the code offers a `Simulation` class that is constructed from a `MoleculeCollection` consisting of the initial simulation setup together with all required parameters. The `Simulation` class also offers basic utility functions for computing and controlling the temperature as well as collecting various statistics for tracking the simulation process.

2.1 Verlet List Algorithm

The basic idea of the Verlet list [7] is to perform a full $\mathcal{O}(N^2)$ traversal of the particle neighborhood only every n steps, storing the indices of all neighboring particles within the cutoff distance $r_{cut} \in \mathbb{R}^+$ extended by some suitable *skin* distance $\delta_{skin} \in \mathbb{R}^+$ to be reused for the other $n-1$ steps. During these $n-1$ steps this *bookkeeping device* significantly reduces the computational effort as only the particles within $r_{cut} + \delta_{skin}$ need to be examined — note that the actual cutoff w.r.t. to the potential calculation is still r_{cut}, the skin layer is only used to track particles that may move into the cutoff sphere within $n-1$ steps.

As long as the skin distance satisfies $\delta_{skin} \geqslant 2n\tilde{v}\tau$ for step size τ and velocity bound $\tilde{v}$ this is not expected to increase the error beyond what is already given by the cutoff. The original publication by Verlet [7] suggests to use to root mean square speed as $\tilde{v}$ but other choices such as the maximum velocity at update time are also possible. It is important to take into account velocity changes caused by temperature control to prevent e.g. underestimation of the skin layer depth during heating phases. Tuning of the related parameters n and τ_{skin} can be used for controlling how large the per-particle neighbor lists are and thus to some degree the resulting performance. See figure 3 for a benchmark illustrating this relationship.

The average number of neighbors to be stored per-particle may be estimated using

$$\frac{1}{N}\sum_{i=1}^{N} |\text{neighbors}(x_i)| \sim \frac{4}{3}\pi(r_{cut} + \delta_{skin})^3\rho_N - 1$$

for given particle density ρ_N. Comparing this to the average number of neighbors in a linked cell method

$$\frac{1}{N}\sum_{i=1}^{N} |\text{neighbors}(x_i)| \sim 27 r_{cut}^3\rho_N - 1$$

the Verlet list approach only needs to investigate $\sim$ 16% of the neighbor candidates required by linked cell during the $n-1$ optimized timesteps for sufficiently small δ_{skin}. This directly suggests that combining both methods by e.g. using a linked cell like approach for the Verlet list construction may be a convenient next step for improving the performance, see e.g. [8] for work along those lines.

Additionally performance may be improved by using more sophisticated criteria for determining when the neighbor lists need to be invalidated. However, the algorithm's basic version proved sufficiently fast for just-in-time simulation and visualization of the rather small $N \in \mathcal{O}(10^3)$ particle systems explored by this work. The code comfortably reached around $1\,\mathrm{ps}$ simulated time per real world second for a system of 1000 argon atoms in a periodic $512\,\mathrm{nm}^3$ box at 300 K with $\tau = 0.0005\ \mathrm{ps}$ and $r_{cut} = 0.3395 \cdot 2.5\ \mathrm{nm}$ in early benchmarks using a desktop-grade Nvidia GeForce RTX 2070 GPU. Despite that, reductions of the neighborhood search time complexity will still quickly become relevant for larger problems as time spent on list updating increasingly dominates the non-updating time steps.

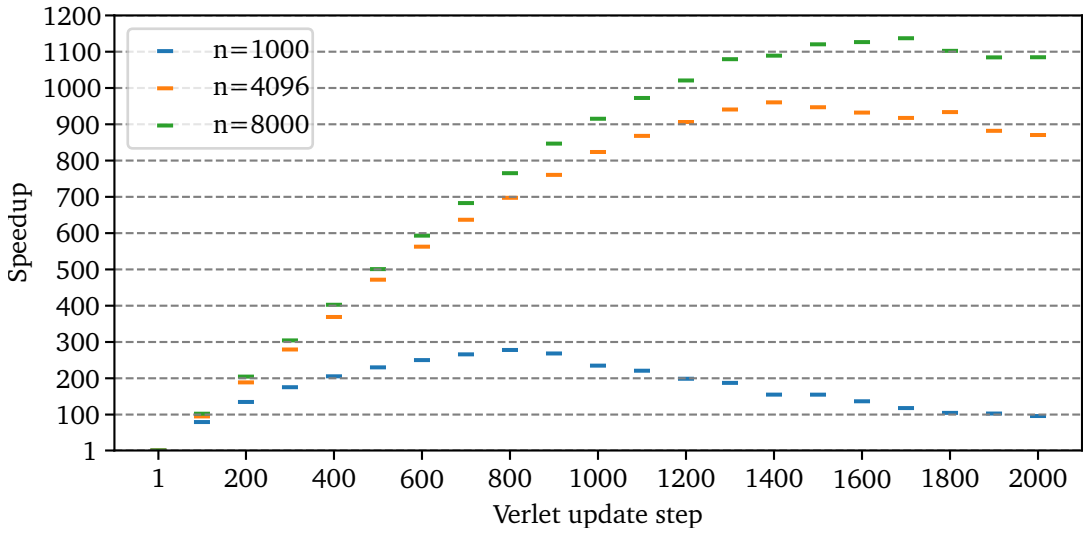


Figure 3: Speedup compared to $r_{skin} = 0$ for different particle counts

Figure 3 shows speedup values relative to the performance when resolving the neighborhood at each step for three differently sized simulations of argon at 300 K and with a average density of $\rho_N = 8$ atoms per cubic nanometer. The smallest simulation at $n = 1000$ atoms presents a special case as this number doesn't saturate the 2304 processing elements of the test GPU which combined with a small memory footprint results in both the lowest speedup and highest number of *mega particle updates per second* at 3.22 compared to 0.73 for $n = 8000$ and 1.77 for $n = 4096$ particles.

Investigation of a heterogenous approach where the list construction is performed on the CPU for which e.g. sorting approaches are more straight forward to implement while the bulk updates and force calculations stay on the GPU might also warrant a closer look. Distribution of processing to multiple GPUs could be realized by merging the Verlet list for GPU-local processing and linked cell approach for communication.

2.1.1 Periodic Boundaries

Integration of periodic boundaries in a linked cell approach is straight forward as one only needs to change which cells are considered as neighbors to construct e.g. a periodic torus geometry. In the case of the Verlet list algorithms periodic boundaries can be resolved during list construction by considering not just the actual cell locations as neighbor candidates but also the locations shifted along any periodic offset and adding the shifted candidate to the neighborhood list if it falls within the considered distance. This should be done in the inner loop of list construction in order to not increase the memory bandwidth demands even further.

Listing 2 shows an excerpt of how this implemented. One thing to highlight there is the usage of masks built from evaluations of comparison expressions instead of branching constructs. This is a common pattern for vectorizable code on both CPUs

and GPUs. Due to this pattern the code not only computes forces without any branches but can do the same also for the list construction — the possible branching due to the loops can be ignored here as the bounds are the same for all threads.

```
for (unsigned int jParticle=0; jParticle < $n_atoms; ++jParticle) {
  for (int sX=-1; sX <= 1; ++sX) {
    for (int sY=-1; sY <= 1; ++sY) {
      for (int sZ=-1; sZ <= 1; ++sZ) {
        vec_t shift = $domain_size * (vec_t)(sX,sY,sZ);
        vec_t jPos = get(pos, jParticle) + shift;

        scalar_t r = length(jPos - iPos);

        unsigned int iMolecule = molecules[iParticle];
        unsigned int jMolecule = molecules[jParticle];

        scalar_t iMass = pos[iParticle].w;
        scalar_t jMass = pos[jParticle].w;

        scalar_t mask = (r < cutoff + skin)
                      * (iParticle != jParticle)
                      * (iMolecule != jMolecule);

        // [...] Lennard-Jones parametrization

        coulomb_i_indices[idx]   += mask * jParticle;
        coulomb_i_charge[idx]    += mask * coulomb_q(iMass) * coulomb_q(jMass);
        coulomb_i_shift[idx].xyz += mask * shift;

        idx += mask;
      }
    }
  }
}
```

Listing 2: Excerpt of Verlet list construction for a periodic cube in OpenCL

Prior to reconstruction of the neighborhood list any molecules that are outside the prime domain are wrapped back. It should also be noted that the mask excludes any pair potentials between atoms of the same molecules and not just between every four consecutive atoms. This was done for convenience as no impact on the benzene model was observed — intramolecular pair potentials may be easily unmasked if needed.

2.1.2 Force Computation

Using the explicit neighbor lists maintained by the Verlet list algorithm enables branch free computation of any pairwise forces on the GPU. Listing 3 illustrates how this is implemented for Coulomb forces. Kernel functions such as these are applied using one thread for each particle which is enabled by the massive parallelism provided by GPUs.

```
__kernel void compute_coulomb(__global data_vec_t* pos,
                              __global data_vec_t* force,
                              __global unsigned int* coulomb_count,
                              __global unsigned int* coulomb_indices,
                              __global scalar_t*     coulomb_charge,
                              __global data_vec_t*   coulomb_shift,
                              scalar_t cutoff)
{
  unsigned int iParticle = get_global_id(0);
  unsigned int nNeighbors = coulomb_count[iParticle];

  vec_t f = 0;

  for (unsigned int iNeighbor = iParticle*$max_coulomb;
                    iNeighbor < iParticle*$max_coulomb + nNeighbors;
                  ++iNeighbor) {
    f += coulomb(pos, iParticle,
                 coulomb_indices[iNeighbor],
                 coulomb_charge[iNeighbor],
                 coulomb_shift[iNeighbor].xyz,
                 cutoff);
  }

  force[iParticle].xyz += f;
}
```

Listing 3: Example for the computation of pairwise forces in OpenCL

One significant difference compared to how such computations are carried out in the lecture code are the storage of particle and parameter information in a *Structure-of-Arrays* layout instead of in a continguous `Particle` structure in a *Array-of-Structures* which is essential for vectorization of which GPUs can be considered a specical case. Notably the computation of each pair force is also carried out twice instead of only once to avoid write conflicts that would occur if more than one thread needs to update the accumulated force of a single particle at the same time.

2.2 Molecular Structures

The lecture introduced the usage of a 2D array-of-pointers `Particle *M[][]` to encode the neighborhood relations within a linear molecule s.t. `M[i][j]` points to the j-th atom of the i-th molecule. The actual bonded interaction potentials are then applied depending on list length and particle positions within the list.

While possible in principle this approach has various downsides when targeting a GPU: It performs molecule and particle specific branching and needs to be parallelized on a per-molecule basis possibly providing worse device saturation as there are fewer molecules than particles. As this approach would also need to be adapted in order to facilitate the cyclic and branching neighborhood of benzene — e.g. using a adjacency graph-like approach which is even more unsuited to GPU parallelization — a different molecule datastructure was chosen for the code in this work.

Similar to how a per-particle list of neighbor indices is already being maintained for non-bonded potentials, the code also maintains lists for each type of bonded potential. This also includes separate lists of any bond-specific parameters in a vectorization-friendly data layout and enables branch free (ignoring the loop bounds that do not lead to *true* instruction stream divergence) per-particle parallelization.

Listing 4 illustrates how the angle forces are computed in this approach. One thing to highlight there is the branch-free nature of the `angle` function that computes all three forces of a given angle potential and returns only one of them by masking. Adding to this that each angle potential configuration is added to the angle potential list for each involved particle means that each particle force is computed nine times in total. This might seem very wasteful at first glance but is a deliberate tradeoff to avoid synchronization and also considering that on modern hardware it is frequently more efficient to recompute values more often than strictly needed instead of increasing branch divergence and memory bandwidth.

This optimized data structure is transparently generated by the code given a user-friendly description of some molecule as a Python data structure. An example for this is provided by Listing 6 in the upcoming modeling section. Any required maximum number of bonds, buffer sizes and indices are maintained in the background without requiring user intervention.

It should be noted that the approach as documented here potentially leads to lots of unnecessary threads when a simulation consists of a small number of molecules and a large number of single atoms. If this should become a performance issue it is easily fixed by maintaining a list of molecule-bound atoms and calling the kernel functions on this list instead of all particles.

Individual per-potential kernels can optionally be combined into a single kernel that processes either all or the used subset of potentials in order to balance GPU utilization and reduce the number of expensive kernel calls. This approach also isolates the implementations of different bonded potentials, allowing straight forward introduction of new or adapted versions of intramolecular interactions without needing to consider existing potentials.

```
vec_t angle(__global data_vec_t* pos, unsigned int iParticle,
            unsigned int i, unsigned int j, unsigned int k,
            scalar_t theta0, scalar_t k0) {
  vec_t iPos = get(pos, i);
  vec_t jPos = get(pos, j);
  vec_t kPos = get(pos, k);
  // [...]
  vec_t fi = factor * (sd2 * sq(length(rkj)) * rij - rkj);
  vec_t fk = factor * (sd2 * sq(length(rij)) * rkj - rij);
  return (iParticle == i) *   fi
       + (iParticle == k) *      fk
       + (iParticle == j) * -(fi+fk);
}

__kernel void compute_angles(__global data_vec_t* pos,
                             __global data_vec_t* force,

                             __global unsigned int* angle_count,
                             __global unsigned int* angle_indices,
                             __global scalar_t* angle_theta0,
                             __global scalar_t* angle_k)
{
  unsigned int iParticle = get_global_id(0);

  vec_t f = 0;

  unsigned int nAngles = angle_count[iParticle];
  for (unsigned int iAngle = iParticle*$max_angles;
                    iAngle < iParticle*$max_angles + nAngles;
                  ++iAngle) {
    f += angle(pos, iParticle,
               angle_indices[0*$n_angles+iAngle],
               angle_indices[1*$n_angles+iAngle],
               angle_indices[2*$n_angles+iAngle],
               angle_theta0[iAngle],
               angle_k[iAngle]);
  }

  force[iParticle].xyz += f;
}
```

Listing 4: Excerpt of OpenCL kernel for computing the angle potentials for all particles

2.3 Just-in-time Visualization

It can both be quite interesting to watch visual representations of simulations and also useful for quickly ensuring that the simulation evolves as expected as well as to gain a first insight into the resulting dynamics. As graphical visualizations are the original purpose of graphics processing units it is straight forward to visualize data of simulations that are already being performed on the GPU. In the present case the memory buffer containing the particle positions can simply be passed to an OpenGL shader that draws spheres for every location and lines representing the bonds between atoms.

It should be noted that different from the simulation code, the sphere-drawing part and some scaffolding for manipulating the view was not developed from scratch but repurposed from a previous code developed by the author for investigating hard sphere gas models in kinetic theory [9]. This includes optimized sphere drawing using appropriately shaded 2D squares instead of polygon meshes.

In addition to pure visualization of particle positions the code also provides velocity histograms and temperature plots based on matplotlib [10] that are continuously updated and displayed alongside the 3D view.

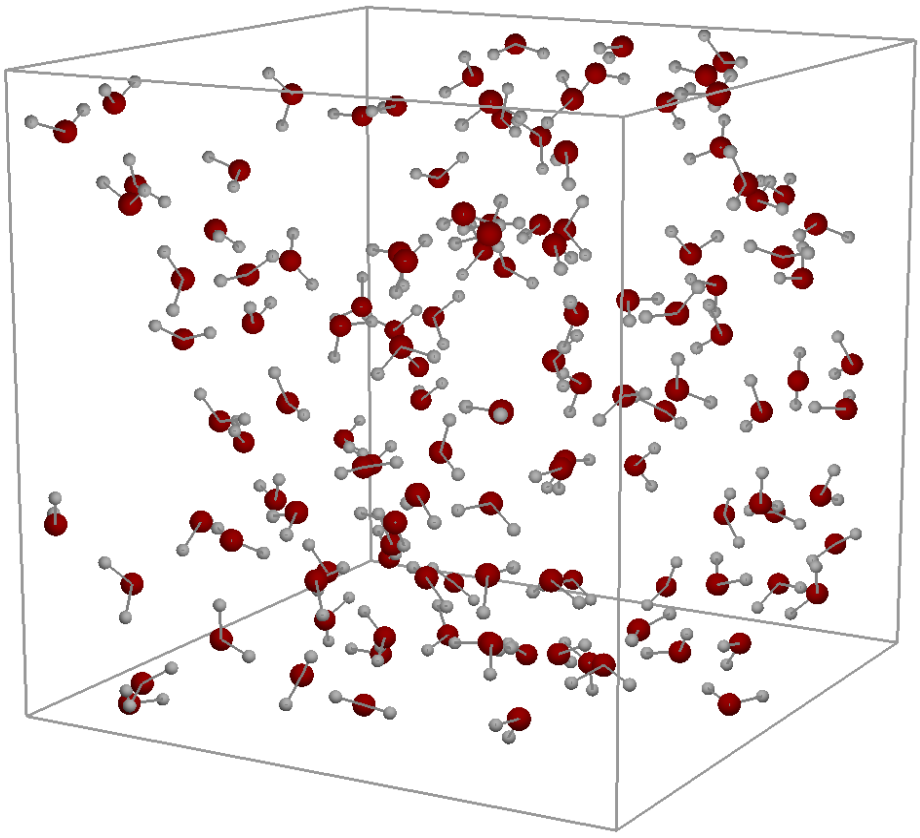


Figure 4: Just-in-time visualization of a MD simulation

3 Modeling Benzene

Using the toolbox described by the previous sections the benzene molecule can be modeled to the degree that the characteristic dimer configurations emerge. Bond potentials are used to maintain the given distances between atoms, angle potentials are used to maintain the 120° angles of the rings and torsion potentials ensure that the molecule stays planar. Both Lennard-Jones and Coulomb potentials are used together to model the intermolecular forces, utilizing Lorentz-Berthelot combination rules for C−H Lennard-Jones interactions.

3.1 Dimensionalization

In order to get physically relevant results from a simulation it is essential to use a consistent set of unit for all computations. This is not just important for being able to correctly convert any collected results to physical quantities but also to ensure that all quantities in the system relate correctly to each other.

A convenient base unit choice for the scales at which simulations on the molecular level are commonly performed are nanometers for lengths, picoseconds for time, the atomic mass unit for mass and kilojoules per mole for energies. This tuple of length, time, mass and energy units can then be used to derive any other units and convert any physical constants that may be required within the context of this simulation. For example Boltzmann's constant can be converted to

$$k_B = 8.31446262 \times 10^{-3}\,\mathrm{kJ\,mol^{-1}\,K^{-1}}$$

in order to compute the temperature of a system of N particles in Kelvin

$$T = \frac{2}{3Nk_B}\sum_{i=0}^{N}\frac{m_i}{2}\|v_i\|^2$$

directly from velocities v_i in nanometers per picosecond and masses in atomic mass units.

Table 1 provides an overview of relevant parameters for the benzene simulation scaled the way in which they are used by the code. One thing to highlight there is that Coulomb's constant in the eponymous potential needed to be scaled in order to correctly relate Lennard-Jones and Coulomb forces in the simulation. Using $(4\pi\epsilon_0)^{-1}$ with $\epsilon_0 = 1$ yields a simulation where Coulomb forces were negligible compared to the other interaction forces — specifically this prevented reproduction of any but the sandwich state of the benzene dimer. Note that this factor is also used by the MD framework GROMACS [11] (which uses the same base unit selection) to relate mechanical and electrical quantities.

In order to test the correct relation of the units and the basic functionality of the software a simulation of an argon gas for temperatures between 0 and 300 K was performed while tracking the distribution of velocity magnitudes and comparing it to the

respective Maxwell-Boltzmann distribution. A video including both the temperature plot and a velocity histogram is available at https://youtu.be/FK2TxbCkx8M.

Description		Value	Unit
Time discretization	τ	0.0005	ps
C mass	m	12.011	u
H mass	m	1.008	u
C−C bond potential	k_b	33700	$kJ\,mol^{-1}\,nm^{-2}$
	r_0	0.14	nm
C−H bond potential	k_b	34000	$kJ\,mol^{-1}\,nm^{-2}$
	r_0	0.108	nm
C−C−C angle potential	k_Θ	63	$kJ\,mol^{-1}$
	Θ_0	2.0944	rad
C−C−H angle potential	k_Θ	65	$kJ\,mol^{-1}$
	Θ_0	2.0944	rad
Torsion potential	k_ϕ	50	$kJ\,mol^{-1}$
	ϕ_0	0	rad
C Lennard-Jones potential	σ	0.355	nm
	ϵ	0.07	$kJ\,mol^{-1}$
H Lennard-Jones potential	σ	0.242	nm
	ϵ	0.03	$kJ\,mol^{-1}$
C Coulomb charge	q	-0.115	e
H Coulomb charge	q	0.115	e
Electric conversion factor*	f	138.935458	$kJ\,mol^{-1}\,nm\,e^{-2}$

* Coulomb's constant $(4\pi\epsilon_0)^{-1}$ is f when scaled to the unit selection

Table 1: Simulation parameters in consistent units

3.2 Dimer

Benzene molecules tend to arrange themselves into certain pairwise low-energy states that are stable at low temperatures with a disassociation limit at 60 K [12]. Measurement and reproduction of detailed information on the structure of these dimer shapes is a major focus of benzene simulations in literature [2][12][13][14].

Listing 5 illustrates how simulations can be set up in the author's MD code [6]. The spatial and intramolecular bond configurations of various molecules including benzene are maintained as tuples in a `library` model from where they may be easily instantiated into the simulation domain at any desired position, optionally rotated by an arbitrary chain of rotation matrices. Listing 6 contains the configuration of benzene.

Various NTV ensemble simulations of two molecules positioned at or reasonably close to the dimer configurations as given by e.g. [13] were performed and were able to reliably reproduce various characteristics such as the center-center distances of both the T-shape and sandwich dimer at 5.19 Å resp. 3.77 Å. The parallel displaced dimer as described in literature was only found to be stable at an increased displacement of about 5 Å. It should be noted that the initialization close to or at the dimer configuration was done to simplify analysis of the different shapes. The various configurations also occur *naturally* in larger simulations but are less stable and less reproducible there due to the interactions between many molecules.

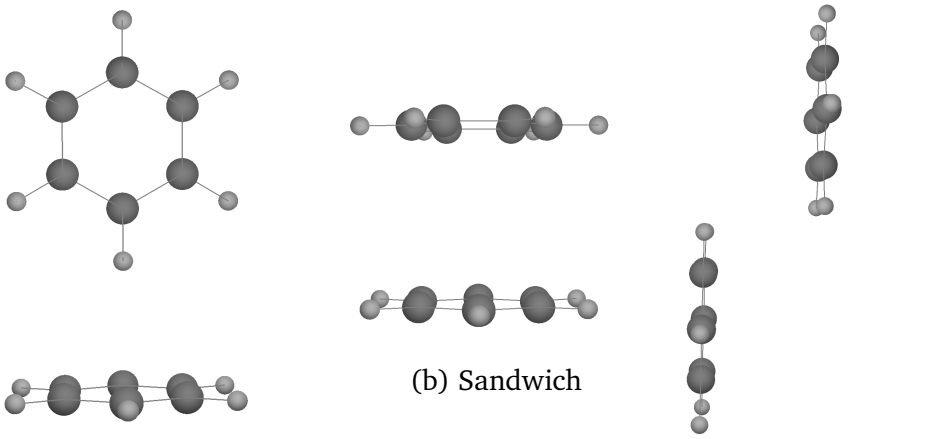
As a further check the simulations were also tested without applying the Coulomb potential which rendered both the parallel displaced and T-shaped dimers unstable but significantly increased the stability and occurence frequency of a sandwich-like shape. This supports that the combination of Lennard-Jones / Van der Waals forces and opposing charges of the involved carbon and hydrogen atoms together explain the naturally occurring dimer geometries. An overview of these simulations as a video is available at https://youtu.be/8pNgH4Rt9eo.

```
from interacticle import MoleculeCollection, LennardJones, Coulomb, Simulation

import interacticle.visualizer

from interacticle.visual import WireBox, MolecularLinks
from library import Benzene

setup = MoleculeCollection()

setup.potential(LennardJones(12.011, 0.355, 0.07))
setup.potential(LennardJones( 1.008, 0.242, 0.03))
setup.potential(Coulomb(12.011, -0.115))
setup.potential(Coulomb( 1.008,  0.115))

# T-shaped dimer
setup.add(Benzene, (1,1,1.519),
          rotations=[([0,1,0], 15), ([1,0,0], -9), ([1,0,0], 90), ([0,1,0], 90)])
setup.add(Benzene, (1,1,1.000),
          rotations=[([0,1,0], 15), ([1,0,0], -9), ([0,0,1], 30)])

setup.domain_size = 2
setup.tau = 0.0005
setup.cutoff = 0.242*2.5

simulation = Simulation(setup, opengl = True)
interacticle.visualizer.simulate(
    setup, simulation,
    [ WireBox(setup.domain_size), MolecularLinks(simulation) ],
    steps_per_frame = 100)
```

Listing 5: Simulation setup of a T-shaped benzene dimer

```
from molecules import Molecule, Atom, Bond, Angle, Torsion

Benzene = Molecule()
Benzene.atoms = [
    Atom(-0.0739,  0.1189, -0.000733, 0, 0, 0, 12.011),
    Atom( 0.0614,  0.1208,  0.035167, 0, 0, 0, 12.011),
    Atom( 0.1353,  0.0019,  0.035867, 0, 0, 0, 12.011),
    Atom( 0.0739, -0.1189,  0.000667, 0, 0, 0, 12.011),
    Atom(-0.0614, -0.1208, -0.035133, 0, 0, 0, 12.011),
    Atom(-0.1353, -0.0019, -0.035833, 0, 0, 0, 12.011),

    Atom(-0.1309,  0.2106, -0.001233, 0, 0, 0,  1.008),
    Atom( 0.1088,  0.214,   0.062267, 0, 0, 0,  1.008),
    Atom( 0.2397,  0.0034,  0.063467, 0, 0, 0,  1.008),
    Atom( 0.1309, -0.2106,  0.001267, 0, 0, 0,  1.008),
    Atom(-0.1088, -0.214,  -0.062233, 0, 0, 0,  1.008),
    Atom(-0.2397, -0.0034, -0.063533, 0, 0, 0,  1.008)
]
Benzene.connections = [
    Bond(0, 1, 33700, 0.14),
    Bond(0, 5, 33700, 0.14),
    Bond(1, 2, 33700, 0.14),
    Bond(2, 3, 33700, 0.14),
    Bond(3, 4, 33700, 0.14),
    Bond(4, 5, 33700, 0.14),

    Bond(0, 6, 34000, 0.108),
    Bond(1, 7, 34000, 0.108),
    Bond(2, 8, 34000, 0.108),
    Bond(3, 9, 34000, 0.108),
    Bond(4,10, 34000, 0.108),
    Bond(5,11, 34000, 0.108),

    Angle( 0,1,2, 120, 63),
    Angle( 1,2,3, 120, 63),
    Angle( 2,3,4, 120, 63),
    Angle( 3,4,5, 120, 63),
    Angle( 4,5,0, 120, 63),
    Angle( 5,0,1, 120, 63),

    Angle( 6,0,1, 120, 65),
    Angle( 7,1,2, 120, 65),
    Angle( 8,2,3, 120, 65),
    Angle( 9,3,4, 120, 65),
    Angle(10,4,5, 120, 65),
    Angle(11,5,0, 120, 65),

    Torsion( 0,1,2,3, 180, 50),
    Torsion( 1,2,3,4, 180, 50),
    Torsion( 2,3,4,5, 180, 50),
    Torsion( 3,4,5,0, 180, 50),
    Torsion( 4,5,0,1, 180, 50),
    Torsion( 5,0,1,2, 180, 50),

    Torsion( 6,0,1,2, 0, 50),
    Torsion( 7,1,2,3, 0, 50),
    Torsion( 8,2,3,4, 0, 50),
    Torsion( 9,3,4,5, 0, 50),
    Torsion(10,4,5,0, 0, 50),
    Torsion(11,5,0,1, 0, 50),
]
```

Listing 6: Description of the benzene molecule and its fixed bonds



(a) T-shape (b) Sandwich (c) Displaced

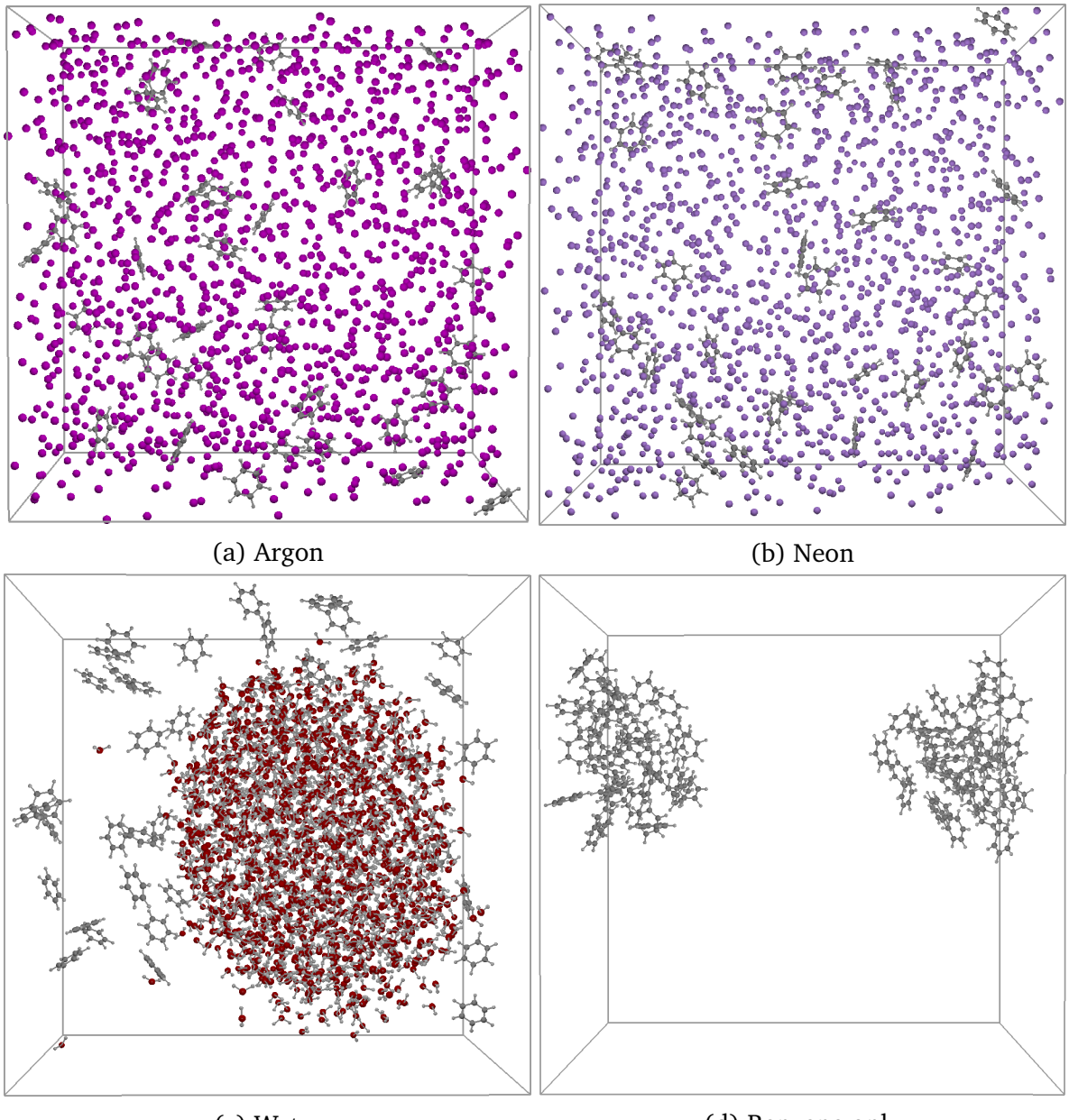
Figure 5: Configurations of the benzene dimer

3.3 Solvents

As a last test of the code's capabilities, benzene molecules are observed in solutions with other atoms and molecules, specifically the noble gases argon and neon as well as water. Lennard Jones parameters for Argon and Neon are taken from the MolMod database [3]. Water molecules are modeled using the FBA/ϵ model [15] which is a version of the common SPC/ϵ model adapted to use angle and bond potentials instead of a static setup for the molecular structure using e.g. virtual sites. As before in the case of benzene only, instantiating this new molecule into the simulation setup was very straight forward using the toolbox of potentials.

In the absence of other particles, benzene molecules were observed to form clusters consisting in part of approximate instances of the preferred configurations. This effect was less pronounced in simulations of benzene molecules together with many argon or neon atoms. Interactions with water molecules caused large, likely non-physical, deformations of the planar benzene geometry fixed by increasing the torsion potential force coefficient. Independently of this, benzene and water tended to separate in various simulations at different temperatures. No detailed analysis was performed but some snapshots of these simulations can be seen in figure 6

A next step for analysing the dynamics of these more complex systems could be to track the radial distribution of the neighborhood in order to infer e.g. preferred structures. This is commonly done to test e.g. water models as the radial distribution of e.g. oxygen-oxygen distances is both characteristic for the system and measurable in reality.



(a) Argon (b) Neon

(c) Water (d) Benzene only

Figure 6: Simulations of benzene together with other molecules

References

[1] Volker Grimm. *Numerical Simulation in Molecular Dynamics*. 2020.

[2] Hiroto Tachikawa. “Direct ab initio molecular dynamics (MD) study of the ionization on a benzene dimer”. In: *RSC Adv.* 2 (17 2012), pp. 6897–6904. DOI: 10.1039/C2RA20246H.

[3] Simon Stephan, Martin T. Horsch, Jadran Vrabec, and Hans Hasse. “MolMod – an open access database of force fields for molecular simulations of fluids”. In: *Molecular Simulation* 45.10 (2019), pp. 806–814. DOI: 10.1080/08927022.2019.1601191.

[4] J. E. Stone, D. Gohara, and G. Shi. “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems”. In: *Computing in Science & Engineering* 12.3 (2010), pp. 66–73. DOI: 10.1109/MCSE.2010.69.

[5] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 1441412697.

[6] Adrian Kummerländer. *Interacticle: Code for simulation of interacting particles*. https://code.kummerlaender.eu/interacticle/. 2021.

[7] Loup Verlet. “Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules”. In: *Phys. Rev.* 159 (1 1967), pp. 98–103. DOI: 10.1103/PhysRev.159.98.

[8] Pedro Gonnet. “Pseudo-Verlet lists: a new, compact neighbour list representation”. In: *Molecular Simulation* 39.9 (2013), pp. 721–727. DOI: 10.1080/08927022.2012.762097.

[9] Adrian Kummerländer. *BoltzGas: Simulation of velocity distribution in hard sphere gases*. https://code.kummerlaender.eu/boltzgas/. 2020.

[10] J. D. Hunter. “Matplotlib: A 2D graphics environment”. In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[11] Mark Abraham, Teemu Murtola, Roland Schulz, Szilárd Páll, Jeremy Smith, Berk Hess, and Erik Lindahl. “GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers”. In: *SoftwareX* 1 (July 2015). DOI: 10.1016/j.softx.2015.06.001.

[12] Anil Kumar Tummanapelli and Sukumaran Vasudevan. “Communication: Benzene dimer—The free energy landscape”. In: *The Journal of Chemical Physics* 139.20 (2013), p. 201102. DOI: 10.1063/1.4834855.

[13] William L. Jorgensen and Daniel L. Severance. “Aromatic-aromatic interactions: free energy profiles for the benzene dimer in water, chloroform, and liquid benzene”. In: *Journal of the American Chemical Society* 112.12 (1990), pp. 4768–4774. DOI: 10.1021/ja00168a022.

[14] C. David Sherrill, Bobby G. Sumpter, Mutasem O. Sinnokrot, Michael S. Marshall, Edward G. Hohenstein, Ross C. Walker, and Ian R. Gould. "Assessment of standard force field models against high-quality ab initio potential curves for prototypes of $\pi-\pi$, CH/π, and SH/π interactions". In: *Journal of Computational Chemistry* 30.14 (2009), pp. 2187–2193. DOI: https://doi.org/10.1002/jcc.21226.

[15] Raúl Fuentes-Azcatl and Marcia C. Barbosa. "Flexible bond and angle, FBA/ϵ model of water". In: *Journal of Molecular Liquids* 303 (2020), p. 112598. ISSN: 0167-7322. DOI: https://doi.org/10.1016/j.molliq.2020.112598.

Erklärung

Ich versichere wahrheitsgemäß, diese Ausarbeitung sowie den zugehörigen Programmcode selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde, sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, den 31. März 2021