FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT • DEPARTMENT INFORMATIK

Lehrstuhl für Informatik 10 (Systemsimulation)

Efficient Parallel Charge Summation

Patrick Mühlbauer

Masterarbeit

Efficient Parallel Charge Summation

Patrick Mühlbauer

Masterarbeit

Aufgabensteller:	Prof. Dr. Ulrich Rüde
Betreuer:	Daniel Ritter, M.Sc.(hons)
Bearbeitungszeitraum:	04.12.2013 – 04.06.2014

Erklärung:

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Systemsimulation (Informatik 10), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Masterarbeit einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 4. Juni 2014 .....

Abstract

The assignment of point charges to a regular grid is an essential step when computing long-range forces with the smooth particle mesh Ewald (SPME) method. Within this thesis, three different approaches to compute the charge assignment are presented and implemented using OpenCL. The first approach uses B-splines for the interpolation and atomic operations to avoid race conditions. The second uses a per grid point charge gathering, where no synchronisation is needed and the third approach uses a Gaussian filter to spread the charges, leading to shorter runtimes, but bad results regarding accuracy. In order to test the runtimes and the accuracy of the three approaches, a SPME implementation is developed and used to simulate a potassium chlorid (KCL) crystal. How to integrate OpenCL code into the `pydlpoly` molecular dynamics simulation package is shown at the end where the OpenCL kernels are called by C++ functions, which are called from the Fortran code.

Acknowledgements

First of all I would like to thank Daniel Ritter for being a supportive and motivating supervisor during the period of my master thesis.
I also want to thank my friends Amelie Hofer and Björn Meier for correcting.
Special thanks goes to my parents and my grandparents for supporting me.

Contents

List of Figures

Chapter 1

Introduction

In the past when the power of computers started to grow, it became possible for computer simulation to evolve. With better hardware, more and more complex processes could be modeled. Nowadays, the development of computing systems, especially parallel ones, is so advanced, that computer simulations can be found in various fields, ranging from physics, chemistry and biology to human systems in economics, psychology and engineering. In these fields, simulations can replace expensive experiments or at least complement them and thereby reduce costs or increase productivity. Many processes have to be inspected at molecular level to get a deeper understanding of the problem. The simulation of such problems is usually done using particle methods and other methods of molecular dynamics.
Molecular dynamics (MD) is a classical field in computer simulation with a simple underlying mathematical model. There is a second order ordinary differential equation (ODE) for each particle, resulting in a system of ODEs. This system has to be approximated using numerical methods. To calculate the forces acting on the particles, different numerical methods exist and were optimized over time in order to be able to handle the different force fields and potentials efficiently. The force fields and potentials are categorized regarding their interaction range, short-range and long-range. The most important method to compute short-range terms is the linked-cell method, whereas long-range potentials can be computed using the (smooth) particle mesh Ewald method, the P^3M or various tree methods. These methods are implemented in many different simulation packages, which were often implemented efficiently with Fortran and parallelized using MPI.

In the past years, a new approach for parallel computing, the General-Purpose Computing on Graphics Processing Units (GPGPU), became popular. GPUs are suited perfectly for problems which can be solved using stream processing, applying the same operations to every component of an input data. This can be applied to MD simulations, performing the same operations for every particle.

In this thesis a SPME method was implemented, using the open GPGPU computing language OpenCL. SPME is one of the methods handling long-range interactions. The basic method, together with implementation details is presented in chapter (3). The implementation was used to evaluate the efficiency and accuracy of three different approaches for the charge assignment step. The theory and the implementation, together with some step-by-step optimizations, are explained in chapter (4). Chapter (5) gives an introduction on how to integrate C/C++, together with the use of OpenCL, into `pydlpoly`, which is a molecular simulation package written in Fortran. Conclusions are provided in chapter (6).

Chapter 2

Basic concepts of Molecular Dynamics

Molecular dynamics (MD) play an important role in the field of computer simulation especially for chemical physics and the modeling of biomolecules. The movements of atoms and molecules have to be tracked over a certain period of time to get insights into their structure and dynamics. Nowadays it is possible to simulate not only small systems, but millions of these particles. That is not only possible because of today's hardware. Over the years, multiple MD methods were developed and improved to reduce the computation time without loss of accuracy.
In this chapter the basic model of these methods is developed, by discretizing Newton's Second Law using the Störmer-Verlet method followed with some background on potentials for the different types of interactions, short-range and long-range.

2.1 Discretization of Newton's Equation of Motion

When simulating a system of N particles the main task is to predict the system's state over a certain period of time. This state is defined by the particles positions $\mathbf{x}(t)$, their velocity $\mathbf{v}(t) = \dot{\mathbf{x}}(t)$ and their acceleration $\mathbf{a}(t) = \dot{\mathbf{v}}(t) = \ddot{\mathbf{x}}(t)$ at time t. With this state one can also compute global measures like the systems potentianl energy or its mean temperature. The interactions between particles are represented by empirical force fields and are described by discretized versions of Newton's second law:

$$\mathbf{F}_i(t) = m_i \cdot \mathbf{a}_i(t), \tag{2.1}$$

where $\mathbf{F}_i(t)$ is the acting force and m_i the mass of particle i at time t. Using the velocity $\mathbf{v}_i(t)$ and the representation as derivative of position $\mathbf{x}_i(t)$ leads to following system of equations:

$$\begin{aligned} &\dot{\mathbf{x}}_i(t) = \mathbf{v}_i(t), \\ &\ddot{\mathbf{x}}_i(t) = \dot{\mathbf{v}}_i(t) = \mathbf{a}_i(t) = \frac{\mathbf{F}_i(t)}{m_i}, \qquad i = 1, \ldots, N. \end{aligned} \tag{2.2}$$

For (2.2), valid, physically possible, initial conditions $\mathbf{x}_i^0 := \mathbf{x}_i(t_0), \mathbf{v}_i^0 := \mathbf{v}_i(t_0)$ and $\mathbf{F}_i^0 := \mathbf{F}_i(t_0)$ at initial time t_0 have to be chosen and can then be simulated by numerical integration. An efficient and stable approach for discretizing and integrating Newton's equations is the Störmer-Verlet method where different variants can be used. With the Velocity-Störmer-Verlet method, the position and the velocity are computed at the same time and the system

(2.2) results in the following set of discretized equations:

$$\begin{aligned} \mathbf{x}_i(t+\Delta t) &= \mathbf{x}_i(t) + \mathbf{v}_i(t)\Delta t + \frac{\mathbf{F}_i(t)}{2m}\Delta t^2, \\ \mathbf{v}_i(t+\Delta t) &= \mathbf{v}_i(t) + \frac{\mathbf{F}_i(t) + \mathbf{F}_i(t+\Delta t)}{2m}\Delta t. \end{aligned} \tag{2.3}$$

For actually computing the positions and velocities for the next time step $t + \Delta t$ one needs to know the acting forces $\mathbf{F}_i(t + \Delta t)$. The different approaches to compute $\mathbf{F}_i(t + \Delta t)$ will be explained in the next section.

2.2 Short-range and long-range potentials

This section shows how $\mathbf{F}_i(t + \Delta t)$ can be computed and how different types of potentials affect the force calculation.

The naive way of computing $\mathbf{F}_i(t + \Delta t)$ is to consider the interactions pairwise. An interaction between two particles i and j is then described as potential $\phi(r_{ij})$, where r_{ij} is the distance between particle i and j, defined as $r_{ij} = |\mathbf{x}_j(t) - \mathbf{x}_i(t)|$. The resulting force is given as

$$\mathbf{F}_{ij}(t) = \nabla\phi(r_{ij}), \tag{2.4}$$

and the force acting on particle i is

$$\mathbf{F}_i(t) = \sum_{\substack{j=1 \\ j\neq i}}^{N} \mathbf{F}_{ij} \tag{2.5}$$

It is clear that $\mathcal{O}(N^2)$ operations are needed to compute the sum in (2.5), which is a major drawback of this method.

In the second way of computing $\mathbf{F}_i(t + \Delta t)$, only interactions with particles in a certain neighborhood are taken into account and the rest is handled differently, according to the properties of the considered potential. There are two types of potentials, short-range and long-range potentials. If a potential $\phi(r_{ij})$ is decreasing faster than $\frac{1}{r_{ij}^d}$, with d as the dimension of the problem, it is classified as short-range potential, otherwise as long-range potential. Whether or not a particle j belongs to the neighborhood of particle i is defined by a cutoff radius $r_{\text{cut}} > 0$. This results in a truncated potential defined by

$$\hat{\phi}(r_{ij}) := \begin{cases} \phi(r_{ij}) & \text{for } r_{ij} \leq r_{\text{cut}}, \\ 0 & \text{otherwise.} \end{cases} \tag{2.6}$$

In case of $\phi(r_{ij})$ being a short-range potential, it can be replaced by $\hat{\phi}(r_{ij})$ without loss of accuracy. One example of such a short-range potential is the Lennard-Jones potential, which is given by

$$U(r_{ij}) = 4 \cdot \varepsilon \left(\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right) = 4 \cdot \varepsilon \left(\frac{\sigma}{r_{ij}} \right)^{6} \cdot \left(\left(\frac{\sigma}{r_{ij}} \right)^{6} - 1 \right). \tag{2.7}$$

The potential is parametrized by σ and ε, determining the zero crossing and the depth of the potential, see figure 2.1. The method to compute the short-range forces in an efficient manner is the so called linked cell method, which is relatively easy to implement. The simulation domain Ω is split into uniform cells (i.e. cubes in $\mathbb{R}^3$) with length $r > r_{\text{cut}}$. Interactions in

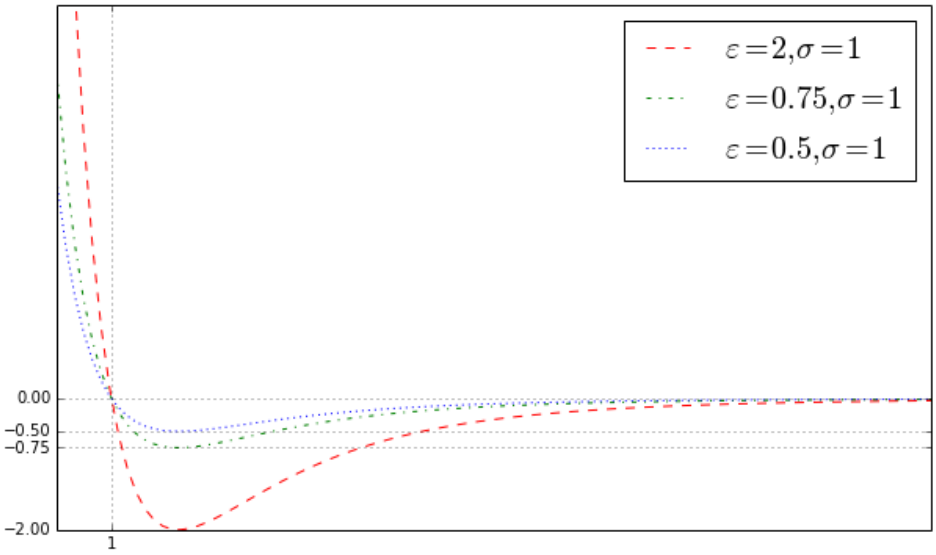

Figure 2.1: Lennard-Jones potential with different values for σ and $\varepsilon = 1$.

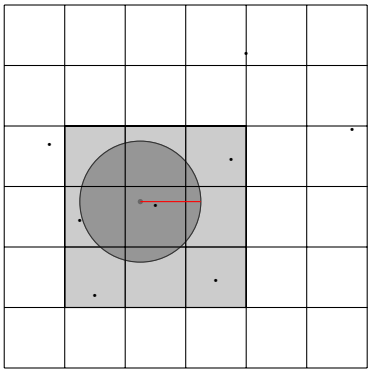

Figure 2.2: Linked Cell method in 2D with cells of size $r_{\text{cut}} \times r_{\text{cut}}$.

the truncated potential are then limited to particles within a cell and to particles in adjacent cells, see figure 2.2. This method needs the particles to be assigned to linked lists, one list for each cell. To calculate the acting force on a particle, one simply has to iterate over the lists associated with the cells in the particle's neighborhood. This leads to a computational complexity of $\mathcal{O}(N)$ if the number of particles in each cell is bounded. If the considered potential is a long-range potential, one cannot just use the currently described approach, since the use of a cutoff radius may lead to a significant loss of accuracy. Therefore the linked cell method cannot be applied directly and other methods are needed to avoid the computational complexity of $\mathcal{O}(N^2)$. The next chapter gives more insights about long-range potentials and how to handle them.

Chapter 3

Mesh-based methods for long-range potentials

It was shown how short-range potentials like (2.7) can be computed in an efficient way using the linked-cell method. For long-range potentials, like the Coulomb potential, the force between particles that are far away from each other, has to be computed. In this case other methods of approximation have to be used to avoid the computational cost of $\mathcal{O}(N^2)$. The basic idea for handling long-range potentials is to split them into a short-range part and a long-range part, i.e.

$$E = E^{\text{short}} + E^{\text{long}} \tag{3.1}$$

E^{short} can be treated with the linked cell method like before and for E^{long}, which may look like

$$E^{\text{long}} = \frac{1}{4\pi\varepsilon_0} \sum_{i=1}^{N} \sum_{j=i+1}^{N} q_i q_j \frac{1}{\|x_j - x_i\|} = \frac{1}{2} \frac{1}{4\pi\varepsilon_0} \sum_{i=1}^{N} \sum_{\substack{j=1 \\ j\neq i}}^{N} q_i q_j \frac{1}{\|x_j - x_i\|}, \tag{3.2}$$

different approximations can be used. The fundamental idea of these methods is to split the long-range part E^{long} once again into a smooth long-range part E^{lr} and a singular short-range part E^{sr}, i.e.

$$E^{\text{long}} = E^{\text{sr}} + E^{\text{lr}}. \tag{3.3}$$

The short-range part E^{sr} can again be calculated using the linked cell method and E^{lr} can be approximated with so-called grid based methods such as the smooth particle-mesh Ewald (SPME) method of Essman et al. [5], which will be explained later in this chapter.

3.1 The Poisson Equation for long-range potentials

For the computation of a long-range potential such as (3.2) different methods exist. Which type of method is used depends on the representation of the potential. It is possible to use the integral formulation

$$\Phi(\mathbf{x}) = \frac{1}{4\pi\varepsilon_0} \int_{\mathbb{R}^3} \rho(\mathbf{y}) \frac{1}{||\mathbf{y} - \mathbf{x}||} d\mathbf{y}, \tag{3.4}$$

with charge density $\rho(\mathbf{x})$ and the dielectric constant ε_0, which then has to be discretized, following by computing a matrix-vector multiplication. Various methods for this approach exist, like multilevel methods [3] or the tree code of Barnes and Hut [2]. The other representation of the potential Φ is Poisson's equation, which is given by

$$-\Delta\Phi(\mathbf{x}) = \frac{1}{\varepsilon_0}\rho(\mathbf{x}) \text{ on } \mathbb{R}^3, \tag{3.5}$$

where Δ is the Laplace operator and ρ has to satisfy

$$\int_\Omega \rho(\mathbf{x})\mathrm{d}x = 1. \tag{3.6}$$

Equation (3.5) can be solved efficiently and accurately, if the right-hand side, $\rho(\mathbf{x})$, is smooth. This does not apply if the potential is induced by N point charges. Here, the charge distribution is a sum of δ-distributions

$$\rho(\mathbf{x}) = \sum_{i=1}^{N} q_i \delta(x - \mathbf{x}_i). \tag{3.7}$$

The δ-distribution is defined by the properties

$$\begin{aligned} \delta(x) &= 0 \text{ for } x \neq 0, \\ \int_{\mathbb{R}^d} \delta(x) &= 1. \end{aligned} \tag{3.8}$$

So since (3.7) is neither smooth nor continuous, another method has to be used to get a good approximation for the solution of (3.5). One way is to introduce a smooth function as replacement for $\delta(\mathbf{x})$, a shield function $\varrho(\mathbf{x})$. $\varrho(\mathbf{x})$ has to satisfy following conditions:

1. $\int_{\mathbb{R}^3} \varrho(\mathbf{x})\mathrm{d}\mathbf{x} = 1$,
2. ϱ is smooth,
3. ϱ is symmetric arround 0 and
4. ϱ has compact support.

An example of an appropriate shielding charge distribution ϱ are Gaussians

$$\varrho(\mathbf{x}) := \left(\frac{\beta}{\sqrt{\pi}}\right)^3 e^{-\beta^2||x||^2}. \tag{3.9}$$

The β specifies the width of the distribution, see figure 3.1. Actually the functions from (3.9) have no compact support and therefore do not satisfy condition 4, which is solved by defining them as

$$\varrho(\mathbf{x}) := \begin{cases} \left(\frac{\beta}{\sqrt{\pi}}\right)^3 e^{-\beta^2||x||^2} & \text{if } x < r_{\text{cut}}, \\ 0 & \text{otherwise.} \end{cases} \tag{3.10}$$

Since Gaussians are decreasing very fast, this can be done without a greater loss of accuracy. The used charge distribution therefore becomes

$$\hat{\rho}(\mathbf{x}) = \sum_{i=1}^{N} q_i \varrho(x - \mathbf{x}_i). \tag{3.11}$$

Another requirement for solving equation (3.5) are appropriate boundary conditions. A common approach is to extend the simulation domain to infinity by periodicity, which allows to split the potential into a smooth and a singular part. Particles in the simulation domain then interact with all other particles in the box as well as with all particles in the periodic images, see figure 3.2. The positions of the particles in the images are given by $\mathbf{x}_j^{\mathbf{n}} = \mathbf{x}_j + (n_1 \cdot L_1, n_2 \cdot L_2, n_3 \cdot L_3)^T$, where $n_1, n_2, n_3 \in \mathbb{Z}^3$ and L_1, L_2, L_3 are the lengths of

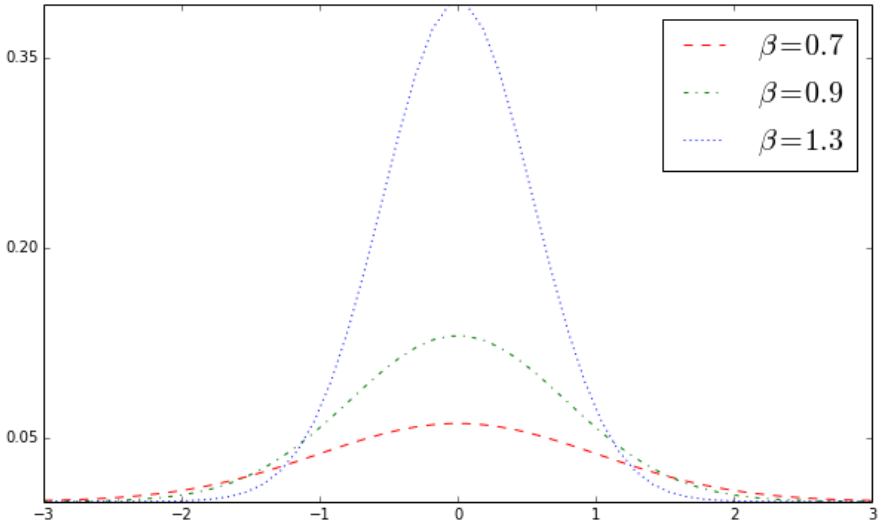

Figure 3.1: Gaussians for different widths β.

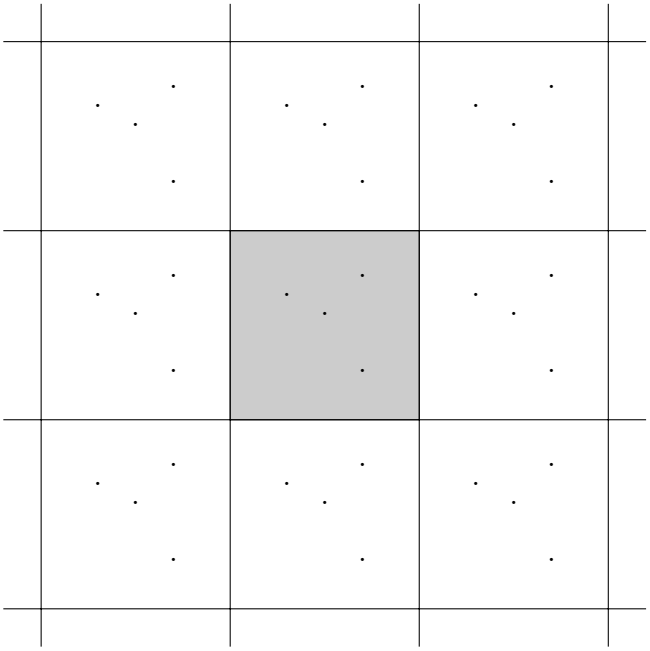

Figure 3.2: Periodic extended simulation domain in $\mathbb{R}^2$.

the simulation domain $\Omega := [0, L_1[\times[0, L_2[\times[0, L_3[$. With such boundary conditions, the given charge distribution results in

$$\rho(\mathbf{x} + (n_1 L_1, n_2 L_2, n_3 L_3)) = \rho(\mathbf{x}), \ \mathbf{n} \in \mathbb{Z}^3 \tag{3.12}$$

still satisfying $\int_\Omega \rho(\mathbf{x}) d\mathbf{x} = 0$ and the total potential in Ω is

$$\Phi(\mathbf{x}) = \sum_{\mathbf{n} \in \mathbb{Z}^3} \Phi_{\mathbf{n}}(\mathbf{x}). \tag{3.13}$$

The electrostatic energie from (3.2) is now given by

$$E^{\text{long}}(\mathbf{x}_i,\ldots,\mathbf{x}_N)=\frac{1}{2}\frac{1}{4\pi\varepsilon_0}\sum_{\mathbf{n}}\sum_{i=1}^{N}\sum_{\substack{j=1\\ i\neq j \text{ for } \mathbf{n}=\mathbf{0}}}^{N} q_iq_j\frac{1}{||\mathbf{x}_i-\mathbf{x}_j+\mathbf{n}||}. \tag{3.14}$$

Now by using charge clouds $\varrho_i^{\mathbf{n}}$, the former introduced shield functions, the charge distribution can be split up into a smooth and singular part. This is done by attaching the charge clouds to the point charges q_i at positions $\mathbf{x}_i$ and then neutralising the resulting shielding by attaching an additional charge cloud with opposite sign. The charge distribution introduced by N point charges and their periodic images can then be written as

$$\rho(\mathbf{x})=\sum_{\mathbf{n}\in\mathbb{Z}^3}\sum_{j=1}^{N} q_j\delta_{\mathbf{x}_j}^{\mathbf{n}}(\mathbf{x}) \tag{3.15}$$

and

$$\begin{aligned}\rho(\mathbf{x})&=(\rho(\mathbf{x})-\rho^{\text{lr}}(\mathbf{x}))+\rho^{\text{lr}}(\mathbf{x})=\rho^{\text{sr}}(\mathbf{x})+\rho^{\text{lr}}(\mathbf{x})\\&=\sum_{\mathbf{n}\in\mathbb{Z}^3}\sum_{j=1}^{N} q_j(\delta_{\mathbf{x}_j}^{\mathbf{n}}(\mathbf{x})-\varrho_{\mathbf{x}_j}^{\mathbf{n}}(\mathbf{x}))+\sum_{\mathbf{n}\in\mathbb{Z}^3}\sum_{j=1}^{N} q_j\varrho_{\mathbf{x}_j}^{\mathbf{n}}(\mathbf{x})\end{aligned} \tag{3.16}$$

Splitting up the electrostatic energy from (3.14) according to (3.16) results in

$$\begin{aligned}E^{\text{long}}&=\frac{1}{2}\frac{1}{4\pi\varepsilon_0}\sum_{\mathbf{n}}\sum_{i=1}^{N}\sum_{\substack{j=1\\ i\neq j \text{ for } \mathbf{n}=\mathbf{0}}}^{N} q_iq_j\int_{\mathbb{R}^3}\frac{\delta_{\mathbf{x}_j^{\mathbf{n}}}(\mathbf{y})-\varrho_{\mathbf{x}_j}^{\mathbf{n}}(\mathbf{y})+\varrho_{\mathbf{x}_j}^{\mathbf{n}}(\mathbf{y})}{||\mathbf{y}-\mathbf{x}_i||}\mathrm{d}\mathbf{y}\\&=E^{\text{sr}}+E^{\text{lr}}\\&=\frac{1}{2}\frac{1}{4\pi\varepsilon_0}\sum_{\mathbf{n}}\sum_{i=1}^{N}\sum_{\substack{j=1\\ i\neq j \text{ for } \mathbf{n}=\mathbf{0}}}^{N} q_iq_j\int_{\mathbb{R}^3}\frac{\delta_{\mathbf{x}_j^{\mathbf{n}}}(\mathbf{y})-\varrho_{\mathbf{x}_j}^{\mathbf{n}}(\mathbf{y})}{||\mathbf{y}-\mathbf{x}_i||}\mathrm{d}\mathbf{y}\\&+\frac{1}{2}\frac{1}{4\pi\varepsilon_0}\sum_{\mathbf{n}}\sum_{i=1}^{N}\sum_{\substack{j=1\\ i\neq j \text{ for } \mathbf{n}=\mathbf{0}}}^{N} q_iq_j\int_{\mathbb{R}^3}\frac{\varrho_{\mathbf{x}_j}^{\mathbf{n}}(\mathbf{y})}{||\mathbf{y}-\mathbf{x}_i||}\mathrm{d}\mathbf{y}.\end{aligned} \tag{3.17}$$

Substituting ρ^{lr} from (3.16) into (3.4) leads to

$$\Phi^{\text{lr}}(\mathbf{x})=\frac{1}{4\pi\varepsilon_0}\sum_{\mathbf{n}}\sum_{j=1}^{N} q_j\int_{\mathbb{R}^3}\frac{\varrho_{\mathbf{x}_j}^{\mathbf{n}}(\mathbf{y})}{||\mathbf{y}-\mathbf{x}||}\mathrm{d}\mathbf{y}. \tag{3.18}$$

Now given the term

$$\sum_{i=1}^{N} q_i\Phi^{\text{lr}}(\mathbf{x}_i)=\frac{1}{4\pi\varepsilon_0}\sum_{\mathbf{n}}\sum_{i=1}^{N}\sum_{j=1}^{N} q_iq_j\int_{\mathbb{R}^3}\frac{\varrho_{\mathbf{x}_j}^{\mathbf{n}}(\mathbf{y})}{||\mathbf{y}-\mathbf{x}_i||}\mathrm{d}\mathbf{y}. \tag{3.19}$$

and comparing it with (3.18) shows, that E^{lr} can be written as

$$E^{\text{lr}}=E_{\text{rec}}+E_{\text{corr}}=\frac{1}{2}\sum_{i=1}^{N} q_i\Phi^{\text{lr}}(\mathbf{x}_i)-\frac{1}{2}\frac{1}{4\pi\varepsilon_0}\sum_{\mathbf{n}}\sum_{i=1}^{N} q_i^2\int_{\mathbb{R}^3}\frac{\varrho_{\mathbf{x}_i}^{\mathbf{0}}(\mathbf{y})}{||\mathbf{y}-\mathbf{x}_i||}\mathrm{d}\mathbf{y}, \tag{3.20}$$

where E_{corr} is a correction term given by

$$E_{\text{corr}} = -\frac{1}{2}\frac{1}{4\pi\varepsilon_0}\sum_{\mathbf{n}}\sum_{i=1}^{N} q_i^2 \int_{\mathbb{R}^3} \frac{\varrho_{\mathbf{x}_i}^{\mathbf{0}}(\mathbf{y})}{||\mathbf{y}-\mathbf{x}_i||}\mathrm{d}\mathbf{y}. \tag{3.21}$$

The correction term removes the nonimaged interactions, which are not calculated but included in E_{rec}. Since E_{corr} does not depend on time directly and $\varrho_{\mathbf{x}_i}^{\mathbf{0}}$ is symmetric around $\mathbf{x}_i$, it can be shown that E_{corr} is constant over time for given $q_1, q_2, \dots q_N$ and has to be computed just once at the beginning of a simulation. So the electrostatic energy can be written as

$$E^{\text{long}} = E^{\text{sr}} + E_{\text{rec}} + E_{\text{corr}}. \tag{3.22}$$

The short-range forces and the reciprocal ones, acting on a particle at position x_i, are then given by

$$\begin{aligned} \mathbf{F}_i^{\text{sr}} &= -\nabla_{\mathbf{x}_i} E^{\text{sr}} \\ \mathbf{F}_{i,\text{rec}} &= -\nabla_{\mathbf{x}_i} E_{\text{rec}} = -\frac{1}{2}\sum_{j=1}^{N} q_j \nabla_{\mathbf{x}_i}\Phi_{\text{rec}}(\mathbf{x}_j) \end{aligned} \tag{3.23}$$

The computation of the short-range part can be done using the linked cell method. For the reciprocal part E_{rec}, one discretizes equation (3.5), for example with the Galerkin method. There, K basis functions ϕ_k are used to set up a linear system of equations $A\mathbf{c} = \mathbf{b}$. With the solution $\mathbf{c} = (c_0, \dots, c_{K-1})^T$, the solution Φ_{rec} gets approximated by a finite sum

$$\Phi_{K,\text{rec}} = \sum_{k=0}^{K-1} c_k \phi_k. \tag{3.24}$$

This results in

$$E_{\text{rec}} = \frac{1}{2}\sum_{i=1}^{N} q_i \Phi_{i,\text{rec}}(\mathbf{x}) \approx \frac{1}{2}\sum_{i=1}^{N} q_i \Phi_{K,\text{rec}}(\mathbf{x}_i) = \frac{1}{2}\sum_{i=1}^{N} q_i \sum_{k=0}^{K-1} c_k \phi_k(\mathbf{x}_i) \tag{3.25}$$

for the reciprocal part of the energy, which can be calculated in $\mathcal{O}(N)$, if the ϕ_k have bounded support. Analogously, the forces are given by the approximation

$$\mathbf{F}_{i,\text{rec}} \approx -\frac{1}{2}\sum_{j=1}^{N} q_j \nabla_{\mathbf{x}_i}\Phi^{K,\text{rec}}(\mathbf{x}_j) = -\frac{1}{2}\sum_{j=1}^{N} q_j \sum_{k=0}^{K-1} \nabla_{\mathbf{x}_i}(c_k\phi_k(\mathbf{x}_j)), \tag{3.26}$$

which are also of order $\mathcal{O}(N)$, if the ϕ_k have local support. Consequently, the following three steps to compute the long-range terms have to be done:

1. Discretization of the potential equation (3.5), e.g. by using the Galerkin method. This leads to a linear system of equation which then can be solved using fast direct methods such as the fast Fourier transform (FFT) [4], or iterative methods like a multigrid or multilevel method. The accuracy of this step depends on the parameters used (e.g. the mesh size for finite element method) and on the type of discretization.

2. Solution of the resulting linear system of equations using FFT or multilevel methods.

3. Computation of the energies and forces. From the resulting approximation of the potential, the reciprocal energy E_{rec} and the forces $\mathbf{F}_{i,\text{rec}}$ can be computed.

By conducting these steps instead of evaluating (3.2) directly, the computational costs can be reduced from $\mathcal{O}(N^2)$ to $\mathcal{O}(N\log(N))$ where the logarithmic factor arises from using the fast Fourier transform. In the next section a special variant of mesh-based methods, the smooth particle mesh Ewald method, will be presented, followed by an implementation of this method, which is a slightly modified version of the implementation presented in [6] and uses OpenCL. The next chapter then compares different approaches for the implementation of step 1.

3.2 The smooth particle mesh-ewald method (SPME)

The SPME method was introduced 1995 and is one of the most popular approaches for calculating Coulombic interactions using the steps described in the previous section. Before elaborating on the implementation of the method in the next section, the basic concepts used for the approximations will be presented. The Gaussians from (3.9) are used as shield functions and B-splines with order > 2 are used as local basis functions ϕ_k. The linear system is solved by using FFTs and the forces are computed as gradients of the approximation of the solution, which can be done directly, since the B-splines are differentiable.
Given the complementary error function

$$\operatorname{erfc}(x) := 1 - \operatorname{erf}(x) = 1 - \frac{2}{\sqrt{\pi}} \int_0^x e^{-y^2} \mathrm{d}y, \tag{3.27}$$

the short-range part E^{sr} from (3.17) can be written as

$$E^{\mathrm{sr}} \approx \frac{1}{2} \frac{1}{4\pi\varepsilon_0} \sum_{\mathbf{n}\in\mathbb{Z}^3} \sum_{i=1}^{N} \sum_{\substack{j=1 \\ j\neq i \text{ for } \mathbf{n}=0 \\ r_{ij}^{\mathbf{n}} < r_{cut}}}^{N} q_i q_j \frac{\operatorname{erfc}(\beta r_{ij}^{\mathbf{n}})}{r_{ij}^{\mathbf{n}}}, \tag{3.28}$$

with the distance $r_{ij}^{\mathbf{n}} := ||\mathbf{x}_j^{\mathbf{n}} - \mathbf{x}_i||$. The forces $\mathbf{F}_i^{\mathrm{sr}}$ are obtained by applying the gradient operator which results in

$$\mathbf{F}_i^{\mathrm{sr}} \approx -\frac{1}{4\pi\varepsilon_0} q_i \sum_{\mathbf{n}\in\mathbb{Z}^3} \sum_{\substack{j=1 \\ j\neq i \text{ for } \mathbf{n}=0 \\ r_{ij}^{\mathbf{n}} < r_{cut}}}^{N} q_j \frac{1}{(r_{ij}^{\mathbf{n}})^2} \left(\operatorname{erfc}(\beta r_{ij}^{\mathbf{n}}) + \frac{2\beta}{\sqrt{\pi}} r_{ij}^{\mathbf{n}} e^{-(\beta r_{ij}^{\mathbf{n}})^2} \right) \frac{\mathbf{r}_{ij}^{\mathbf{n}}}{r_{ij}^{\mathbf{n}}}, \tag{3.29}$$

with the distance vector $\mathbf{r}_{ij}^{\mathbf{n}} := \mathbf{x}_j^{\mathbf{n}} - \mathbf{x}_i$. Computing E^{sr} and $\mathbf{F}_i^{\mathrm{sr}}$ can be done directly with the linked cell method. With the help of the error function, E_{rec} can be written as

$$E^{\mathrm{lr}} = \frac{1}{2} \frac{1}{4\pi\varepsilon_0} \sum_{\mathbf{n}\in\mathbb{Z}^3} \sum_{i=1}^{N} \sum_{\substack{j=1 \\ j\neq i \text{ for } \mathbf{n}=0}}^{N} q_i q_j \frac{\operatorname{erf}(\beta r_{ij}^{\mathbf{n}})}{r_{ij}^{\mathbf{n}}}, \tag{3.30}$$

This term is approximated according to (3.25) and $\mathbf{F}_i^{\mathrm{lr}}$ can be computed with (3.26). The discretization is done by the Galerkin method with trigonometric functions for the basis functions and trial functions. Define the reciprocal lattice vectors $\mathbf{m}$ by $\mathbf{m} = \left(\frac{m_1}{L_1}, \frac{m_2}{L_2}, \frac{m_3}{L_3}\right)^T$ with $m_1, m_2, m_3 \in \mathbb{Z}$ and the simulation domain's volume as $|\Omega| = L_1 \cdot L_2 \cdot L_3$. The reciprocal energy can then be written as

$$E_{\mathrm{rec}} = \frac{1}{2\pi|\Omega|\varepsilon_0} \sum_{\mathbf{m}\neq\mathbf{0}} \frac{\exp(-\pi^2\mathbf{m}^2/\beta^2)}{\mathbf{m}^2} S(\mathbf{m}) S(-\mathbf{m}), \tag{3.31}$$

where $S(\mathbf{m})$ is the structure factor defined by

$$S(\mathbf{m}) = \sum_{j=1}^{N} q_j \exp(2\pi\imath \mathbf{m} \cdot \mathbf{x}_j) = \sum_{j=1}^{N} q_j \exp(2\pi\imath(m_1 x_{j1}/L_1 + m_2 x_{j2}/L_2 + m_3 x_{j3}/L_3)) \tag{3.32}$$

To approximate the structure factors, the complex exponentials of (3.32) are interpolated using B-splines M_p of order p which are defined in one dimension by the recursion

$$\begin{aligned} M_p(x) &= \frac{x}{p-1} M_{p-1}(x) + \frac{p-x}{p-1} M_{p-1}(x-1) \\ M_2(x) &= \begin{cases} 1-|x-1|, & \text{for } x \in [0,2], \\ 0, & \text{otherwise,} \end{cases} \end{aligned} \tag{3.33}$$

and the derivate is given by

$$\frac{dM_p}{dx}(x) = M_{p-1}(x) - M_{p-1}(x-1). \tag{3.34}$$

In fact, the circumstance, that complex exponentials can be interpolated with Euler exponential splines, is used. When the order p is even, one obtains the approximation

$$\exp\left(2\pi\imath \frac{m_i}{K_i} u_i\right) \approx b_i(m_i) \sum_{k=-\infty}^{\infty} M_p(u_i - k) \cdot \exp\left(2\pi\imath \frac{m_i}{K_i} k\right), \tag{3.35}$$

with scaled fractional coordinates $u_i = \frac{K_i}{L_i} \cdot x_i$ and positiv integers K_i, for $i = 1, 2, 3$. The $b_i(m_i)$ from (3.35) are defined by

$$b_i(m_i) = \frac{\exp\left(2\pi\imath(p-1)\frac{m_i}{K_i}\right)}{\sum_{k=0}^{p-2} M_p(k+1) \exp\left(2\pi\imath \frac{m_i k}{K_i}\right)} \tag{3.36}$$

As indicated in [8], the error in this approximation is bounded by $(2\frac{|m_i|}{K_i})^p$, but when p is odd and $2|m_i| = K_i$ this interpolation fails. Now with the approximation above, the approximation for the structure factor $S(\mathbf{m})$ leads to

$$\hat{S}(\mathbf{m}) = b_1(m_1) b_2(m_2) b_3(m_3) F(Q)(m_1, m_2, m_3), \tag{3.37}$$

where F denotes the discrete Fourier transformation and Q is given by

$$Q(k1, k2, k3) = \sum_{\mathbf{n} \in \mathbb{Z}^3} \sum_{i=1}^{N} q_i \prod_{d=1}^{3} M_p((\mathbf{u}_i)_d - k_d - n_d K_d). \tag{3.38}$$

Q is a vector representing the mesh, with the point charges q_i interpolated to it. In the next chapter, different approaches for the implementation of this interpolation will be presented and compared. Now with B and C defined as

$$B(m_1, m_2, m_3) = |b_1(m_1)|^2 \cdot |b_2(m_2)|^2 \cdot |b_3(m_3)|^3 \tag{3.39}$$

and

$$C(m_1, m_2, m_3) = \frac{1}{4\pi^2 |\Omega| \mathbf{m}^2} \exp\left(-\pi^2 \frac{\mathbf{m}^2}{\beta}\right) \text{ for } \mathbf{m} \neq \mathbf{0}, C(0,0,0) = 0, \tag{3.40}$$

the approximate reciprocal energy can be re-expressed by the convolution

$$\begin{aligned} \hat{E}_{\text{rec}} =& \frac{1}{2\pi |\Omega| \varepsilon_0} \sum_{\mathbf{m} \neq \mathbf{0}} \frac{\exp(-\pi^2 \mathbf{m}^2/\beta)}{\mathbf{m}^2} B(m_1, m_2, m_3) \\ & \cdot F(Q)(m_1, m_2, m_3) F(Q)(-m_1, -m_2, -m_3) \\ =& \frac{1}{4\pi\varepsilon_0} \sum_{m1=0}^{K_1-1} \sum_{m2=0}^{K_2-1} \sum_{m3=0}^{K_3-1} Q(m_1, m_2, m_3) \cdot (\theta_{\text{rec}} * Q)(m_1, m_2, m_3), \end{aligned} \tag{3.41}$$

where the pair potential θ_{rec} is given by $\theta_{\text{rec}} = F(B \cdot C)$. This multiplication of $B \cdot C$ with Q in the reciprocal space replaces the Laplacian from (3.5). The reciprocal atomic force can now be obtained by applying the gradient to (3.31) with respect to $\mathbf{x}_i$, resulting in

$$\hat{F}_{\text{rec},i} = \frac{\partial \hat{E}_{\text{rec}}}{\partial (\mathbf{x}_i)_d} = \sum_{m1=0}^{K_1-1} \sum_{m2=0}^{K_2-1} \sum_{m3=0}^{K_3-1} \frac{\partial Q}{\partial (\mathbf{x}_i)_d (m_1, m_2, m_3)} \cdot (\theta_{\text{rec}} * Q)(m_1, m_2, m_3). \tag{3.42}$$

Since Q is differentiable in the particle positions due to the use of B-splines, (3.42) can be computed directly.

An implementation of the currently described method follows in the next section.

3.3 Implementation of the SPME method

The implementation of SPME consists of two parts, the computation of the pairwise E^{sr} of equation (3.28) and the computation of the reciprocal term $\hat{E}_{\text{rec}}$ of (3.41). The evalutation of E^{sr} is integrated in the computation of short-range terms using the linked cell method while the evaluation of $\hat{E}_{\text{rec}}$ is split up into different computational steps. In this section an implementation using OpenCL is presented, where the algorithms were mostly taken from [6]. The main componenents are listed below:

1. *Linked list creation:* The particles are assigned to cells which are used for the linked cell method.

2. *Short-range force computation:* The short-range terms including E^{sr} are computed using the linked cell method.

3. *Charge assignment:* The point charges are spread on to the array Q of equation (3.38). Every particle contributes to n^3 grid points, where the amount depends on the B-spline coefficients $M_n((u_i)_d - j)$, $i = 1, \ldots, N,\ j = 0, \ldots, n-1$.

4. *3DFFT (forward):* Transformation of Q into reciprocal space.

5. *Energy computation:* The reciprocal energy terms are computed by scaling the transformed Q with BC. (3.41)

6. *3DFFT (backward):* Transformation back to real space; computes the convolution θ_{rec}.

7. *Force computation:* application of equation (3.42)

These steps and other implementation details are explained in the following.

The data type for particle positions, forces and velocities is `cl_real4`, a `typedef` of OpenCL's vector types `cl_double4` and `cl_float4`, depending on what precision should be used. Since only three components are used as x, y and z coordinates, the fourth component of each vector is used to save additional data of the particles. The particles point charges are saved as w-component of the position vectors, as are the sigmas at the force vectors and the particles masses at the velocity vectors.

For the creation of the linked lists, two buffers are used, one for the particles and one for the grid cells. The first one is a vector of integers from 0 to $N-1$ and the second one is a vector of length M (number of mesh points) initialized with -1. Then one has to calculate for every particle i, which cell $j, j = 0, \ldots, M$ contains the particle and the values of the two buffers are switched. For example if particle $i = 5$ is in cell $j = 3$, the -1 of the cell buffer is written to the particle id buffer at $i = 5$ and the 5 is written to the cell buffer at location $j = 3$. Since this process runs in parallel for every particle and a cell can contain multiple particles, memory save operations have to be used to avoid race conditions. The following OpenCL kernel therefore uses the `atomic_xchg` function.

```
__kernel void
create_linked_lists(__global int *atom_ids, __global int *cells,
                    __global const real4 *positions, int4 nc, real4 h) {
    size_t gid = get_global_id(0);
    real3 tmp = floor(pos.xyz / spme_cellh.xyz);
    int3 m = ((int3) ((int)tmp.x, (int)tmp.y, (int)tmp.z)) % nc.xyz;
    atom_ids[gid] = atomic_xchg(&cells[I(m, nc)], atom_ids[gid]);
}
```

Listing 3.1: OpenCL kernel to create the linked lists.

The number of grid cells is given by the `int4` vector `nc` and the width of the cells is given by `h`. To calculate a 1D index from a 3D index, the inline function `I` is used.

After the particles were assigned to the grid cells, one can iterate over all particles contained in a cell. If a cell still has the value -1, this implies that the cell does not contain any particles, otherwise the value points to the first particle (or the id of the particle). As an example, the iteration over particles in cell 0 may look like

$$cells[0] = 3 \rightarrow particle\_ids[3] = 7 \rightarrow particle\_ids[7] = 10 \rightarrow particle\_ids[10] = -1.$$

So cell 0 contains the particles with $i \in \{3, 7, 10\}$. Now to compute the short-range terms, one merely has only iterate over a small number of grid cells and their contained particles, see figure 2.2.

```
__kernel void
compute_force_lc(__global const int *atom_ids, __global const int *cells,
                 __global const real4 *positions, int4 nc, real4 h,
                 __global real4 *forces, real4 l, real r_cut, real beta) {
    uint gid = get_global_id(0);
    forces[gid].xyz = 0;
    int3 p;
    int3 m;
    int3 mp;
    real4 pos = positions[gid];
    real3 tmp = floor(pos.xyz / h.xyz);
    m = ((int3) ((int)tmp.x, (int)tmp.y, (int)tmp.z)) + nc.xyz;

    int atom_idx;
    real4 x_pbc; // distance vector for periodic boundary conditions
    real4 neighbor;
    real r, r2, s, f1, f2;
    // iterate over all neighbor cells
    for(p.x = -1; p.x <= 1; ++p.x) {
        for(p.y = -1; p.y <= 1; ++p.y) {
            for(p.z = -1; p.z <= 1; ++p.z) {
                mp = (m + nc.xyz + p) % nc.xzy;
                atom_idx = cells[I(mp, nc)];
                while(atom_idx != -1) {
                    if(atom_idx == gid) {
                        atom_idx = atom_ids[atom_idx];
                        continue;
                    }
                    neighbor = positions[atom_idx];
                    x_pbc = calculate_pbc_point(
                        neighbor,
                        pos,
                        l.xyz
                    );
                    r = length(x_pbc.xyz);
                    if(r <= r_cut) {
                        r2 = r * r;
                        s = (forces[gid].w + forces[atom_idx].w) * 0.5;
                        // short range term for KCL melting
                        f1 = -x.w / r2 * (1 + sign(x.w) *
                             256 * pown(s / r, 8)) / r;
                        // F_sr
                        f2 = -x.w * (erfc(beta * r) / r + beta *
                             M_2_SQRTPI * exp(-(beta * beta * r * r))) / r2;

                        forces[gid].xyz += force(x_pbc, s, r_cut, beta);
                    }
                    atom_idx = atom_ids[atom_idx];
                }
            }
        }
    }
}
```

Listing 3.2: Computation short range force terms using the linked cell method.

Here, `l` is a vector type holding the lengths of the simulation domain, `r_cut` is the cutoff radius and `beta` is the width of the Gaussian. The kernel iterates over the cell containing the current particle and the eight neighbor cells. To calculate the distance between two particles, periodic boundary conditions are taken into account. If the distance between the particles is greater than the cutoff radius, no force is calculated.

The next step is the charge assignment, where each point charge q_i is spread over n^3 mesh points. For every particle, the $3n$ B-spline coefficients are calculated, which are used to compute how much q_i contributes to each of the n^3 mesh points. In the naive implementation used here, one thread is used for every point charge, so race conditions occur, when different threads attempt to accumulate charge to the same mesh point. Therefore an atomic memory operation is used to add the charge to the grid, which is implemented as an atomic compare-and-set loop, since `atomic_add` does not exist for floating point datatypes. The following chapter analyzes the charge assignment step in more detail. For the naive implementation, the OpenCL kernel may look like,

```
__kernel void
spread_charge(__global real4 *positions, int4 nc,
              __global real *Q, real4 mesh_h) {
    uint gid = get_global_id(0);
    real array[PMAX * PMAX];
    real bspx[PMAX];
    real bspy[PMAX];
    real bspz[PMAX];

    int3 p, mp, m;
    real4 pos = positions[gid];
    real3 tmp = floor(pos.xyz / mesh_h.xyz);
    m = ((int3) ((int)tmp.x, (int)tmp.y, (int)tmp.z)) + nc.xyz;

    real w = fmod(pos.x, mesh_h.x) / mesh_h.x;
    compute_bspline_point(bspx, w, array);

    w = fmod(pos.y, mesh_h.y) / mesh_h.y;
    compute_bspline_point(bspy, w, array);

    w = fmod(pos.z, mesh_h.z) / mesh_h.z;
    compute_bspline_point(bspz, w, array);

    for(p.x = 0; p.x < PMAX; ++p.x) {
        real t = bspx[PMAX - 1 - p.x];
        for(p.y = 0; p.y < PMAX; ++p.y) {
            real u = bspy[PMAX - 1 - p.y];
            for(p.z = 0; p.z < PMAX; ++p.z) {
                real v = bspz[PMAX - 1 - p.z];
                mp = (m - p) % nc.xyz;
                real contribution = pos.w * t * u * v;
                _atomic_add(&Q[I(mp, nc)], contribution);
            }
        }
    }
}
```

Listing 3.3: OpenCL kernel for per particle charge spreading.

where `mesh_h` contains the widths of the mesh and `Q` is the array Q of (3.38). The function `_atomic_add` implements an atomic addition for floating point values using a compare-and-set loop, see listing 4.1. PMAX is the maximum spline order, which is a macro given to the OpenCL program, when building it, as a compile flag.

To perform the following 3DFFTs, the clFFT library [1] is used. For simplicity, two complex-to-complex 3D transforms are performed instead of one inplace interleaved real-to-hermitian and one inplace interleaved hermitian-to-real, which would probably more efficient. The complex-to-complex transformations need two buffers, one holding the real values and one holding the complex ones.

The energy terms are calculated as product of the transformed Q with the product $B \cdot C$, which is calculated only once at the beginning of the simulation. This data can be held in *constant* memory, if the data is not to big. The GPU's constant memory is a small (64KB), read-only region of global memory but with cached read access. Here, $B \cdot C$ is not stored in constant memory due to the constant memory size limitation. A simple kernel then looks like:

```
__kernel void
compute_rec_energy(__global real *Q, __global real *Qimag,
                   __global real *BC) {
    size_t gid = get_global_id(0);
    Q[gid] *= BC[gid];
    Qimag[gid] *= BC[gid];
}
```

Listing 3.4: Simple OpenCL kernel to compute reciprocal energy.

The last step covers the computation of the forces at the particle positions from the solution Q of the Poisson equation. This is done by applying the gradient, see equation (3.42). Therefore the $3n$ B-splines are computed once more together with their first derivatives. So this time the coefficients are of type `real2`, where the derivative is given by the y-component.

```
__kernel void
compute_force_spme(__global real4 *positions, __global real *Q,
                   __global real4 *forces, int4 nc, real4 mesh_h) {
    size_t gid = get_global_id(0);
    int3 p, mp, m;
    real q;
    real4 pos = positions[gid];
    real3 tmp = floor(pos.xyz / mesh_h.xyz);
    m = ((int3) ((int)tmp.x, (int)tmp.y, (int)tmp.z)) + nc.xyz;

    real array[PMAX * PMAX];
    real2 bspx[PMAX];
    real2 bspy[PMAX];
    real2 bspz[PMAX];

    real w = fmod(pos.x, mesh_h.x) / mesh_h.x;
    compute_bspline_point2(bspx, w, array);

    w = fmod(pos.y, mesh_h.y) / mesh_h.y;
    compute_bspline_point2(bspy, w, array);

    w = fmod(pos.z, mesh_h.z) / mesh_h.z;
    compute_bspline_point2(bspz, w, array);

    for(p.x = 0; p.x < PMAX; ++p.x) {
        real2 t = bspx[PMAX - 1 - p.x];
        for(p.y = 0; p.y < PMAX; ++p.y) {
            real2 u = bspy[PMAX - 1 - p.y];
            for(p.z = 0; p.z < PMAX; ++p.z) {
                mp = (m - p) % nc.xyz;
                q = pos.w * Q[I(mp, nc)];
                real2 v = bspz[PMAX - 1 - p.z];
                forces[gid].xyz -= q *
                                   (real3)(t.y * u.x * v.x,
                                           t.x * u.y * v.x,
                                           t.x * u.x * v.y) *
                                   mesh_h.xyz;
            }
        }
    }
}
```

Listing 3.5: OpenCL kernel to compute the reciprocal force terms for each particle.

Iterating over the n^3 mesh points again, the computed long-range force terms are simply added to the already computed short-range terms.

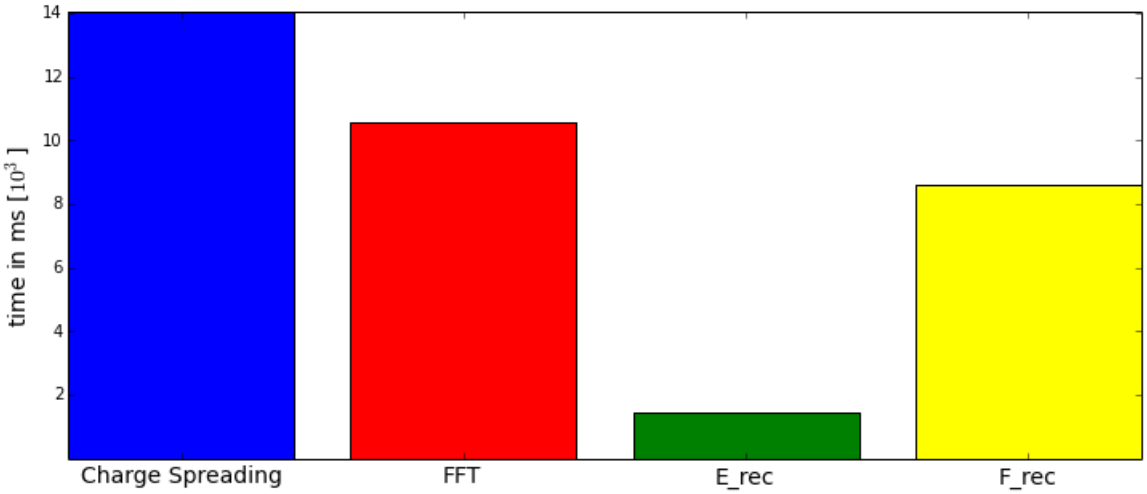

Figure 3.3: Bar plot for the different steps of reciprocal force computation.

The code, that has just been presented was run for a potassium chloride (KCL) crystal with the following configuration: 32768 particles were used, in a simulation domain with 90^3 mesh points and a B-spline order of 4. The cutoff radius was set to 24. In the first 25 steps, the system gets heated up by scaling the velocities with a scaling value of 1.08 and then cooled down after the 625th step, for again 25 steps with a scaling value of 0.92. For the last 600 steps, no scaling is applied. The force term for the used potential for the short range forces is given by

$$F_{ij} = -\frac{q_i q_j}{(r_{ij})^2}\left(1 + \text{sign}(q_i q_j)\cdot 2^8\left(\frac{\sigma_{ij}}{r_{ij}}\right)^8\right)\frac{\mathbf{r}_{ij}}{r_{ij}} \tag{3.43}$$

Parameters values used in the simulaion are given by table 3.1. Additional information can be

$\sigma_K = 2.1354$	$\sigma_{CL} = 2.9291$
$m_K = 38.9626$	$m_{CL} = 35.4527$
$q_K = 1$	$q_{CL} = -1$
$\beta = 0.1$	$\delta t = 1.0$

Table 3.1: Parameter values for the KCL simulation

found in [6]. The whole simulation was run for 1250 steps and the times for the different kernels was measured, see figure 3.3. In the computation of the reciprocal energy and force terms As can be seen from the figure, nearly 20% of the time needed for computing the reciprocal part is used for the charge spreading. More subtle ways of computing the charge assignment would be preferable and different approaches are compared in the next chapter.

Chapter 4

Comparison of different approaches for charge assignment

In the last chapter, the SPME method together with an implementation using OpenCL was presented and it became obvious that a significantly amount of the computational time was used for the interpolation of the point charges to the mesh points. This chapter gives a deeper insight in this particular step, presents different implementations and compares them regarding efficiency and accuracy.

The charge assignment is governed by the Gaussian equation

$$\rho(\mathbf{m}) = \left(\frac{\beta}{\sqrt{\pi}}\right)^3 \sum_{\substack{j=1 \\ ||\mathbf{m}-\mathbf{x_j}||<r_{\text{cut}}}} q_j \exp(-\beta^2||\mathbf{m}-\mathbf{x_j}||^2), \tag{4.1}$$

where $\mathbf{m}$ denotes a gridpoint and q_j is the point charge of particle j at position $\mathbf{r}_j$. The coefficient β is the width of the Gaussian and r_{cut} is once more the cutoff radius. Then $\rho(\mathbf{m})$ is the complete charge distribution on the grid at location $\mathbf{m}$. Now the following algorithm has to be implemented:

Algorithm 1 Charge summation

```
for i in 1 to N do
    for m in N(x_i) do
        Q(m) ← Q(m) + ρ(x_i)
    end for
end for
```

$\mathcal{N}(\mathbf{x}_i)$ denotes the neighborhood of particle i at position $\mathbf{x}_i$ and is defined by $\mathcal{N}(\mathbf{x}_i) = \{\mathbf{m}|||\mathbf{m}-\mathbf{x}_i|| < r_{\text{cut}}\}$. In the following sections, different parallel implementations for algorithm (1) will be described, whereby the first two methods use B-Splines to approximate $\rho(\mathbf{x_i})$. The first one is the straightforward implementation as a charge spreading by parallelizing the outer loop whereas the second approach uses an per grid point charge gathering, so the two loops are basically switched. In the third approach, a discrete Gaussian filter is used to approximate (4.1) and to spread the charges. The tests were run on a *Nvidia GeForce 580GTX* and the parameters were taken from [6]. For an increasing number of particles, the size of the cubic domain is increased by the same factor, keeping the particle density constant, see table 4.1.

The particles are positioned uniformly over the domain, like in a typical start configuration for MD simulations.

particles	1728	4096	8000	17576	32768	64000	140608	262144	592704	1191016
domain length	96	144	192	240	288	360	480	576	768	960
mesh points	24^3	36^3	48^3	60^3	72^3	90^3	120^3	144^3	192^3	240^3

Table 4.1: Parameters for test runs taken from [6].

4.1 Naive per particle charge spreading

Like already mentioned, the implementation of this approach is straightforward and was presented in the last chapter, see listing 3.3. One thread is used to spread the charge for each particle to the gridpoints in the neighborhood $\mathcal{N}(\mathbf{x_i})$, where the neighborhood consists of n^3 gridpoints for B-spline order n. This parallel charge assignment leads to synchronisation problems, when different threads attempt to add their contribution to the same grid location, see figure 4.1. Therefore thread-safe atomic memory operations have to be used to avoid this

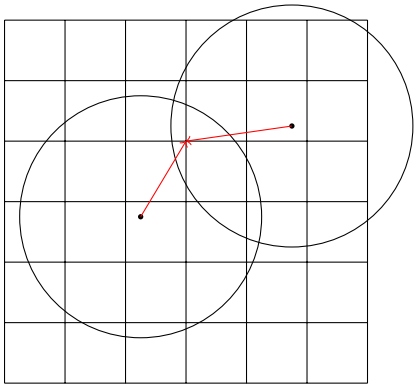

Figure 4.1: Synchronisation problem for parallel charge assignment in 2D

problem. Since atomic additions in OpenCL only exist for integer values, one either has to emulate a floating-point atomic addition by using an atomic compare-exchange loop or has to change the implementation to use fixed-point arithmetic. The implemantation used here looks like:

```
void
_atomic_add(volatile __global real *source, const real operand)
{
    union {
        unsigned int i;
        real r;
    } new_value;

    union {
        unsigned int i;
        real r;
    } prev_value;

    do {
        prev_value.r = *source;
        new_value.r = prev_value.r + operand;
    } while(
            atomic_cmpxchg((volatile __global unsigned int *) source,
                           prev_value.i, new_value.i) != prev_value.i);
}
```

Listing 4.1: Emulation of atomic_add for floating point values using atomic_xchg

Figure 4.2 shows the amount of time needed for the `_atomic_add`. Ignoring the smaller

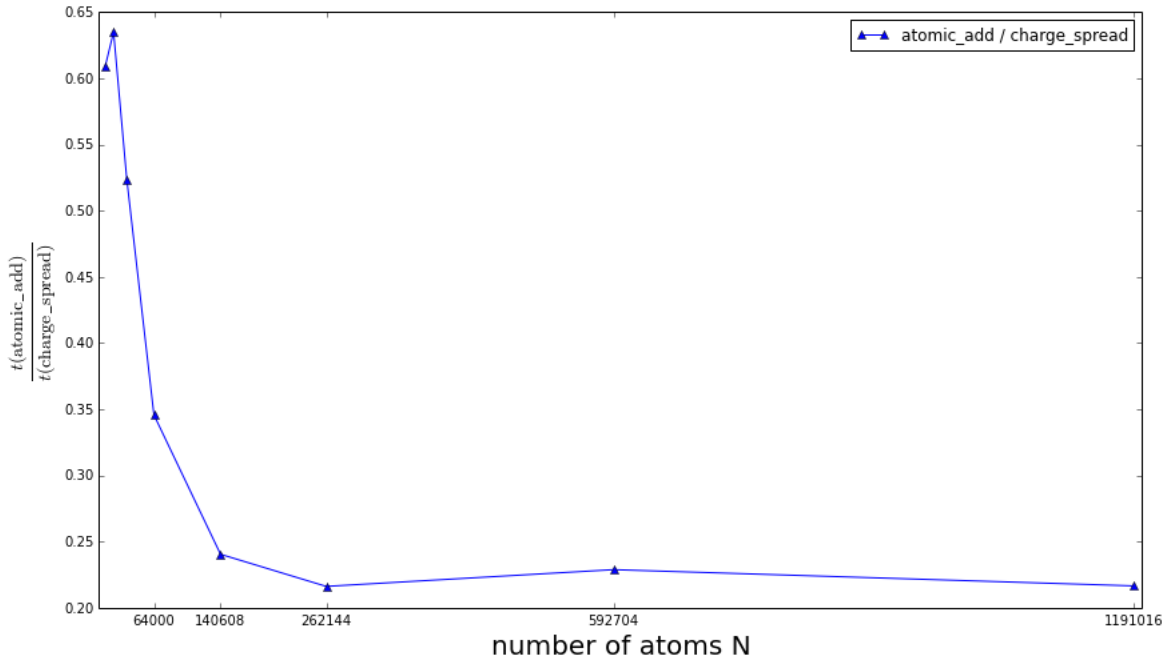

Figure 4.2: The amount of time needed for _atomic_add for different numbers of particles.

setups, the atomic compare-exchange loop needs over 20% of the time needed to compute the charge spreading, so algorithms avoiding atomic operations are preferable. In addition to the necessity of using these slow atomic operations, this approach has an essentially unordered memory access pattern to the uncached global memory, which leads to poor performance. It is preferable for threads in a workgroup to access memory in contiguous address ranges, which is examined in the next section.

4.2 Per grid point charge gathering

To avoid the use of atomic operations the point charges are not spread to the n^3 grid points for each particle. Instead a per grid point charge gathering is done. Therefore, particles have to be assigned to grid points, in the following referred to as grid cells for clarity. This is done the same way as in the case of the linked cell method. One can then iterate for every cell over the n^3 neighbors (including the current cell) and over the particles the cell contains. Now using three work item dimensions and linked lists, the implementation looks as follows:

```
__kernel void
linked_list(__global real4 *positions, int4 nc, __global real *Q,
            real4 h, __global int *atom_ids, __global int *cell_ids) {
    int3 g = (int3) (get_global_id(0),
                     get_global_id(1),
                     get_global_id(2));
    uint cell_idx = I(g, nc);
    int neighbor_idx;
    int particle_id;
    real array[PMAX * PMAX];
    real bspx[PMAX];
    real bspy[PMAX];
    real bspz[PMAX];
    int3 p;
    int3 m;
    real4 pos;
    real w;
    Q[cell_idx] = 0.0;
```

```
    real sum = 0.0;
    for(p.x = 0; p.x < PMAX; ++p.x) {
        m.x = (g.x + p.x) % nc.x;
        for(p.y = 0; p.y < PMAX; ++p.y) {
            m.y = (g.y + p.y) % nc.y;
            for(p.z = 0; p.z < PMAX; ++p.z) {
                m.z = (g.z + p.z) % nc.z;
                neighbor_idx = I(m, nc);
                particle_id = cell_ids[neighbor_idx];
                while(particle_id != -1)
                {
                    pos = positions[particle_id];
                    w = fmod(pos.x, h.x) / h.x;
                    compute_bspline_point1(bspx, w, array);

                    w = fmod(pos.y, h.y) / h.y;
                    compute_bspline_point1(bspy, w, array);

                    w = fmod(pos.z, h.z) / h.z;
                    compute_bspline_point1(bspz, w, array);
                    sum += pos.w * bspx[PMAX - 1 - p.x]
                                 * bspy[PMAX - 1 - p.y]
                                 * bspz[PMAX - 1 - p.z];
                    particle_id = atom_ids[particle_id];
                }
            }
        }
    }
    Q[cell_idx] += sum;
}
```

Listing 4.2: Per grid cell charge gathering using linked lists.

Now each thread computes the charge of one cell, which is given by the three global ids, one for each dimension. Using these indices, the code iterates over the $n^3 = \text{PMAX}^3$ neighboring cells and then over their assigned particles, for which the B-splines are calculated. To avoid unnecessary accesses to the global Q array, a temporary variable `sum` is used to accumulate the charge contributions.

In typical configurations for molecular dynamics simulations, the number of cells is significantly higher than the number of particles, so the system is very sparse. So one cell mostly contains only one particle, making the iteration over the linked lists almost unnecessary. In the following, each cell is permitted to hold only a single charge and additional charges are placed in a separate list. In fact, the current code that assignes the particles to cells can be used without any changes. From the `cell_ids` buffer, one can obtain the particle ids assigned to a cell. The particles which could not be added to a cell can be read from the `atom_ids` buffer. This variant can be implemented like in listing 4.2 by replacing the `while`-loop with an `if`-statement and removing the code for the linked lists. For the additional charges the naive charge spreading kernel is used, by adding a check, if `atom_ids[gid] != -1`.

A further improvement is the use of local memory. The kernel is now executed in groups of $nc.x$ work items operating on a single row along the x-axis. Each work item first iterates over n^2 cells in the $y-z$-plane and loads the particle ids from the global `cell_ids` buffer into a local one from which the ids may be accessed by other threads of this work group, see figure 4.3. A probably more efficient way would be to store the $3n$ B-spline coefficients in the local memory for shared use, reducing their number of computations from n^3 to n^2. This approach was not used here, due to the limitations of the local memory size. The kernel useing the local memory for the `cell_ids` buffer then looks like:

```
__kernel void
accumulate_charges(__global real4 *positions, int4 nc,
                   __global real *Q, real4 h, __global int *atom_ids,
                   __global int *cell_ids) {
```

```
    uint gx = get_local_id(0);
    uint gy = get_global_id(1);
    uint gz = get_global_id(2);
    uint cell_idx = gx + nc.x * (gy + nc.y * gz);
    int neighbor_idx;
    int particle_id;

    real array[PMAX * PMAX];
    real bspx[PMAX];
    real bspy[PMAX];
    real bspz[PMAX];

    __local int cells_local[ROW_LENGTH][PMAX][PMAX];

    real4 pos;
    real w;

    int3 m, p;
    m.x = gx;

    Q[cell_idx] = 0.0;

    for(p.y = 0; p.y < PMAX; ++p.y) {
        m.y = (gy + p.y) % nc.y;
        for(p.z = 0; p.z < PMAX; ++p.z) {
            m.z = (gz + p.z) % nc.z;
            neighbor_idx = I(m, nc);
            cells_local[gx][p.y][p.z] = cell_ids[neighbor_idx];
        }
    }
    barrier(CLK_LOCAL_MEM_FENCE);

    real sum = 0.0;
    for(p.x = 0; p.x < PMAX; ++p.x) {
        m.x = (gx + p.x) % nc.x;
        for(p.y = 0; p.y < PMAX; ++p.y) {
            m.y = (gy + p.y) % nc.y;
            for(p.z = 0; p.z < PMAX; ++p.z) {
                m.z = (gz + p.z) % nc.z;
                particle_id = cells_local[m.x][p.y][p.z];
                if(particle_id != -1)
                {
                    pos = positions[particle_id];
                    w = fmod(pos.x, h.x) / h.x;
                    compute_bspline_point1(bspx, w, array);

                    w = fmod(pos.y, h.y) / h.y;
                    compute_bspline_point1(bspy, w, array);

                    w = fmod(pos.z, h.z) / h.z;
                    compute_bspline_point1(bspz, w, array);
                    sum += pos.w * bspx[PMAX - 1 - p.x]
                                 * bspy[PMAX - 1 - p.y]
                                 * bspz[PMAX - 1 - p.z];
                }
            }
        }
    }
    Q[cell_idx] += sum;
}
```

Listing 4.3: Per grid cell charge gathering with local memory improvement.

After loading the n^2 ids into the local memory, the threads of the work group have to be synchronized, which is done by using a barrier. During the iteration over the n^3 neighbor cells, the particle ids can now be loaded from the local array. The `ROW_LENGTH` constant is passed to the kernel as compile flag like `PMAX`, so that it has the size of the array at compile time. The plot in figure 4.4 shows the different runtimes for the different variants of the per grid point gathering. Despite of the improvement of the charge gathering by using a hybrid

approach and local memory, the naive charge spreading is much faster on the used GeForce GTX 580.

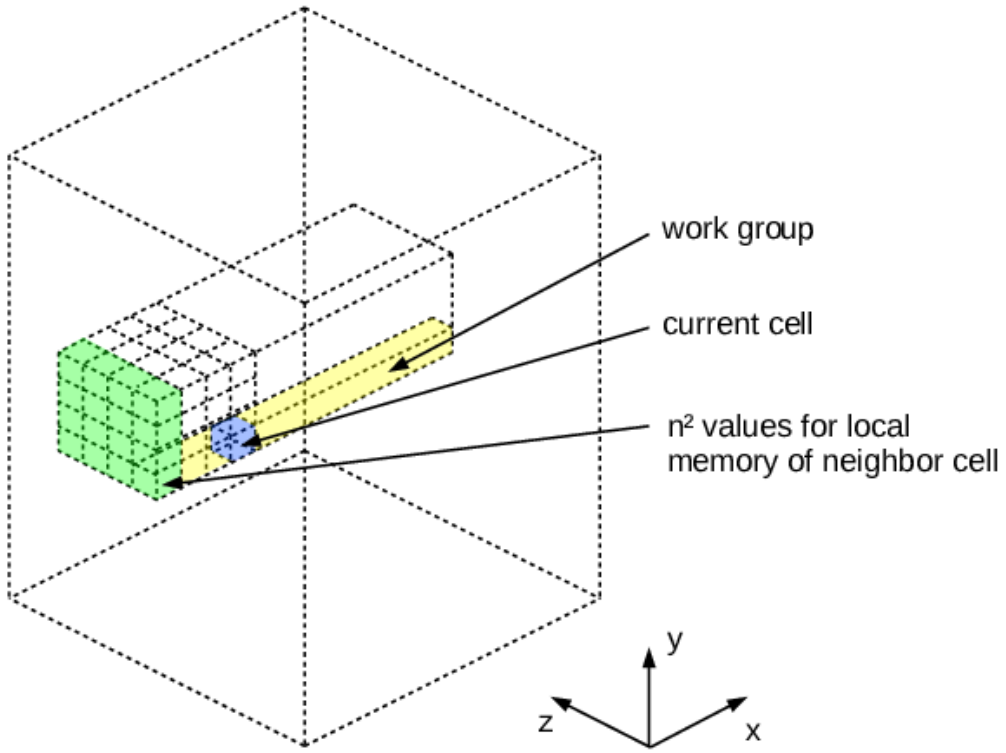

Figure 4.3: 3D mesh grid, showing the current cell (blue) processed by one work item of a work group along the x-axis (yellow). Every work item loads n^2 values into local memory (green), which can be accessed by other work items of the work group.

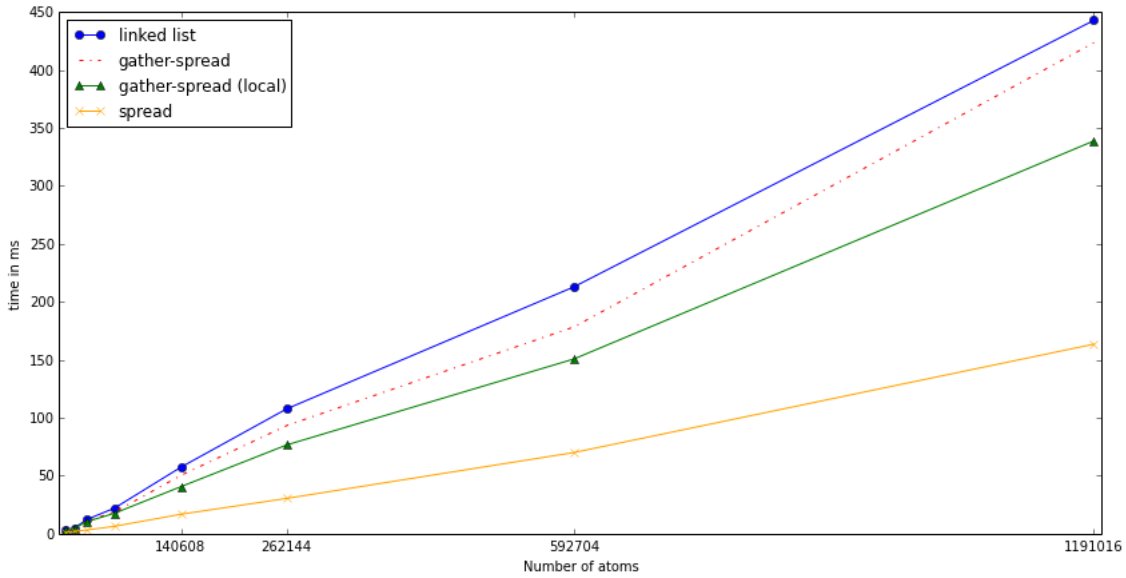

Figure 4.4: Plot for different variants of charge gathering.

4.3 Gaussian filtering for charge spreading

In the next approach, the approximation is not performed by using B-splines. The point charges are spread over the grid by applying a Gaussian filter which is defined in 1D by

$$g_{\beta,h} = \left[\exp(-\beta^2 h^2), \exp(-\beta^2 \cdot 0), \exp(-\beta^2 h^2)\right], \tag{4.2}$$

where β is the width of the Gaussian and h is the width of the mesh, assuming the same width for all dimensions. In 3D, this filter could either be applied three times or a full 3D filter could be used, as Gaussian filters are separable.

The approach in this section consists of two steps, the interpolation of the charges to the eight direct neighbors and the filtering of the mesh. For the interpolation the function $\rho(\mathbf{x})$ from (4.1) is used and has to be multiplied by the volume of a cell. So the interpolation of the particles to the lattice is given by

$$G_\beta(\mathbf{x}_i) = h^3 \left(\frac{\beta}{\sqrt{\pi}}\right)^3 \sum_{\substack{\mathbf{m} \\ ||\mathbf{m}-\mathbf{x_i}||<r_{\text{cut}}}} \exp(-\beta^2 ||\mathbf{m} - \mathbf{r_i}||^2). \tag{4.3}$$

r_{cut} and β have to be chosen in such a way, that for given grid spacing h it holds that

$$G_\beta(\mathbf{x}_i) = 1. \tag{4.4}$$

$G_\beta(\mathbf{x}_i)$ can now be split up by splitting the exponential function:

$$G_\beta(\mathbf{x}_i) = G_{\beta_1}(\mathbf{x}_i) \cdot G_{\beta_2}(\mathbf{x}_i), \tag{4.5}$$

satisfying

$$\beta^2 = \beta_1^2 + \beta_2^2. \tag{4.6}$$

Now using a Gaussian filter with size $k = \lfloor \frac{\text{PMAX}+1}{2} \rfloor$, $G_{\beta_2}(\mathbf{x}_i)$ is approximated by

$$G_{\beta_2}(\mathbf{x}_i) \approx \left[G_{\beta_k}\right]^k. \tag{4.7}$$

Note that the filter is of odd size, while for B-splines, even orders are used. Applying the filter of size k is equivalent to applying a filter of size 3 n times, for $k = 2n + 1 \Leftrightarrow n = \frac{k-1}{2}$ and according to (4.6) the width for n filters is

$$\beta^2 = n\beta_n^2 \Longrightarrow \beta_n = \frac{1}{\sqrt{n}}\beta_2. \tag{4.8}$$

Therefore, (4.5) is given by the approximation

$$G_\beta(\mathbf{x}_i) \approx G_{\beta_1}(\mathbf{x}_i) * \left[G_{\beta_n}\right]^n. \tag{4.9}$$

Since the filter does not change, it is computed once and then stored in constant memory, which is faster than uncached global memory. In the following, a full 3D filter is used for the implementation.

```
__kernel void
spread_charge_gauss(__global const real4 *positions, int4 nc,
                    __global real *Q, real4 h, real beta) {
    uint gid = get_global_id(0);
    real4 pos = positions[gid];
    int3 m, mp;
    real w, r;
    real3 l_box = h.xyz * (real3)((real) nc.x, (real) nc.y, (real) nc.z);
    real V = h.x * h.y * h.z * pos.w;
```

```
        real3 mtmp = floor(pos.xyz / h.xyz);
        m = (int3) (mtmp.x, mtmp.y, mtmp.z) + nc.xyz;
        int3 p;
        int3 mm;
        for(p.x = HALF_FILTER_SIZE; p.x < 2 + HALF_FILTER_SIZE;  ++p.x) {
            for(p.y = HALF_FILTER_SIZE; p.y < 2 + HALF_FILTER_SIZE; ++p.y) {
                for(p.z = HALF_FILTER_SIZE; p.z < 2 + HALF_FILTER_SIZE; ++p.z) {
                    mm = (m - p + (int3)(HALF_FILTER_SIZE) + nc.xyz) % nc.xyz;
                    r  = distance_to_point(pos.xyz, mm, nc.xyz, l_box);
                    mp = (m - p + nc.xyz) % nc.xyz;
                    w = V * gauss(beta, r);
                    _atomic_add(&Q[I(mp, nc)], w);
                }
            }
        }
    }
```

Listing 4.4: 1. Step of Gaussian filter approach.

The spreading is done by using the Gaussian function from (4.1) and the summation uses the atomic function from listing 4.1. `HALF_FILTER_SIZE` is again a macro constant, given to the kernel as compile flag. With this constant, the contributions are shifted, so that the cells, being in the center, are the same as in the B-spline approaches, see figure 4.5. The function

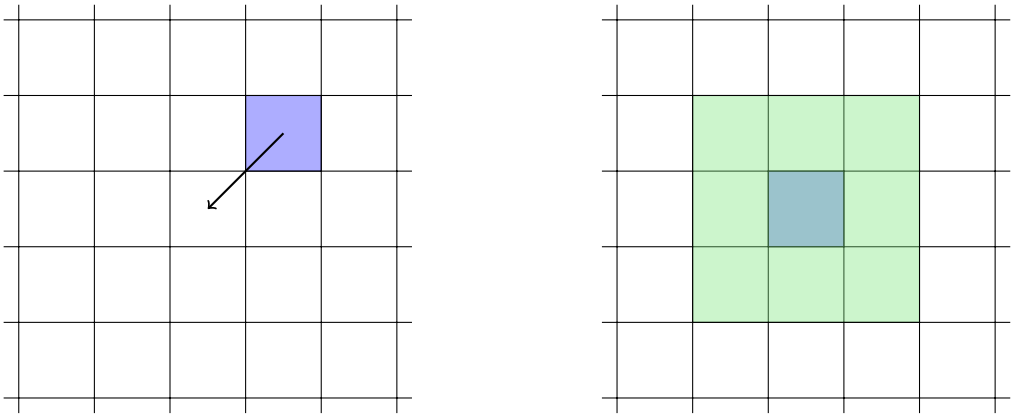

Figure 4.5: 1. Shifting the filter with half filter size $\frac{k}{2} = 1$ in each dimension. 2. The shifted center and the grid cells getting filtered.

`distance_to_point` computes the distance of a mesh point to a particle and takes periodic boundary conditions into account. The result of the first step is now used as input for the filter kernel and the filtered mesh is written to a separate buffer. Each thread iterates over k^3 cells of the input buffer, multiplying their values with the associated values of the filter and adding the result to the output buffer.

```
__kernel void
filter_charge_array_naive(__global real *Q, __global real *Q_out,
                          __constant real *filter, int4 nc) {
    uint gx = get_global_id(0);
    uint gy = get_global_id(1);
    uint gz = get_global_id(2);

    int3 p;
    int3 m;
    real sum = 0.0;
    for(p.x = 0; p.x < FILTER_SIZE; ++p.x) {
        m.x = (gx - p.x + nc.x) % nc.x;
        for(p.y = 0; p.y < FILTER_SIZE; ++p.y) {
            m.y = (gy - p.y + nc.y) % nc.y;
            for(p.z = 0; p.z < FILTER_SIZE; ++p.z) {
                m.z = (gz - p.z + nc.z) % nc.z;
                sum += Q[I(m, nc)] * filter[Idx(p, FILTER_SIZE)];
```

```
            }
        }
    }
    Q_out[gx + nc.x * (gy + nc.y * gz)] = sum;
}
```

Listing 4.5: A simple filter kernel without optimizations.

Here, `Idx` is just a helper function to calculate a 1D index from 3D coordinates and `FILTER_SIZE` is the size of the filter, k, given to the kernel as compile flag. Multiple threads have to access the same global memory addresses, so using local memory is an obvious improvement like in the charge gathering kernel. Again `nc.x` threads are put in one work group, each loading k^2 values of the charge array into local memory. The filtering is done in a second step after the synchronisation barrier, where then all values can be taken from local and constant memory.

```
__kernel void
filter_charge_array(__global real *Q, __global real *Q_out,
                    __constant real *filter, int4 nc) {
    uint gx = get_local_id(0);
    uint gy = get_global_id(1);
    uint gz = get_global_id(2);
    __local real Q_local[ROW_LENGTH][FILTER_SIZE][FILTER_SIZE];
    int3 m, p;
    m.x = gx;
    for(p.y = 0; p.y < FILTER_SIZE; ++p.y) {
        m.y = (gy - p.y + nc.y) % nc.y;
        for(p.z = 0; p.z < FILTER_SIZE; ++p.z) {
            m.z = (gz - p.z + nc.z) % nc.z;
            Q_local[gx][p.y][p.z] = Q[I(m, nc)];
        }
    }
    barrier(CLK_LOCAL_MEM_FENCE);
    real sum = 0.0;
    for(p.x = 0; p.x < FILTER_SIZE; ++p.x) {
        m.x = (gx - p.x + nc.x) % nc.x;
        for(p.y = 0; p.y < FILTER_SIZE; ++p.y) {
            for(p.z = 0; p.z < FILTER_SIZE; ++p.z) {
                sum += Q_local[m.x][p.y][p.z] * filter[Idx(p, FILTER_SIZE)];
            }
        }
    }
    Q_out[gx + nc.x * (gy + nc.y * gz)] = sum;
}
```

Listing 4.6: Filter Kernel using local memory.

The plot in figure 4.6 shows the runtimes for the two implementations. For big setups, the implementation with local memory outperforms the naive version, e.g. for the setup with 1191016 particles it needs only 60% of the time.

Although, the implementation using the Gaussian filtering approach is faster than the naive charge spreading one, an important aspect is not taken into account here, the accuracy of the different variants. In the SPME method presented in the previous chapter, the same interpolation scheme is used for charge assignment and back interpolation. Whether or not it is possible just to replace the assignment using B-splines with Gaussian filters is tested in the following. The full SPME implementation will be run with different configurations for the different kernels for charge assignment, comparing their runtimes in a real simulation with changing particle positions.

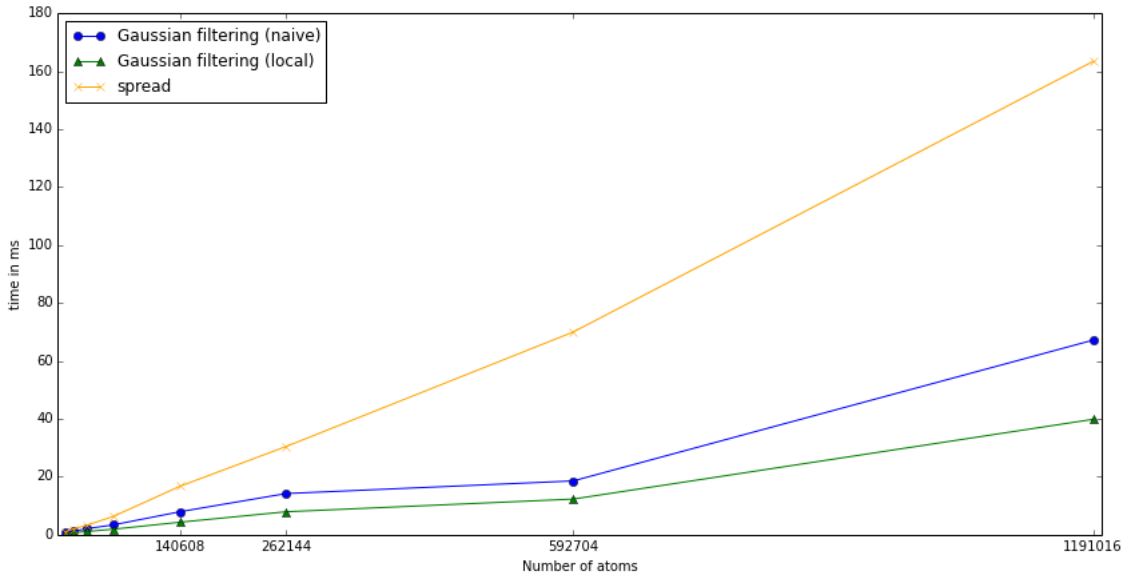

Figure 4.6: Plot for different variants of charge gaussian filtering compared with naive charge spreading.

4.4 Performance and accuracy in real simulations

In this section the above explained approaches for charge assignment are integrated in a full simulation, simulating a KCL crystal with different number of atoms and different number of mesh points. The simulation domain and the number of mesh points are chosen such that the density of the particles keeps constant, see table 4.2. To test the accuracy of the Gaussian filtering against the naive charge spreading, the energy relative errors ϵ_R defined as

$$\epsilon_R = \frac{|E_{\text{rec}} - \hat{E}_{\text{rec}}|}{|E_{\text{rec}} + E^{\text{sr}} - \frac{\beta}{\sqrt{\pi}} \sum_{i=1}^{N} q_i^2|} \tag{4.10}$$

were compared.

particles	1728	8000	17576	32768	64000	140608	262144	592704
domain length	144	240	288	360	480	576	768	960
mesh points	36^3	60^3	72^3	90^3	120^3	144^3	192^3	240^3

Table 4.2: Parameters for full simulation test runs.

With regard to the performance of the different approaches in a full simulation, one gets the same results as the separated test runs in the previous sections, see figure 4.7. The gathering approach needs much more time than the other two approaches, while the Gaussian filtering is faster than the naive charge spreading. Now the question is, how accurate the solution with Gaussian filtering is. Therefore, for all configurations up to 140608 particles from table 4.2, the relative error of 4.10 was calculated for both charge assignment variants, the filtering and the naive spreading, see figure 4.8. It can be seen, that the error of the filtering approach is much larger than the error of the standard B-spline approach. ϵ_{gauss} also varies a lot, while ϵ_{naive} remaines constant. This effect can be seen even more clearly in figures 4.9 and 4.10, where the relative error was computed every 30th (and 60th respectively) time step. So even

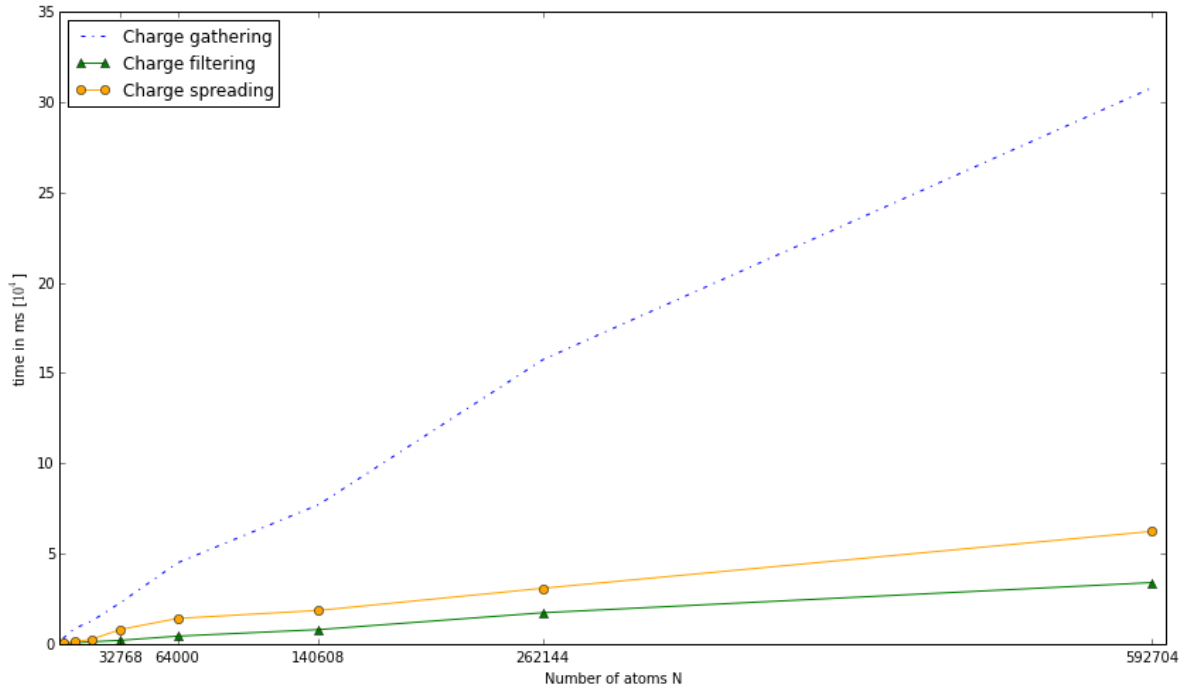

Figure 4.7: Runtimes for the different charge assignment kernels in the KCL simulation for 1250 time steps.

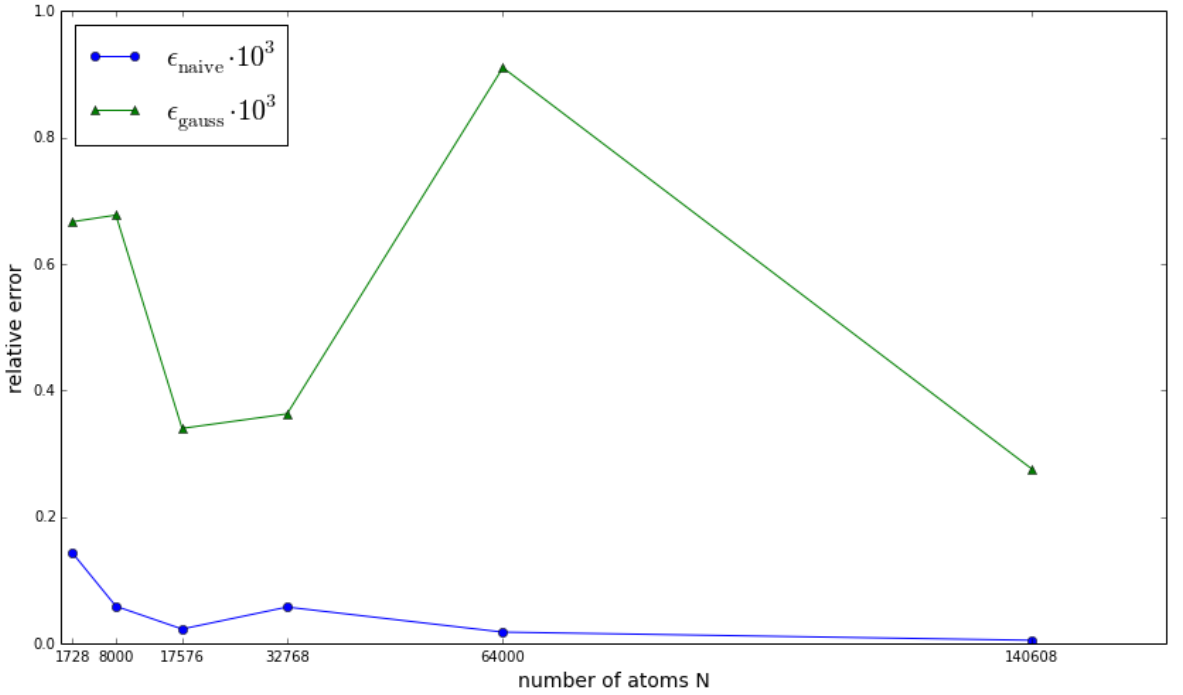

Figure 4.8: Relative errors in the last time steps when using the naive charge spreading and the Gaussian filtering in the SPME method for simulating a KCL crystal.

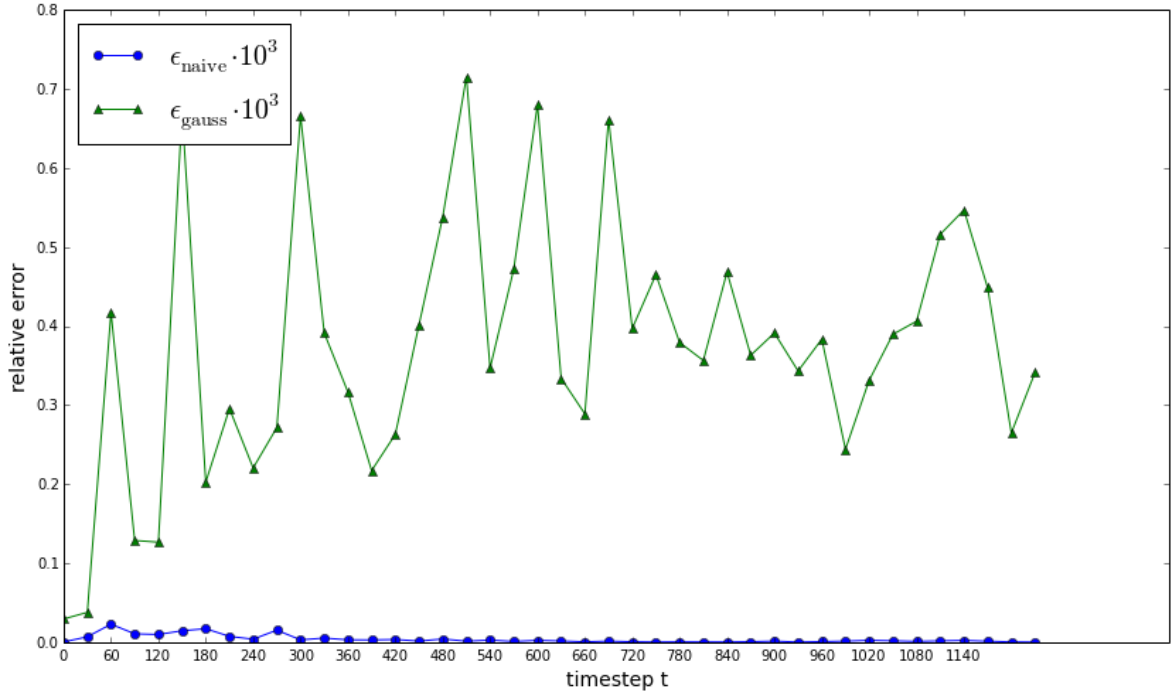

Figure 4.9: Relative errors over a whole simulation with 17576 particles. Errors were computed every 30th timestep.

for a smaller grid spacing h the varying still persists. In contrast it can be seen, that for this implementation, the B-spline approach provides the same accuracy for both grid spacings, see figure 4.11. So even if one would get good results with the filter approach for very fine grid spacings h, the B-spline approach would be a more reasonable choice to use. When using a finer grid, the computational cost is increasing and the B-spline method, needing a coarser grid for accurate results, would then be more efficient. The bad results of the filtering approach regarding accuracy are all the worse considering that a higher order is used than in the B-spline approaches. If one would use a B-spline order of n, the filter size would be $n+1$ like shown before, so much more grid points are taken into account for the charge spreading. For the here used example, this are $4^3 = 64$ points for the B-spline versions and $5^3 = 125$ points for the filtering one. The B-spline approaches need less than a half of the points in the computation. For real world simulations, a method with such bad results for accuracy is not applicable.

In the following chapter it is shown how to integrate OpenCL code into the `pydlpoly` software.

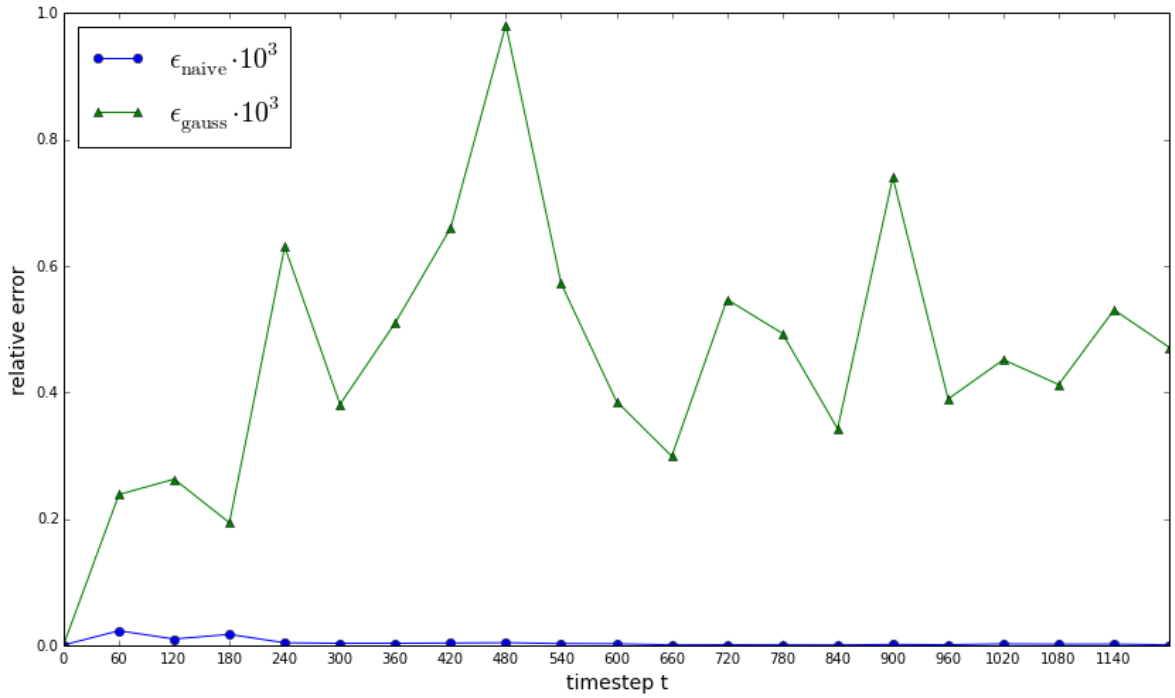

Figure 4.10: Relative errors over a whole simulation with 17576 particles and errors computed every 60th timestep. The grid spacing was reduced to 1.8.

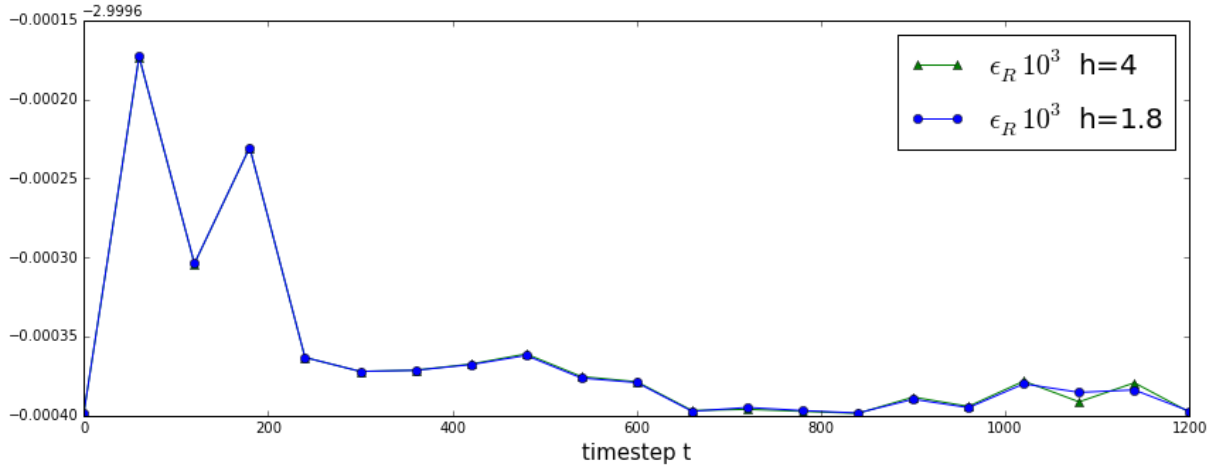

Figure 4.11: Relative error of the B-spline method for h=1.8 and h=4.0.

Chapter 5

OpenCL Code Integration into pydlpoly

The original reason for examining the SPME method and the most suitable approach for the charge assignment for an implemantation on a GPU, were long runtimes when computing Coulomb interactions with `pydlpoly`. `pydlpoly` is a parallel molecular dynamics simulation package, which is a wrapped version of DL_POLY [7], allowing the use of the Python programming languange. The main code is written in Fortran90 and uses MPI for parallelization. There is no GPU code in `pydlpoly` yet and this chapter serves as a proof of concept, how OpenCL could be integrated into the main code. Here, OpenCL is used with C++, so the first step will be to show, how C++ can be called from Fortran.

5.1 Fortran and C/C++ Interoperability

To call a C or C++ function from Fortran, several differences between the languages have to be considered. The first one is the memory layout of multidimensional arrays, which is of colloum-major order in Fortran and of row-major order in C/C++. Also, indicies of arrays in Fortran start with 1, not 0 like in C/C++, if not specified otherwise.

```
c   Fortran
    integer a(4, 2)
    a(3, 2) = 42

//  C/C++
    int a[2][4];
    a[1][2] = 42;
```

When writing a function in C/C++, which shall be called from Fortran, it is important to use lower case letters for the names with an underscore at the end. The reason for this is the fact, that the entry point names of some Fortran compilers have an underscore appended and the case is not preserved but represented in lower case. A subroutine in Fortran is a function returning `void` in C/C++, where the arguments have to be passed as pointers, since all arguments in Fortran are passed by reference.

```
c    Fortran
     integer a, b
     a = 3
     call MYSUBROUTINE(a, b)
c    after calling 'mysubroutine', b has the value 12
```

```
c    see C++ function below

//   C/C++
     void mysubroutine_(int &a, int &b) {
         b = a * 4; // b = 12
     }
```

There are other things one might encounter when using Fortran together with C/C++, which are not shown here. To create an executable out of the different parts, one first compiles the separate files without linking and then links them together using the created object files.

```
gfortran -c mymoduleF.f
g++ -c mymoduleC.cpp
gfortran -o mymodule mymoduleF.o mymoduleC.o -lc -lstdc++
```

Here the `-lc` and `-lstdc++` options are only necessary, if there is use of the C/C++ standard libraries.

5.2 Computing pydlpoly's charge summation with an OpenCL kernel

The integration of the charge assignment kernel is now done by calling C++ functions from the Fortran code, as described in the previous section. Therefor, a new file for the C++ code is created next to the Fortran modules, here called `spme.cpp`. The same holds for the OpenCL kernel file, `spme.cl`. The first function of the C++ code creates the OpenCL context and initilizes several OpenCL buffers and the kernel, which are in global namespace, so they can be used in other C++ functions. This function can then execute the kernel doing the actual computation.. It therefore gets all arrays like the charges of the particles, the precomputed splines and the pointer to the grid, which gets the charges assigned to. The initialization function is called from the setup module, `setup_module.f`, with the number of atoms, the number of mesh cells for each direction and the maximum spline order as arguments.

```
// setup_module.f
    subroutine parset(idnode, mxnode, buffer)
    ...
    call setup_opencl(mxspme, kmaxd, kmaxe, kmaxf, mxspl)
    return
    end subroutine parset

// spme.cpp
    void
    setup_opencl_(int &mxspme, int &kmaxd, int &kmaxe, int &kmaxf, int &mxspl)
    {
        // create context, buffers etc.
        ...
    }
```

In the SPME module, `spme_module.f`, the loop, where the 3D charge array is constructed is then replaced by the call to the second C++ function:

```
// spme_module.f
    subroutine ewald_spme(...)
```

```
    ...
    do ipass = 1, npass
    call compute_charge_array
     x    (natms, nospl, kmax1, kmax2, kmax3,
     x    ll, kk, jj, qqc, chge, bspx, bspy, bspz,
     x    kmaxd, kmaxf, kmaxe, tzz, tyy, txx)
    enddo
    ...
    end subroutine ewald_spme

// spme.cpp
    void
    compute_charge_array_(
        int &natms, int &nospl, int &kmax1, int &kmax2, int &kmax3,
        int &ll, int &kk, int &jj, double *qqc, double *chge,
        double *bspx, double *bspy, double *bspz,
        int &kmaxd, int &kmaxf, int &kmaxe,
        double *tzz, double *tyy, double *txx
    )
    {
        // write buffers
        ...
        NDRange global(natms);

        queue.enqueueNDRangeKernel(kernel, NullRange, global, NullRange);
        queue.enqueueReadBuffer(qqc_buf, CL_TRUE, 0,
                                kmaxd * kmaxe * kmaxf * sizeof(double), qqc);
    }
```

The code for the kernel is shown in listing 5.1, which basically uses the same algorithm as the Fortran code.

```
__kernel void
compute_charge_array(int natms, int nospl, int kmax1, int kmax2, int kmax3,
    __global double *qqc, __global double *chge,
    __global double *bspx, __global double *bspy, __global double *bspz,
    int kmaxd, int kmaxf, int kmaxe,
    __global double *tzz, __global double *tyy, __global double *txx)
{
    uint i = get_global_id(0);
    if(i < natms)
    {
        int ll, kk, jj;
        for(int l = 0; l < nospl; ++l)
        {
            ll = (int)(txx[i]) - l + 2;
            if(ll > kmax3) {
                ll = 0;
            }
            if(ll < 0) {
                ll += kmax3;
            }
            for(int k = 0; k < nospl; ++k)
            {
                kk = (int)(tyy[i]) - k + 2;
                if(kk > kmax2) {
                    kk = 0;
                }
                if(kk < 0) {
```

```
                    kk += kmax2;
                }
                for(int j = 0; j < nospl; ++j)
                {
                    jj = (int)(txx[i]) - j + 2;
                    if(jj > kmax1) {
                        jj = 0;
                    }
                    if(jj < 0) {
                        jj += kmax1;
                    }
                    int index = jj + kmaxd * (kk + kmaxe * ll);
                    double tmp = qqc[index] +
                        chge[i] * bspx[i + natms * j] *
                        bspy[i + natms * k] * bspz[i + natms * l];
                     _atomic_add(&qqc[index], tmp);
                }
            }
        }
    }
}
```

Listing 5.1: Kernel computing the charge array Q using the algorithm of DL_POLY

Note that the multidimensional arrays from Fortran are treated as one dimensional arrays. The last thing to consider is the use of file paths in the C++ file. To read the kernel file, one might think the path to the file is simply the filename itself, since the two files are in the same directory. But the path is relative to the current working directory, which is set in the Python code. Assume the following directory structure:

```
pydlpoly
├── dl_poly
│   ├── ...
│   ├── spme_module.f
│   ├── spme.cpp
│   └── spme.cl
├── so
└── test
    ├── run.py
    └── MOF_1
```

Here, the `run.py` file is used to start the simulation by using the `pydlpoly` Python class of the `pydlpoly` package, and it then creates a working directory for the current run, here `MOF_1` (in the next run, it would be `MOF_2` and so on). With this directory as the current working directory, using `"spme.cl"` in `spme.cpp` would search for the file in afore-said directory, not in the `dl_poly` directory. In this case the correct filename would be `"../../spme.cl"`. After extending the Makefile to compile and link the C++ file together with the Fortran files, and adding the option to link against the OpenCL libraries, the simulation can be started as usual and the construction of the 3D charge array is done by the OpenCL kernel. For compiling the C++ file, the `-fPIC` option must not be forgotten, since a shared object file will be created.

Chapter 6

Conclusion

In this thesis, different approaches for parallel charge assignment were presented. Charge assignment is a crucial step of the smooth particle mesh Ewald (SPME) method, a method to handle long-range interactions in biomolecular simulations. SPME reduces the usual computational costs of $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log N)$ by employing fast Fourier transforms. The basic steps of the SPME method then consists of the assignment of the point charges, modeled by Gaussians, to a grid, solving Poisson's equation by using FFT's and obtaining the reciprocal energy and potential terms. The reciprocal force terms can then be computed directly, due to the use of differentiable B-splines. An implementation of this method was presented using OpenCL, which was used to compare different approaches for the charge assignment. The standard approach is to use B-splines for the interpolation and to use a per particle charge spreading approach, where the charge is assigned to n^3 points of the grid, n being the used B-spline order. Implementing this variant in parallel leads to conflicts when different work items attempt to contribute charge to the same grid point. Therefore memory save atomic operations have to be used in this approach. Since atomic additions only exist for integer values, the atomic addition was emulated for floating point values using a compare-exchange loop. To avoid the use of atomic operations, a per grid point charge gathering approach was implemented. Therefore particles were first assigned to cells of the mesh, with the requirement, that every cell is permitted to hold only one particle. For additional particles, the standard charge spreading method was used. The number of these particles is usually very small, because in standard MD setups, there is a significantly higher number of mesh points than particles, so the system is very sparse. To improve this approach, values were shared over local memory between work items of the same work groups. Yet, the naive charge spreading approach was still more efficient for the used Nvidia GeForce GTX 580. The third approach used an interpolation and filtering scheme to spread the point charges over the grid. The use of these simple filter operations led to shorter runtimes than with the naive approach, but this brought the drawback of poor accuracy. So the Gaussian filter approach is not applicable when seriously studying the structures and dynamics of molecules. Regarding the tests and hardware used in this thesis, the use of the standard charge spreading is prefered towards using the other approaches. A further part of this thesis was the integration of OpenCL code into the molecular simulation package `pydlpoly`. An easy integration to efficiently test the different kernels for the charge assignment was not possible. This was due to the fact that there is no OpenCL code in `pydlpoly` yet. This is why in a further step it was shown how the OpenCL code can be integrated using the interoperability of the Fortran code of DL_POLY with C/C++ code. The OpenCL kernels then can be called from C/C++ functions. In the future, more parts of the code have to be ported to OpenCL, in order to be able to use a full simulation on the GPU.

Bibliography

[1] Opencl fast fourier transforms (fft's) @ONLINE.

[2] Josh Barnes and Piet Hut. A hierarchical o (n log n) force-calculation algorithm. 1986.

[3] A. Brandt. Multilevel computations of integral transforms and particle interactions with oscillatory kernels. *Computer Physics Communications*, 65:24–38, April 1991.

[4] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Math. comput*, 19(90):297–301, 1965.

[5] Ulrich Essmann, Lalith Perera, Max L Berkowitz, Tom Darden, Hsing Lee, and Lee G Pedersen. A smooth particle mesh ewald method. *The Journal of Chemical Physics*, 103(19):8577–8593, 1995.

[6] Michael Griebel, Stephan Knapek, and Gerhard Zumbusch. *Numerical Simulation in Molecular Dynamics: Numerics, Algorithms, Parallelization, Applications*. Springer Publishing Company, Incorporated, 1st edition, 2007.

[7] K. Trachenko & M.T. Dove I.T. Todorov, W. Smith. *Journal of Materials Chemistry*, 2006.

[8] I. J. Schoenberg. *Cardinal Spline Interpolation*. CBMS-NSF Regional Conference Series in Applied Mathematics. Society for Industrial and Applied Mathematics, 1973.

[9] W Smith and TR Forester. Parallel macromolecular simulations and the replicated data strategy: I. the computation of atomic forces. *Computer physics communications*, 79(1):52–62, 1994.