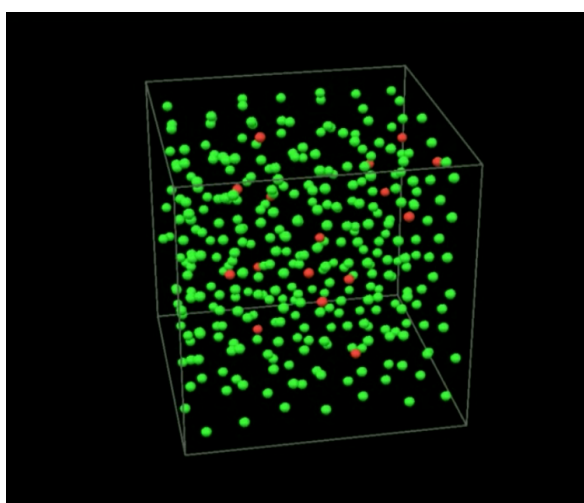




Universiteit Utrecht



Extending Time Scales for Molecular Simulations

MSc. thesis - Mathematical Sciences

J.M. Wolterink, BSc.

Supervisors: Prof. Dr. R.H. Bisseling, Utrecht University
Dr. S. Redon, INRIA Rhône-Alpes

Second reader: Dr. P.A. Zegeling, Utrecht University

Utrecht University, the Netherlands

December 2012

Abstract

Molecular dynamics simulations and protein structure prediction are essential tools in the quest for better understanding of the behavior of complex proteins. Based on these insights, new drugs can be designed to help fight diseases. Here we present a GPU implementation and analysis of adaptively restrained particle simulations (ARPS) [1], which is up to fourteen times faster than conventional full dynamics molecular dynamics implementations on the GPU. Furthermore, we present a protein deformation algorithm which allows interactive manipulation of macromolecular graphs in accordance with popular force fields such as CHARMM, based on recent advances in graphics free form deformation [2, 3]. These two algorithms allow for longer and larger simulations, thus greatly enhancing the power of molecular simulations.

Contents

Contents	ii
Preface	v
1 Introduction	1
1.1 Protein modeling	3
1.2 Contributions	4
2 Molecular Dynamics	7
2.1 Force fields	8
2.2 Integrating the Equations of Motion	10
2.3 Computational Improvements	12
2.4 Molecular Dynamics on Graphics Processing Units	13
3 Adaptively Restrained Particle Simulations on Graphics Hardware	17
3.1 Introduction	17
ARPS: Adaptively Restrained Particle Dynamics	18
ARPS on CPU	19
3.2 ARPS on a GPU	20
Updating forces	20
Theoretical speedup	22
3.3 Implementation	24
Mapping of algorithm to OpenCL kernels	25
Prefix ordering	25
Linked cell lists	26
On the use of cell lists	27
Langevin integration	29
3.4 Results	30
Benchmark 1: Particle NVE argon Liquid	31
Benchmark 2: Collision cascade	35
Benchmark 3: Bonded Interactions	37
Benchmark 4: Obtaining statistics in the NVT ensemble	40
3.5 Conclusion and discussion	41
4 Interactive Manipulation of Large Macromolecules	43
4.1 Introduction	43
4.2 Related work	44
Skinning	44
4.3 Methods	47
Quadratic program	48
Laplacian matrix	48
Mass matrix	49
Shape preservation	49
4.4 Implementation	50
4.5 Results	50
Problem 1: 1-dimensional chain	50

Problem 2: 2-dimensional ring	51
Problem 3: Graphene	52
Problem 4: Molecule manipulation	53
Problem 5: Simulation manipulation	53
4.6 Conclusion and discussion	54
5 Conclusion and discussion	59
Bibliography	63

Preface

The work presented in this thesis was conducted at INRIA Rhône-Alpes, the French national institute for informatics and mathematics in Grenoble. For six months, from February 2012 till August 2012, I had the pleasure of being an intern in the NANO-D team, under supervision of Stephane Redon. I could not have found a better place to do an internship. Stephane and the other members of the NANO-D team create a perfect working environment. Through collaboration and excellent research, but also through hospitality, ever interesting discussions and NANO-ski trips. Stephane's enthusiasm and inspiring views on science were very motivating. Thanks also go to all the other people at or outside INRIA who made my stay in Grenoble such a wonderful experience and with whom I had the pleasure of hiking up a mountain, downing a beer or gliding/tumbling down the ski slopes of Chamrousse.

I would like to express my gratitude to Rob Bisseling, my thesis supervisor in Utrecht. Rob Bisseling guided me through the final stages of my thesis. Without his invaluable advice, the writing process would not have progressed as well as it did. Furthermore, I would like to thank Paul Zegeling for agreeing to be the second reader to my thesis.

Finally I would like to thank my parents, for their patience and support while I went through six and a bit years of university, my two big sisters and my wonderful girlfriend Saskia, for inspiring and supporting me daily.

Jelmer Wolterink

Introduction

Without a doubt, developments in computer science have had radical effects on almost all aspects of everyday life. Not in the least has it changed the way science is conducted. Whereas the classical physicist or chemist used to run experiments in a controlled laboratory environment, nowadays experiments move more and more to the digital realm. Modeling and simulation on computers can help avoid running unnecessary and often complicated laboratory experiments, or experiments which are impossible to run in a laboratory setting. For instance, the behavior of a system of particles might be modeled as an N -body problem on a computer.

The N -body problem has its origins in the modeling of solar systems, but turned out to be a proper way to model systems of particles. However, with $N \geq 3$ this problem is analytically insoluble, so the equations of motions have to be solved numerically [4]. Not soon after the development of the first computers, the first steps were taken towards numerical simulation of systems of particles [5, 6, 7], dubbed molecular dynamics (MD) in the late 1950s. MD has evolved to become a very important computational technique in a wide range of fields, like virology [8], studies on DNA [9] and drug design [10].

Figure 1.1 shows the steps in a typical MD simulation. The goal is to simulate the dynamics of a system of particles which interact through a force field, a concept which is more elaborately described in Chapter 2. Usually, computing the forces on all atoms accounts for more than 90% of the time one step takes. The time steps are very small compared to the timescales of the observed mechanisms and more often than not millions of iterations need to be run to obtain feasible results. Two ongoing challenges are to speed up the time it takes to simulate one step and to increase the size of simulated systems. These challenges are taken on by improvements in hardware as well as better and faster algorithms.

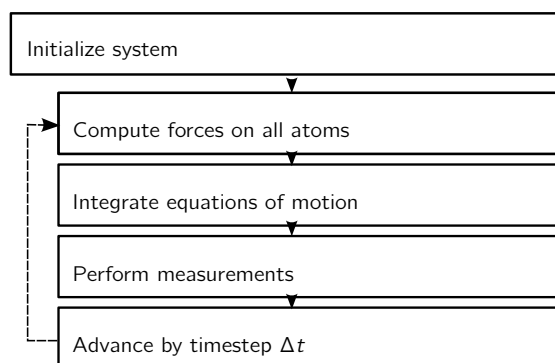


Figure 1.1: Steps in a molecular dynamics simulation.

Increases in computing power have always found their way to MD simulation of particle systems. As the speed of computers increased over the past 60 years, so did the possibilities of MD. In the quest for ever longer simulation intervals for bigger systems and more complex force fields, parallel machines

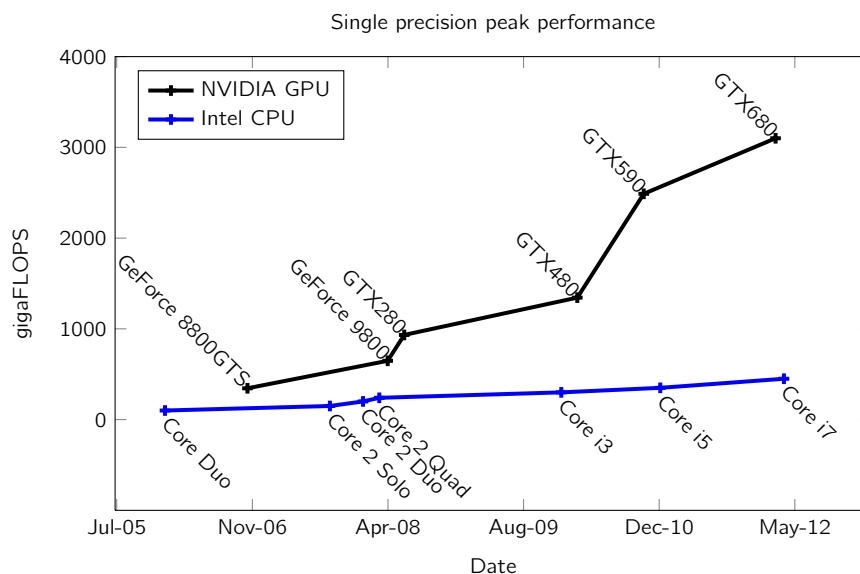


Figure 1.2: Single precision peak performance of flagship Intel CPU chips and NVIDIA graphics GPU chips. Data obtained from [21, 22, 23]

consisting of hundreds of cores were employed for MD simulations. Also, machines were built specifically to speed up MD simulations. Notable examples of these are the ANTON machine (called after Dutch microscopy pioneer Antonie van Leeuwenhoek) which contains highly specialized hardware for both bonded and non-bonded forces, and the MDGRAPE-3 machine which is specifically built for protein structure prediction [11, 12].

In scientific computations, the speed of a machine is usually expressed in FLOPS (or flop/s). A flop is a **f**loating-point **o**peration such as multiplication or addition. Enormous efforts have been put into building ever larger supercomputers which yield high FLOPS. At the moment of writing (August 2012), the fastest supercomputer in the world, Sequoia BlueGene/Q, has a benchmarked peak performance of approximately 20 petaFLOPS, $20 \cdot 10^{15}$ operations per second [13]. Interestingly, recent massively distributed approaches to MD are performing at levels comparable to those of the best supercomputers in the world, with the Folding@home project reporting 9 petaFLOPS/s of user-contributed computing power to speed up protein folding research [14], only a factor two smaller than the BlueGene/Q machine.

These are all remarkable steps in the ongoing search for fast molecular dynamics simulations. The big drawback however is the excessive cost of purpose-built machines or supercomputers and the fact that these are usually not present in a common scientific working environment. As a solution to these problem, the past few years have seen the emergence of molecular simulations on graphics processing units (GPUs), with a tremendous increase in the number of MD on GPU related papers. GPUs are ubiquitous in scientific environments, as they belong to the standard components of most workstations. Initially GPUs were used to increase the quality of graphics in computer games, but recently it has been recognized that GPUs are very useful for some scientific general purpose computations, including MD. Various paradigms have been developed for MD simulations on GPU, up to a point where GPUs reach orders of magnitude speedup with respect to state-of-the-art CPU implementations [15, 16, 17, 18, 19].

Supercomputers still outperform individual GPUs. The fastest graphics cards available at the moment report (single precision) peak performances close to the 4 teraFLOPS mark, around 5,000 times smaller than BlueGene/Q [20]. However, whereas BlueGene uses more than 1.5 million cores and consumes the same amount of energy as approximately 2000 households, at enormous building and operating costs, any individual can put his hands on the newest NVIDIA or AMD graphics card for less than 500 €. Furthermore, a cluster of some thousand GPUs might perform comparable to BlueGene. At the same time, CPUs, which used to be the sensible choice for desktop scientific modeling and simulations, have been lagging behind in performance with respect to graphics processing units. In Figure 1.2 we see how the number of FLOPS of graphics cards has increased rapidly over the past few years, while peak performance for the CPU increases at a much smaller rate.

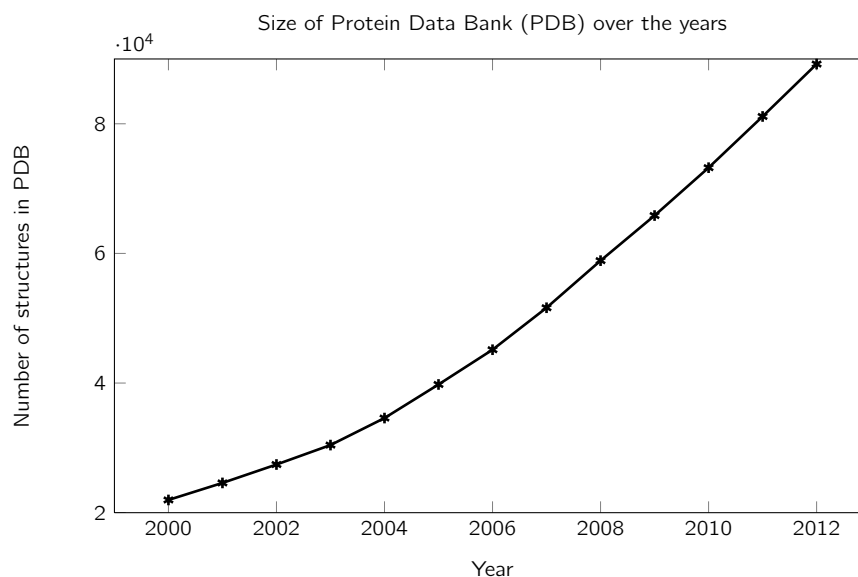


Figure 1.3: Number of structures in the Protein Data Bank (PDB) at January 1 of each year. [24]

1.1 Protein modeling

Over the past few decades it has become more and more clear that a good understanding of the behavior of proteins when interacting with other proteins or solvents might be very helpful e.g. in drug discovery. Driven by non-covalent interactions such as hydrogen bonding, ionic interactions, Van der Waals forces and hydrophobic packing, a protein polymer (or polypeptide) assumes a particular structure. Finding these three-dimensional structures is the topic of the scientific field of structural biology. Using X-ray crystallography or NMR spectroscopy, scientists can determine the structure of proteins, but it remains unclear how exactly a protein, a macromolecular complex of amino acids, folds itself into such a structure. New structures are discovered daily, with the number of known molecular structures in the Protein Data Bank (PDB), an international repository of structural files based at Rutgers, The State University of New Jersey, and the University of California-San Diego (UCSD), growing super-linearly (Figure 1.3) [24].

Though X-ray crystallography or NMR spectroscopy can be employed to find the three-dimensional structures of folded polypeptides, it is a cumbersome process. Therefore, the PDB contains only a small fraction of all proteins. It is much easier to determine the protein sequence, i.e. the order in which amino acid residues are linked together by peptides in a polypeptide. This is where *protein structure prediction* comes in, the computational prediction of protein structure from a protein sequence. Numerous methods have been developed, ranging from molecular dynamics methods using just the protein sequence to heuristic methods employing structural features of related proteins.

Protein structure prediction is a very hard and computationally challenging problem. An obvious tool for studying the behavior of proteins is molecular dynamics. Molecular dynamics simulations on small proteins in explicit solvent in the millisecond range have recently become possible [25, 26]. It should however be noted that wall-clock simulation time increases quadratically with the size of the protein and linearly with the simulated time, which in turn tends to increase as the system size increases; it is often irrelevant to run a short simulation of a very big system. In [27], it is noted that the increase of simulated time and system size reflects Moore's law, which states that the number of transistors on a chip doubles roughly every two years [28]. If Moore's law is to hold until at least 2050, the authors predict all-atom dynamics simulations of an entire bacterial cell (Figure 1.4). Molecular dynamics simulations of proteins are important not only because of their eventual configuration, but also because of the insights they might give in the processes that constitute folding or docking.

Nevertheless, right now it is not possible to run full dynamics all-atom simulations of large proteins with realistic timescales. Nor are there any other experimental techniques which give insight in all stages of protein folding dynamics [29]. Over the past few years, two distinct subfields of protein structure prediction have emerged: protein folding and protein docking. The first one is concerned with the way polypeptides assume their three-dimensional structure as a protein, the latter one deals with the way

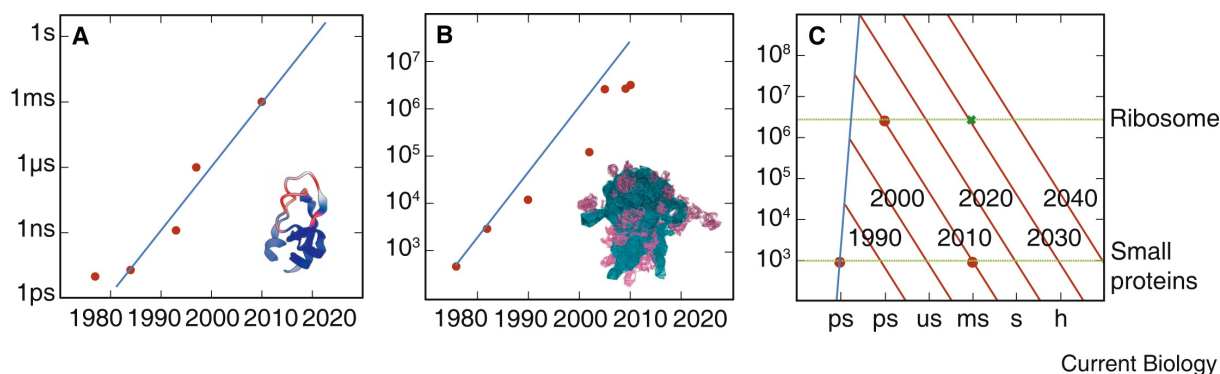


Figure 1.4: Plots showing growth in time scale (a) and system size (b) accessible to all-atom molecular dynamics simulations of proteins. The dots show noticeable milestones in MD simulations, e.g. the millisecond simulation by [26] on the top right in (a). Subfigure (c) shows boundaries of MD simulations changing in time. In an extrapolation towards 2040, we see an expectation of how researchers will be able to simulate larger systems at longer time scales. [27]¹

two or more proteins interact and connect and is very useful to study e.g. drug binding [10]. Advances within both fields are encouraged by the bi-annual CASP competition for protein folding and the ongoing CAPRI experiment for protein-protein docking [30]. Often, these problems are treated as optimization problems, opening possibilities for a wide range of optimization and machine learning algorithms.

Current methods thus make it possible to closely follow the behavior of small proteins using molecular dynamics with short time scales or obtain good estimates of the ultimate configuration of proteins. However, even the highly sophisticated ANTON special-purpose machine takes seven months to simulate one millisecond of the bovine pancreatic trypsin inhibitor (BPTI) protein in water [25].

As a result of the rapid increase in the knowledge of macromolecular structures, the scientific community has seen the emergence of numerous software tools, either to view macromolecular structures or to do modeling and simulation. Notable tools are YASARA, which is distributed with different levels of functionality and PYMOL, a widely popular molecular visualization system [31, 32]. Another interesting tool is ProteinShop, formerly ProtoShop, which allows the user to interactively manipulate protein fragments 'to achieve desired folds by adjusting the dihedral angles of selected coil regions using an Inverse Kinematics method' [33]. ProteinShop's utility is twofold: it allows the user to prepare a protein for structure prediction and it allows the user to intervene in an optimization process to steer the algorithm in the right direction.

To accurately mimic the atomic behavior between two user-imposed positions using MD, e.g. when the user is moving an α -helix away from the rest of the protein, would take ages. Thus, accurate all-atom simulations are out of the question in interactive macromolecular manipulation. These methods are too slow to employ in protein editing, e.g. situations where we want to apply affine transformations to (parts of) a macromolecule, add parts or remove parts. Hence, other methods like the inverse kinematics approach in [33] are necessary. In Chapter 4, we present such a method based on dual quaternion blending according to biharmonic bounded weights.

1.2 Contributions

Molecular dynamics simulations are a tremendous asset to the arsenal of experimental physicists, chemists and biologist worldwide. Over the past few decades numerous methods have been developed to mimic the behavior of complex particle systems on the computer. Computer hardware has gotten more and more advanced and progress in MD simulations reflects the increase in available computing power [27]. Molecular dynamics simulations are essential in the quest for knowledge of the processes that constitute protein docking and folding.

Here, we present two contributions to the fields of Molecular Dynamics (MD) and protein modeling and simulation. The goal of this work is to reduce the wall-clock time it takes to simulate or model

¹Reprinted from Current Biology, Vol 21, Michele Vendruscolo and Christopher M. Dobson, Protein Dynamics: Moores Law in Molecular Biology, R68–R70, 2011, with permission from Elsevier.

systems of particles and hence extend the time scales on which molecular simulations are run. In Chapter 3 we present a method which employs the graphics processing unit (GPU) to run adaptively restrained particle simulations (ARPS). The speedup caused by ARPS, combined with the computational efficiency of the GPU allow us to simulate large particle systems up to two orders of magnitude faster than with classical MD simulations on the central processing unit (CPU). The low cost and ubiquity of GPUs makes ARPS on GPU an easily accessible method to enhance desktop MD simulations. Also, ARPS on GPU might be used to speed up simulations on grids of GPUs.

In Chapter 4 a novel approach to protein modeling is introduced. The goal is similar to that of ProteinShop, but the approach is very different. Commonly used forcefields such as AMBER and CHARMM can be applied to an interactive protein manipulation algorithm which transforms particle positions in a fast yet physically plausible way, based on recent advances in free form deformation [2, 3]. This method can have great implications for the field of protein structure prediction, where it might be used to prepare experiments or allow the user to intervene in running experiments, thus cutting simulation time, increasing quality of predicted protein structures and extending the time scale of the simulation.

The layout of this thesis is as follows. In Chapter 2 we give a more thorough introduction on molecular dynamics and force fields, in Chapter 3 we introduce ARPS on GPU and in Chapter 4 we present our new interactive protein editing algorithm. Both new approaches and their contribution to molecular dynamics and protein structure prediction are more thoroughly discussed in Chapter 5.

Molecular Dynamics

Contents

2.1 Force fields	8
2.2 Integrating the Equations of Motion	10
2.3 Computational Improvements	12
2.4 Molecular Dynamics on Graphics Processing Units	13

Molecular dynamic simulations solve the equations of motion for an N -body system in a numerical way by repetitively applying the steps in Figure 1.1. The simplest form of molecular dynamics, a system of structureless particles, involves little more than Newton's second law [4]. Systems can get more complicated if more constraints are added, like internal degrees of freedom in molecules.

There are multiple ways to define the equations of motion in an N -body system. One way is to use Lagrangian dynamics to express second-order differential constraints on an n -dimensional coordinate system, another way is to use Hamiltonian mechanics to express first-order constraints on a $2n$ -dimensional phase space (n is the number of degrees of freedom in the system). Both approaches assume that particle behavior is dictated by Newton's second law,

$$m\ddot{\mathbf{r}}_i = m\mathbf{a} = \mathbf{f}_i = \sum_{j=1, j \neq i}^N \mathbf{f}_{ji}. \quad (2.1)$$

In this work, we use Hamiltonian mechanics. The value of the Hamiltonian function is the total energy in a system, the sum of kinetic and potential energy. The Hamiltonian function used throughout this work is given as

$$H(\mathbf{q}, \mathbf{p}) = \underbrace{\frac{1}{2}\mathbf{p}^T \mathbf{M}^{-1}\mathbf{p}}_{\text{Kinetic energy}} + \underbrace{V(\mathbf{q})}_{\text{Potential energy}}, \quad (2.2)$$

where $\mathbf{q}, \mathbf{p} \in \mathbb{R}^{3N}$ are coordinate and momentum vectors respectively, and $\mathbf{M} \in \mathbb{R}^{3N \times 3N}$ is a diagonal block mass matrix. The kinetic energy of an individual atom i is given as $k_i = \frac{\mathbf{p}_i^T \mathbf{p}_i}{2m_i}$. Hence we can clearly distinguish between the kinetic energy term $k(p) = \frac{1}{2}\mathbf{p}^T \mathbf{M}^{-1}\mathbf{p}$ and the potential energy term $V(\mathbf{q})$ in this Hamiltonian function [34].

By repeatedly updating momenta and positions of all particles like in Figure 1.1, many features can be observed and statistics can be obtained. A system can be initialized in many ways, e.g. particle momenta might be initialized according to the system temperature T using a Maxwell-Boltzmann-distribution. Forces on atoms are computed as in Eq. (2.1). Then momenta and positions are updated according to some integration scheme and time is increased by Δt .

2.1 Force fields

The Hamiltonian in Eq. (2.2) contains a kinetic energy and a potential energy component. The kinetic energy component depends on momenta, whereas the potential energy component depends on particle coordinates and the way particles interact. The behavior and interactions of atoms within a macromolecule are complicated and susceptible to a lot of external factors. Therefore, to be able to explore this behavior numerically we need a model. A force field is the combination of a functional form and parameter sets used to describe the potential energy of a system of particles. Over the past few decades, force fields such as CHARMM and AMBER have been developed [35, 36]. In the case of the CHARMM force fields, software packages with the same name were developed for MD simulations and analysis. Generally, the atomic forces that govern molecular movements are believed to be a combination of short-range, long-range, bonded and non-bonded forces. For example, the potential energy in a molecule might be modeled as:

$$V(\mathbf{q}) = \sum_{bonds} K_r (r - r_{eq})^2 + \sum_{angles} K_\theta (\theta - \theta_{eq})^2 + \sum_{dihedrals} \frac{K_\phi}{2} [1 + \cos(n\phi - \gamma)] + \sum_{i < j} 4\epsilon_{ij} \left[\left(\frac{\sigma_{ij}}{r} \right)^{12} - \left(\frac{\sigma_{ij}}{r} \right)^6 \right] + \sum_{i < j} \frac{c_i c_j}{r}, \quad (2.3)$$

where the first two terms correspond to simple springs modeling chemical bonds and atomic angles, the third term is a sinusoidal function which models dihedral functions and the fourth term models non-bonded forces using a Lennard-Jones potential for short-range forces and a long-range term modeled on Coulomb's law, with c_i, c_j charges on particles i and j respectively [10]. Interparticle distances are given by r , bond angles by θ, ϕ and γ .

Though most force fields are based on such a model, individual force fields differ in the way parameters are set. Also, some force fields are more extended to allow more elaborate simulations. Force field parameter sets are continuously updated with new empirical results. In Eq. (2.3), there are already 9 undefined parameters: $K_r, r_{eq}, K_\theta, \theta_{eq}, K_\phi, n, \epsilon_{ij}, q_i, q_j$. There are many types of atoms and many possible interacting types of atoms. Typically, force fields are available for molecules with H, C, N, O, S and P atoms. Then to find e.g. the Lennard-Jones potential between atoms of two different types we need to combine parameters. To combine for example the depth of the potential well between two non-bonded atoms of type i and j , the new well depth is estimated as $\epsilon_{ij} = (\epsilon_{ii}\epsilon_{jj})^{1/2}$, according to Lorentz-Berthelot mixing rules.

In the simulations in Chapter 3 we use very simple force fields, where we assume that all particles are identical and there is either only a Lennard-Jones potential or a simple bonded potential between particles. In the methods introduced in Chapter 4 we might use more elaborate forcefields and we should use Lorentz-Berthelot mixing rules to approximate interactions between different kinds of atoms.

Lennard-Jones potential

In the Lennard-Jones potential (or LJ-potential), between pairs of atoms there is an inter-atomic interaction which consists of two parts. The first part repels atoms at close range, to oppose compression. The second part is attractive, causing atoms to bind together in solid and liquid states. There are multiple models for this kind of potential, with trade offs between accuracy and computational simplicity, the most well-known being Morse potential and Lennard-Jones potential. The latter one, first proposed by John Lennard-Jones in 1924 for liquid argon, is:

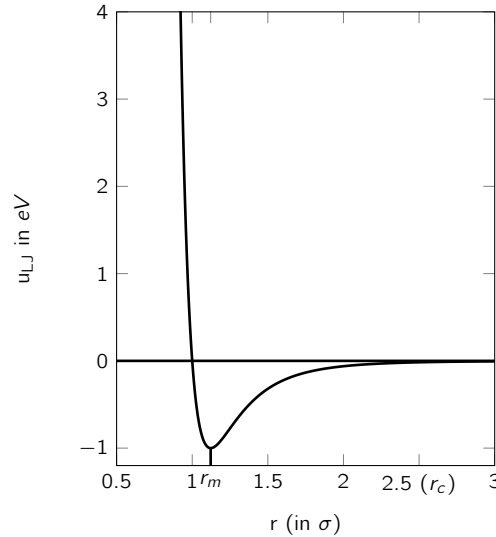
$$u_{LJ}(r) = \epsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^6 \right],$$

where ϵ is the depth of the potential well, σ is the finite distance at which the inter-particle potential is zero, and r is the Euclidean distance between particles. The Lennard-Jones potential is by far the most popular potential for non-bonded short range interactions. This potential can also be expressed as $u_{LJ} = \epsilon \left[\left(\frac{r_m}{r} \right)^{12} - 2 \left(\frac{r_m}{r} \right)^6 \right]$, where r_m is the distance at which the potential reaches its minimum. It is often more convenient to use a truncated potential with a cut-off distance r_c , beyond which the potential is set to zero. Then only particles within a certain neighborhood have to be considered to compute inter-atomic potentials. In Figure 2.1, a good value for r_c would be 2.5σ . For liquid argon, this would correspond to a cutoff distance of $2.5 \cdot 3.4\text{\AA} = 8.5\text{\AA}$ [37]. However, this will cause u_{LJ} to jump at $r = r_c$

as in Figure 2.1B. One could shift the whole potential function up by $-u_{LJ}(r_c)$ to get rid of this jump. This might however influence the obtained statistics. As an alternative, a switching function is often used, a cubic or higher order spline going from 1 to 0 over r_s to r_c . In this work, we use

$$S(r) = \frac{(r_c^2 - r^2)^2(r_c^2 + 2r^2 - 3r_s^2)}{(r_c^2 - r_s^2)^3}.$$

12-6 Lennard-Jones potential, strength versus distance (in σ)



12-6 Lennard-Jones potential with switching function

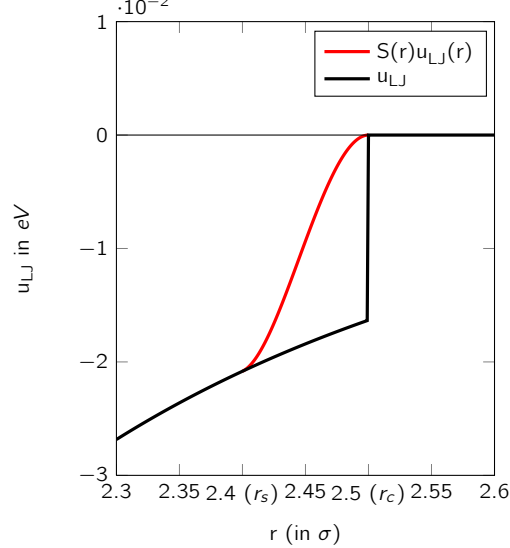


Figure 2.1: A) Strength of 12-6 Lennard-Jones potential versus distance (in σ). The distance where the potential reaches its minimum is r_m . B) Close up of potential with switching function between 2.4σ and 2.5σ . The black line has a jump at r_c , the black line with red switching function smoothly switches between r_s and r_c .

The inter-atomic potential function $u(r)$ ($u_{LJ}(r)$ in this example) corresponds to an inter-atomic force function $\mathbf{f}(\mathbf{r})$ as

$$\mathbf{f}(\mathbf{r}) = -\nabla u(\mathbf{r}) = -\frac{du}{dr}(r)\mathbf{r},$$

where \mathbf{r} is a distance vector, $r = \|\mathbf{r}\|$ and \mathbf{f} is a force vector, so the Lennard-Jones force with switch of atom j on atom i is

$$\mathbf{f}_{ij}(\mathbf{r}_{ij}) = S(r_{ij}) \left(\frac{48\epsilon}{\sigma^2} \right) \left[\left(\frac{\sigma}{r_{ij}} \right)^{14} - \frac{1}{2} \left(\frac{\sigma}{r_{ij}} \right)^8 \right] \mathbf{r}_{ij}.$$

According to Newton's third law, $\mathbf{f}_{ij} = -\mathbf{f}_{ji}$, so usually the number of forces to be computed can be cut in half by evaluating only one of these two, thus significantly reducing computing times.

2.2 Integrating the Equations of Motion

In Hamiltonian mechanics, the equations of motion are derived as

$$\begin{aligned} \dot{\mathbf{p}} &= -\frac{\partial H(\mathbf{q}, \mathbf{p})}{\partial \mathbf{q}}, \\ \dot{\mathbf{q}} &= \frac{\partial H(\mathbf{q}, \mathbf{p})}{\partial \mathbf{p}}. \end{aligned} \quad (2.4)$$

Numerous schemes exist to solve these equations of motion, e.g. Verlet schemes, implicit Euler and explicit Euler [34]. These schemes all have their positive and negative properties. Some schemes are very exact on short time intervals but show drift on longer intervals, other schemes are less exact on short intervals but more reliable on longer intervals (e.g. Verlet). A very useful property of a numerical integrator is that of symplecticity. This means that the algorithm has to be area preserving in p, q -phase space.

This is most easily explained with an example in Figure 2.2. A particle in one-dimensional space is represented by its momentum and position in two-dimensional p, q -phase space. We take two such points $\xi = [\xi^p \ \xi^q]^T$ and $\eta = [\eta^p \ \eta^q]^T$ and draw the parallelogram P containing ξ and η as:

$$P = \{t\xi + s\eta \mid 0 \leq s, t \leq 1\} \subset \mathbb{R}^2.$$

The oriented area of this parallelogram is given as

$$\begin{aligned} \text{or.area}(P) &= \omega(\xi, \eta) = \det \begin{bmatrix} \xi^p & \eta^p \\ \xi^q & \eta^q \end{bmatrix} \\ &= \xi^p \eta^q - \xi^q \eta^p \\ &= \xi^T J \eta, \end{aligned}$$

with

$$J = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}.$$

Now a linear mapping $A : \mathbb{R}^2 \rightarrow \mathbb{R}^2$ on ξ and η is called **symplectic** if $A^T J A = J$, or equivalently $\omega(A\xi, A\eta) = \omega(\xi, \eta) \forall \xi, \eta \in \mathbb{R}^2$. This is easily generalized for higher dimensions. Then in the case of integrators, we can say that its corresponding differentiable map $g : U \rightarrow \mathbb{R}^{2d}$ is symplectic if the Jacobian $g'(p, q)$ is symplectic everywhere [38].

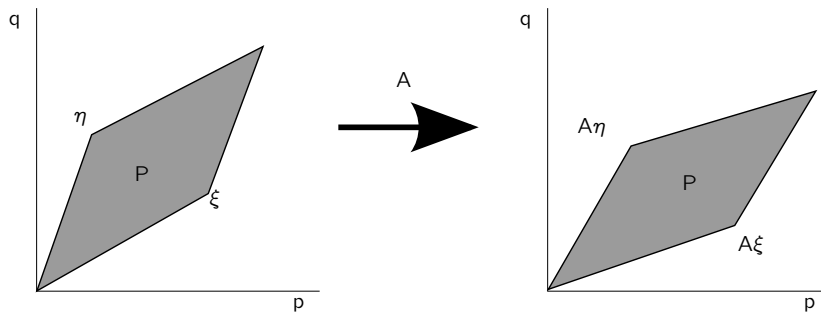


Figure 2.2: Symplectic transformation of parallelogram spanned by ξ and η . Notice how A is area preserving.

A more prosaic example runs as follows: we take all the trajectories in phase space that correspond to a particular energy E and say that these are contained in the hyper volume Ω . Then we run the algorithm on the points in Ω . With a symplectic integrator we would end up with a new volume Ω' which has exactly the same size as Ω . However, a non-symplectic integrator would map Ω to an Ω' with a different (usually larger) size. This does not correspond with energy conservation, a requirement which is often made in an MD simulation. Serious drift problems might be caused by this change of volume [34].

We will now solve the equations of motion for our particular case using the symplectic Euler method and see how we can use an explicit method to solve the equations at each time step. The symplectic Euler method applied to Eq. (2.4) is defined as

$$\begin{aligned}\mathbf{p}_{n+1} &= \mathbf{p}_n - \Delta t \frac{\partial H(\mathbf{q}_n, \mathbf{p}_{n+1})}{\partial \mathbf{q}}, \\ \mathbf{q}_{n+1} &= \mathbf{q}_n + \Delta t \frac{\partial H(\mathbf{q}_n, \mathbf{p}_{n+1})}{\partial \mathbf{p}}.\end{aligned}$$

In the case of the Hamiltonian in Eq. (2.2) we get

$$\frac{\partial H(\mathbf{q}, \mathbf{p})}{\partial \mathbf{p}} = \mathbf{M}^{-1} \mathbf{p}, \quad \frac{\partial H(\mathbf{q}, \mathbf{p})}{\partial \mathbf{q}} = \frac{\partial V(\mathbf{q})}{\partial \mathbf{q}}.$$

Hence

$$\begin{aligned}\mathbf{p}_{n+1} &= \mathbf{p}_n - \Delta t \frac{\partial V(\mathbf{q}_n)}{\partial \mathbf{q}}, \\ \mathbf{q}_{n+1} &= \mathbf{q}_n + \Delta t \mathbf{M}^{-1} \mathbf{p}_{n+1}.\end{aligned}$$

It is clear that at time step $n + 1$ we can first update \mathbf{p}_{n+1} by adding the scaled force vector $\Delta t \mathbf{f} = -\Delta t \frac{\partial V}{\partial \mathbf{q}_n}$ to \mathbf{p}_n . Then we add $\Delta t \mathbf{M}^{-1} \mathbf{p}_{n+1}$ to \mathbf{q}_n to obtain the new position. This is a very nice and cheap explicit way of integration. The fact that this method is explicit is due to the separability of the Hamiltonian: $H(\mathbf{p}, \mathbf{q}) = K(\mathbf{p}) + V(\mathbf{q})$. It was proven in [39] that this Euler method is symplectic. A proof of this is found at page 189 of [38].

Langevin dynamics

The symplectic Euler method is sufficient to simulate particles in the NVE ensemble, where number of particles N , volume V and energy E (and hence density) are kept constant throughout the simulation. If we want to simulate in the NVT ensemble, where N , V and temperature T are kept constant, we need a somewhat more elaborate integration scheme. The relation between T and \mathbf{p} is given as

$$\frac{\mathbf{p}^T \mathbf{M}^{-1} \mathbf{p}}{N} = k_B T,$$

where k_B is the Boltzmann constant. In an NVE simulation, the average momentum might shift along with the potential energy, changing the system temperature. Hence, we need a thermostat to control the temperature in the NVT ensemble. Though we wish to control the system temperature, and for this we should constrain the particle momenta, it is important that the difference between the constrained path and the original Newtonian trajectory is minimized [37]. Several methods have been introduced which rescale the momenta at each time step to satisfy the temperature constraints.

We use a Langevin thermostat, which adjusts the momentum of each particle so that the system temperature matches the set temperature, while minimizing the least-squares difference between the constrained trajectory and the original Newtonian trajectory. The equations change from Eq. (2.4) to:

$$\begin{aligned}\dot{\mathbf{p}} &= -\frac{\partial H(\mathbf{q}, \mathbf{p})}{\partial \mathbf{q}} + \gamma \mathbf{p} + \sigma \mathbf{R}, \\ \dot{\mathbf{q}} &= \frac{\partial H(\mathbf{q}, \mathbf{p})}{\partial \mathbf{p}},\end{aligned}$$

where γ is a $3N \times 3N$ matrix with nonnegative friction parameters on the diagonal and $\mathbf{R} \in \mathbb{R}^{3N}$ is a vector of N random forces drawn from a Gaussian distribution. The random force \mathbf{R}_i on particle i is

scaled according to the desired system temperature. The values along the diagonal of the $3N \times 3N$ matrix σ should satisfy the fluctuation-dissipation relation $\sigma\sigma^T = 2\gamma k_B T$. Note that the equation of motion does not change for \mathbf{q} .

We employ the Brünger-Brooks-Karplus (BBK) type scheme proposed in [40], where we first apply a half step for the Langevin part of this equation, then a full step for the Hamiltonian part and then a half step for the Langevin part [41]. This scheme is first order accurate [42]. One full integration step is given as

$$\begin{aligned}\mathbf{p}_{n+1/2} &= \mathbf{p}_n - \frac{\Delta t}{2} \left(\frac{\partial H(\mathbf{q}_n, \mathbf{p}_{n+1/2})}{\partial \mathbf{q}_n} + \gamma \frac{\partial H(\mathbf{q}_n, \mathbf{p}_{n+1/2})}{\partial \mathbf{p}_{n+1/2}} \right) + \sqrt{\Delta t/2} \sigma \mathbf{R}, \\ \mathbf{q}_{n+1} &= \mathbf{q}_n + \Delta t \frac{\partial H(\mathbf{q}_n, \mathbf{p}_{n+1/2})}{\partial \mathbf{p}_{n+1/2}}, \\ \mathbf{p}_{n+1} &= \mathbf{p}_{n+1/2} - \frac{\Delta t}{2} \left(\frac{\partial H(\mathbf{q}_{n+1}, \mathbf{p}_{n+1})}{\partial \mathbf{q}_{n+1}} + \gamma \frac{\partial H(\mathbf{q}_{n+1}, \mathbf{p}_{n+1})}{\partial \mathbf{p}_{n+1}} \right) + \sqrt{\Delta t/2} \sigma \mathbf{R},\end{aligned}\tag{2.5}$$

which becomes

$$\begin{aligned}\mathbf{p}_{n+1/2} &= \mathbf{p}_n - \frac{\Delta t}{2} \left(\frac{\partial V}{\partial \mathbf{q}_n} + \gamma \mathbf{M}^{-1} \mathbf{p}_{n+1/2} \right) + \sqrt{\Delta t/2} \sigma \mathbf{R}, \\ \mathbf{q}_{n+1} &= \mathbf{q}_n + \Delta t \mathbf{M}^{-1} \mathbf{p}_{n+1/2}, \\ \mathbf{p}_{n+1} &= \mathbf{p}_{n+1/2} - \frac{\Delta t}{2} \left(\frac{\partial V}{\partial \mathbf{q}_{n+1}} + \gamma \mathbf{M}^{-1} \mathbf{p}_{n+1} \right) + \sqrt{\Delta t/2} \sigma \mathbf{R}.\end{aligned}$$

The momentum update equations are implicit, to solve these we use a fixed-point algorithm as in [40].

2.3 Computational Improvements

Implementing MD simulations can be rather straightforward. As can be seen in Figure 1.1 the algorithm basically consists of only one initialization step and four steps which are repeated over and over again. However, using MD naively to obtain statistics on a system of a large number of particles can be very time-consuming. A lot of this work can be prevented by using some simple tricks.

Boundary Conditions

Atoms move inside a three-dimensional box and are destined to get out of this box at some point in the simulation. An obvious way to prevent atoms from going out of the box would be to make the box walls reflect the moving atoms. However, this usually means that an unacceptably large number of interactions happens between the wall and a particle. This is easily shown using an example from [4]. If we take a three-dimensional example with $N = 1000$, roughly 500 atoms are immediately adjacent to the walls. This will be reflected in the statistics obtained from the simulation, which will not represent the interior atom interactions well. As N increases, the relative number of particles close to the walls will decrease, but computation times will increase. Therefore it is more practical to use boundary conditions: the system is infinitely extended at its boundaries and a nearest image convention is used to compute the inter-atomic forces correctly. Though alternative boundaries exist, e.g. helical [43], periodic boundary conditions are very popular due to their conceptual simplicity. If a particle moves e.g. out the right side of the box, it reenters the box on the left side, as in Figure 2.3.

Cell Grids

The potential energy part of the Hamiltonian in 2.2 might take on different shapes, depending on the nature of the problem and the desired accuracy of the simulation. Both bonded and non-bonded interactions might be considered and within non-bonded interaction we might distinguish between short-range interactions, which can be modeled using the simple Lennard-Jones potential in Eq. (2.1), and long-range interactions which are often modeled with Coulomb's law. For long-range interactions, we cannot use a cutoff, hence it is necessary to compute $O(N^2)$ interactions. Methods based on Ewald summation, such as particle mesh Ewald (PME) are able to reduce this to $O(N \log(N))$ by summing short-range

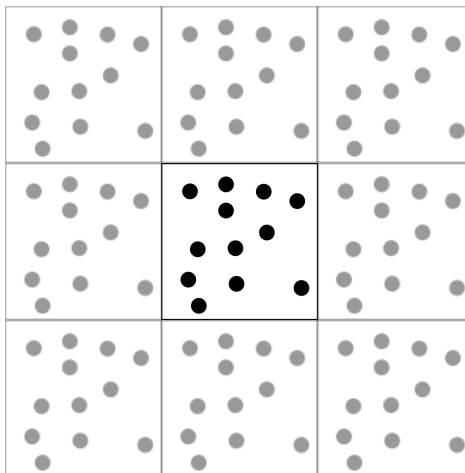


Figure 2.3: Periodic boundary conditions in two dimensions. All simulated particles are in the center box. They move out of this box to reenter it on the other side.

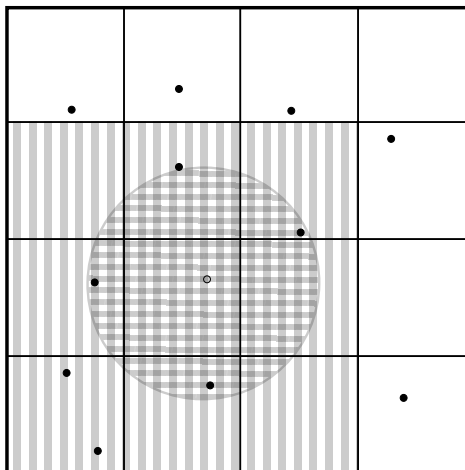


Figure 2.4: The domain of Figure 2.3 split up in 4×4 cells with width $> r_c$. The target white particle can only possibly interact with the other particles in the vertically shaded area, the neighbor cells. In fact, only particles within the horizontally shaded area are within r_c .

interactions in real space and long-range interactions in Fourier space. In the problems which we study here, a simple non-bonded short range Lennard-Jones potential suffices. In this case, we can and we will use cell grids, where the domain is split up in cells with width $r_w > r_c$ so that to compute the interactions of particle i in cell C , we only need to look at all other particles in cell C and its 26 adjacent cells (or 8 adjacent cells in 2-D). We see in Figure 2.4 how this is still an overestimation of the total number of particles within r_c of particle i , but still the complexity is reduced from $O(N^2)$ to $O(N)$.

2.4 Molecular Dynamics on Graphics Processing Units

The past few years have seen a tremendous increase in publications on molecular dynamics algorithms on graphics processing units. The fact that researchers have turned to the GPU for faster and longer simulations is not very surprising, given the ever-increasing gap between state-of-the-art CPU peak performance and GPU peak performance (Figure 1.2). Among GPU-accelerated MD software are: Abalone [44], ACEMD [45], AMBER [36], DL-POLY [46], GROMACS [47], HOOMD-Blue [15, 48], LAMMPS [49, 50], NAMD [51], OpenMM [52].

At the time of writing, there are two standards for general-purpose computation on GPUs (GPGPU): NVIDIA's proprietary Compute Unified Device Architecture (CUDA) and the Open Computing Language (OpenCL) framework, developed by the Khronos group [53]. CUDA is developed specifically for NVIDIA

graphics cards and will not work on CPUs or AMD graphics cards, while OpenCL is a standard for heterogeneous computing, allowing the programmer to employ CPUs, GPUs, and APU (accelerated processing unit). The first version of CUDA was released in February 2007, whereas OpenCL was made available to the public in December 2008. Therefore, and because of the superiority of high-end NVIDIA GPUs at the time, much of the early work on molecular dynamics has been done using CUDA and it has become the de facto standard for scientific GPGPU. NVIDIA encourages this by collecting noticeable GPGPU publications in their GPU Gems series [54] and staying in close touch with the scientific community, e.g. through CUDA teaching and training at licensed academic centers. As a consequence, some very useful CUDA libraries and wrappers have been developed, such as cuBLAS (a GPU-accelerated version of the complete standard BLAS library) and Thrust (a high-level interface for routine GPGPU operations) [55, 56]. All this has led to CUDA being considered the more mature and often faster of the two options.

However, it looks like OpenCL is catching up with NVIDIA, both in maturity and performance. This is due to the fact that OpenCL is a heterogeneous computing standard, with most chip manufacturers (e.g. Intel, NVIDIA, AMD) actively contributing to development of the standard. As a consequence, OpenCL is a highly portable framework. In theory, an OpenCL program compiles just as well on an Intel dual-core CPU chip as on NVIDIA's high-end Tesla GPU. Recent results show that OpenCL and CUDA are on par in most basic math kernels, but also in real-world HPC applications [57, 58]. Furthermore, NVIDIA seems to have lost its top position in accessible GPGPU computing with the release of AMD's HD7970 GPU, which outperforms NVIDIA's flagship GTX680 by far in a range of OpenCL benchmarks. Also, more and more libraries are being developed for OpenCL, such as the ViennaCL linear algebra library [59]. This makes the choice for NVIDIA GPUs for GPGPU less obvious and opens doors for OpenCL. Because of its portability and the availability of AMD's HD7970 graphics card, all work in this thesis was conducted using OpenCL.

Both CUDA and OpenCL on the GPU employ a very similar SIMD-like (Single Instruction Multiple Data) data-parallel computing model. CUDA has coined this model Single Instruction Multiple Threads (SIMT), a relaxed version of SIMD with cheaper random access functions. The difference between conventional single CPU C++ programming and OpenCL GPU programming is best illustrated with a simple example. Say we have two vectors A and B of size n and we want to find their sum $C = A + B$. In C++ we would loop over all elements in B and A to sum the elements sequentially, as in Listing 2.1.

```

1 for(i=0;i<n;++i) {
  C[i]=A[i]+B[i];
3 }
```

Listing 2.1: C++ Vector Addition

In OpenCL, each thread (hereafter called work-item) has the ability to find its own identifying index. Then all we need to do is launch n work-items which will each execute the code in Listing 2.2. Here, every work-item i will read two elements from global memory and store their sum as one element in global memory. No element is read or written more than once.

```

1 __kernel void add(__global float *A, __global float *B, __global float *C) {
  int i = get_global_id(0);
3   C[i]=A[i]+B[i];
  }
```

Listing 2.2: OpenCL Vector Addition

This is a very clear and data-parallel example of the way OpenCL works. To contrast this with other parallel approaches, we might take a look at the data-parallel OpenMP implementation in Listing 2.3 from [60]. Here, each of NP threads is assigned a block of elements to sum. This approach is coarser than the very fine-grained OpenCL standard. The beauty is in the way that the OpenCL code in Listing 2.2 is mapped to different architectures by the implementation of chip manufacturers.

```

void vecadd(float *A, float *B, float *C, int N, int NP, int tid) {
2   int ept = N/NP ; // Elements per thread/work-item
   for(int i = tid*ept ; i < (tid+1)*ept ; i++) {
4     C[i]=A[i]+B[i];
   }
6 }
```

Listing 2.3: OpenMP Vector Addition

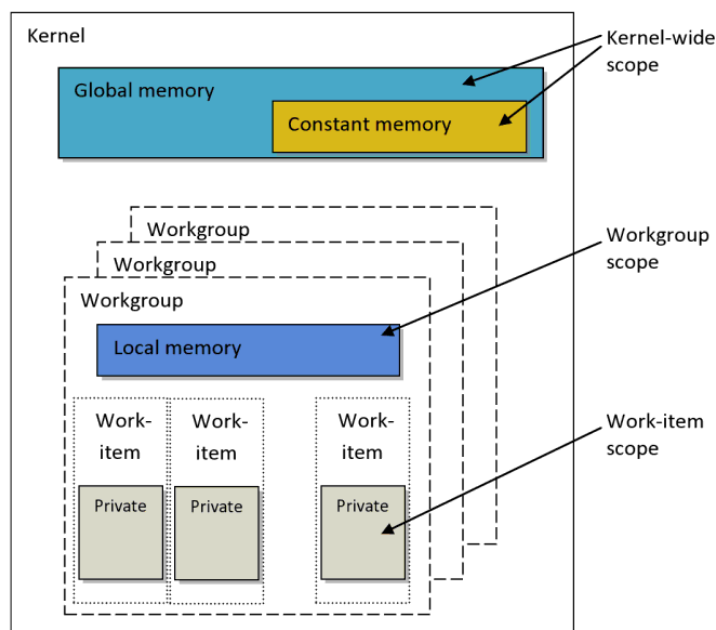


Figure 2.5: OpenCL memory model, with three memory layers: private work-item memory, shared local memory and global memory. Size typically increases as distance and access speed increase.¹

If we look at Figure 2.5, we see how work-items are organized into workgroups. Workgroups then are executed within a kernel, the functional building block of an OpenCL program. Every work-item has a very small piece of private memory, shares some local memory with the rest of the workgroup and has access to a larger global memory. Memory size is inversely related to access speed: work-items have the fastest access to their private memory, but this memory is very small and temporary (it does not persist after the kernel ends). Access to global memory is slow, but this kind of memory has a large capacity and persists through different kernel executions. In Listing 2.2 a kernel is executed on n work-items, organized in workgroups by the compiler. Each work-item i accesses global memory to read $A[i]$ and $B[i]$, adds the values and then writes their sum directly to global memory at position i in vector C . In a subsequent kernel execution, the value $C[i]$ is still there for further use.

In a GPGPU molecular dynamics simulation, for every particle we should keep at least its position and momentum in global memory in order to integrate the equations of motion and run a persistent simulation. In the partitioned Euler method which we employ here, it also makes sense to store the sum of forces on a particular particle. It then seems logical to map the steps in Figure 1.1 to kernels, i.e. have one kernel to compute forces, one kernel to integrate the equations of motion, and one kernel to perform measurements. In fact, these are three dependent steps: all forces need to be computed before we can integrate the equations of motion. And the equations of motion need to be solved before we can obtain statistics. This matches the OpenCL model (Figure 2.6). In an OpenCL program, there is both local and global synchronization. The programmer can put barriers within a workgroup to synchronize all work-items in the workgroup, but there is no way to force global synchronization across all work-items in different workgroups. This only happens in between kernels. Kernel 0 needs to be finished before kernel 1 can be executed.

¹Reprinted from *Heterogeneous Computing with OpenCL*, B. Gaster et al., Copyright (2012), with permission from Elsevier.

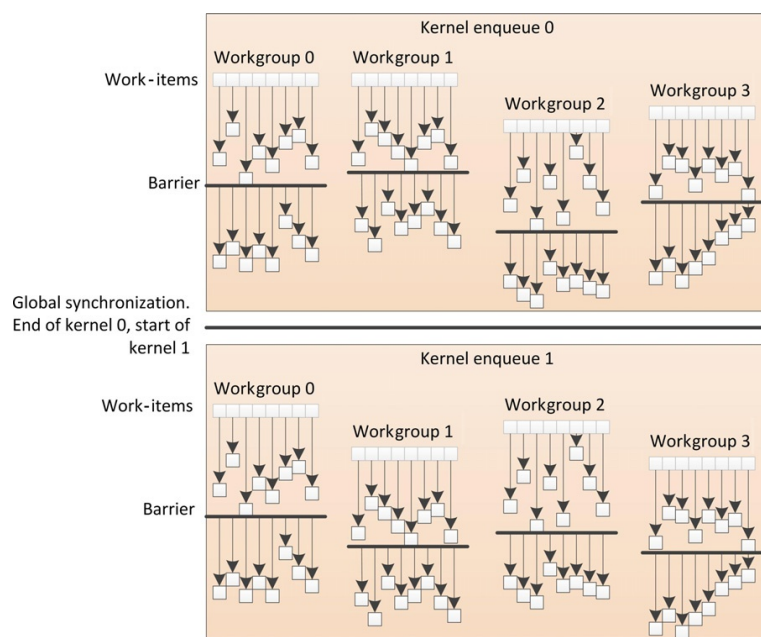


Figure 2.6: OpenCL synchronization. Work-items are organized in workgroups, workgroups are run within kernels. Synchronization happens within a workgroup or between kernels, but never between workgroups in the same kernel.¹

Adaptively Restrained Particle Simulations on Graphics Hardware

Contents

3.1	Introduction	17
	ARPS: Adaptively Restrained Particle Dynamics	18
	ARPS on CPU	19
3.2	ARPS on a GPU	20
	Updating forces	20
	Theoretical speedup	22
3.3	Implementation	24
	Mapping of algorithm to OpenCL kernels	25
	Prefix ordering	25
	Linked cell lists	26
	On the use of cell lists	27
	Langevin integration	29
3.4	Results	30
	Benchmark 1: Particle NVE argon Liquid	31
	Benchmark 2: Collision cascade	35
	Benchmark 3: Bonded Interactions	37
	Benchmark 4: Obtaining statistics in the NVT ensemble	40
3.5	Conclusion and discussion	41

3.1 Introduction

In the previous chapter we have introduced molecular dynamics (MD). MD simulations are a vital tool in many fields of research, providing insight into a large variety of biological and chemical processes. However, the simulated time is often not nearly at the scale of the observed process. For instance in protein folding, we have seen that the most powerful machines right now simulate 1ms, whereas the whole folding process might take seconds or minutes. Furthermore, simulating time increases with system size. Therefore, there is an ongoing quest for fast MD simulations. Optimally, MD simulations should adapt to a particular problem to be more efficient.

Numerous methods have been proposed to accelerate molecular dynamics simulations. In the previous section we have seen some implementation improvements, such as cell grids with cutoff distance, periodic boundary conditions and the use of GPUs to accelerate MD simulations. In all of these approaches, we still compute the original Hamiltonian and do not reduce the number of degrees of freedom in the system. Alternative methods simplify the system or equations of motion to obtain results with a predictable shift or perturbation. One might for instance reduce the number of degrees of freedom in the system by using

coarse-grained representation [61], where there is one bead or 'superparticle' for every 3 or 4 atoms, or use longer time steps [62] to increase the simulated time. Also, force approximations might be used, e.g. in the fast multipole method introduced in [63], or a partial update [64, 65, 66] of the forces might be performed.

Recently, Adaptively Restrained Particle Simulations (ARPS) were proposed [1]. In this Hamiltonian-based approach, a modified inverse inertia matrix is used, which allows to switch positional degrees of freedom on and off in the system during the simulation. As a result, under common assumptions on the potential function, less forces may be computed at each time step.

In this chapter we propose an algorithm carrying out ARPS on a GPU. More precisely, we present a very general implementation of ARPS on the GPU and compare the number of force computations of ARPS on a GPU with those in a conventional, full dynamics GPU implementation. We show that for large systems, the reduction in the number of force computations is reflected in the speedup of ARPS with respect to the full dynamics approach. There are numerous MD implementations on GPUs, but to our knowledge this is the first implementation which adaptively switches degrees of freedom on and off.

The rest of the chapter is organized as follows. First, we give a short review of ARPS. Then, we introduce ARPS on GPU, a model for the speedup and illustrate this method with several results including bonded and non-bonded force computations in the NVE and NVT ensembles. Finally, we present some conclusions and discuss possible future work.

ARPS: Adaptively Restrained Particle Dynamics

In the previous chapter we have seen how a system of N particles in 3-D is described by the Hamiltonian function $H(\mathbf{q}, \mathbf{p})$ in Eq. (2.2).

Typically, the potential term $V(\mathbf{q})$ may be written as a sum of terms that depend on a few relative positional degrees of freedom only, when we do not compute all possible interatomic interactions (e.g. due to a cutoff distance) [61].

The ARPS approach [1] introduces an adaptively restrained (AR) Hamiltonian $H_{AR}(\mathbf{q}, \mathbf{p})$ with a modified inverse inertia matrix $\Phi(\mathbf{q}, \mathbf{p})$ to replace the standard matrix \mathbf{M}^{-1} :

$$H_{AR}(\mathbf{q}, \mathbf{p}) = \frac{1}{2} \mathbf{p}^T \Phi(\mathbf{q}, \mathbf{p}) \mathbf{p} + V(\mathbf{q}). \quad (3.1)$$

In [1], $\Phi(\mathbf{q}, \mathbf{p})$ for simulations in Cartesian coordinates has been proposed. This matrix adaptively switches on and off positional degrees of freedom in the system during the simulation. Precisely, these degrees of freedom are individual positional degrees of freedom of each particle, and they are turned on and off independently: $\phi_i(\mathbf{q}_i, \mathbf{p}_i) = m_i^{-1}(1 - \rho_i(\mathbf{q}_i, \mathbf{p}_i))$, $1 \leq i \leq N$. Here $\rho_i(\mathbf{q}_i, \mathbf{p}_i) \in [0, 1]$ is a twice-differentiable restraining function. When $\rho_i(\mathbf{q}_i, \mathbf{p}_i) = 0$ (no restraining), $\phi_i(\mathbf{q}_i, \mathbf{p}_i) = m_i^{-1}$ and the particle is following full dynamics. When $\rho_i(\mathbf{q}_i, \mathbf{p}_i) = 1$ (full restraining), $\phi_i(\mathbf{q}_i, \mathbf{p}_i) = 0$ and the particle is fully-restrained: it is not moving, whatever the force applied to it. A particle is called **non-restrained** or **active** if it's not fully-restrained. When $\rho_i(\mathbf{q}_i, \mathbf{p}_i) \in (0, 1)$, the particle smoothly switches between both behaviors.

The restraining function was chosen to be dependent on a particle's kinetic energy: $K_i(\mathbf{p}_i) = \mathbf{p}_i^T \mathbf{p}_i / 2m_i$. When this kinetic energy becomes smaller than a certain threshold ε_i^f , the particle stops moving. This is a natural strategy, as we restrain particles that are moving less. The second threshold ε_i^s was also introduced to make the switching smooth:

$$\rho_i(\mathbf{p}_i) = \begin{cases} 1, & \text{if } 0 \leq K_i(\mathbf{p}_i) \leq \varepsilon_i^f, \\ 0, & \text{if } K_i(\mathbf{p}_i) \geq \varepsilon_i^s, \\ s(K_i(\mathbf{p}_i)) \in [0, 1], & \text{elsewhere,} \end{cases} \quad (3.2)$$

where $s(\varepsilon)$ is a twice-differentiable switching function with respect to its argument and, therefore, to \mathbf{p}_i , $s(\varepsilon_i^f) = 1$, $s(\varepsilon_i^s) = 0$.

The equations of motion for each particle can, therefore, be derived from the AR Hamiltonian:

$$\begin{aligned} \dot{\mathbf{p}}_i &= -\frac{\partial H_{AR}}{\partial \mathbf{q}_i} = -\frac{\partial V}{\partial \mathbf{q}_i}, \\ \dot{\mathbf{q}}_i &= \frac{\partial H_{AR}}{\partial \mathbf{p}_i} = \frac{\mathbf{p}_i}{m_i} (1 - \rho_i(\mathbf{p}_i)) - \frac{1}{2m_i} \frac{\mathbf{p}_i^2}{\partial \mathbf{p}_i} \frac{\partial \rho_i(\mathbf{p}_i)}{\partial \mathbf{p}_i}. \end{aligned} \quad (3.3)$$

From these equations one can see that when the particle is fully-restrained ($\rho_i = 1$) it stops ($\dot{q}_i = 0$). However, the particle's momentum will continue to evolve as in a non-adaptively restrained simulation and, at some time step, the kinetic energy will again exceed the threshold so that the particle resumes moving.

As the Hamiltonian H_{AR} is separable, several symplectic and explicit integration schemes are available to propagate Eq. (3.3) in time, for example the symplectic Euler integrator [38] in the NVE ensemble and the Langevin integration scheme introduced in Chapter 2 in the NVT ensemble.

ARPS on CPU

In accordance with [1], let us describe one step of integration of Eq. (3.3) with the symplectic Euler scheme [38] on CPU: starting from the current configuration $(\mathbf{q}_n^i, \mathbf{p}_n^i)$ for every i th particle, we want to obtain the next system configuration $(\mathbf{q}_{n+1}^i, \mathbf{p}_{n+1}^i)$. In Section 2.2 we have seen that in this scheme we first update momentum and then position. Hence, each such step of integration consists of four major sub-steps (Figure 3.1):

1. *Force update*: the forces $\mathbf{f}_{n+1}^i = -\partial V/\partial \mathbf{q}_n^i$ acting on each particle are updated (if necessary) according to current positions and the restraining function.
2. *Momenta update*: for each particle, its momentum is updated:

$$\mathbf{p}_{n+1}^i = \mathbf{p}_n^i + \mathbf{f}_{n+1}^i \Delta t.$$

3. *Restraining function update*: for each i th particle $\rho_{n+1}^i(\mathbf{p}_{n+1}^i)$ is computed according to Eq. (3.2).
4. *Position update*: for each particle, its position is updated:

$$\mathbf{q}_{n+1}^i = \mathbf{q}_n^i + \left(\frac{\mathbf{p}_{n+1}^i}{m^i} (1 - \rho^i(\mathbf{p}_{n+1}^i)) - 0.5 \frac{\|\mathbf{p}_{n+1}^i\|^2}{m^i} \frac{\partial \rho^i(\mathbf{p}_{n+1}^i)}{\partial \mathbf{p}_{n+1}^i} \right) \Delta t.$$

With ARPS less forces need to be updated at each time step because, e.g. for a pairwise potential, the forces between pairs of fully-restrained particles do not change when particles do not change positions. This reduces the time needed in the *Force update* step significantly, explaining the large speedups reported in [1].

Next, we describe the novel algorithm implementing ARPS on a GPU.

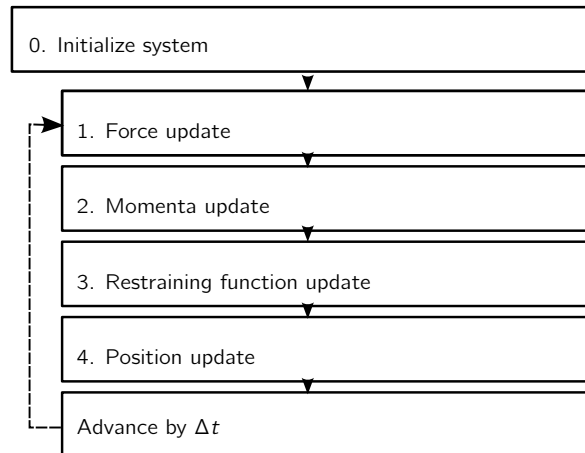


Figure 3.1: Steps in ARPS. These steps are different from those in Figure 1.1 in that a step is inserted between *Momenta update* and *Position update*, where we determine the restraining function of each particle.

3.2 ARPS on a GPU

We propose to extend the application of ARPS from CPU to GPU. Each step of integration on the GPU consists of the same major sub steps as the CPU algorithm. The *Momenta update*, *Restraining function update* and *Position update* steps are identical to those in the CPU algorithm (Figure 3.1), apart from the fact that they are executed in a massively parallel environment, i.e. for each particle there is an independent work-item which computes \mathbf{p}_{n+1}^i , \mathbf{q}_{n+1}^i and ρ_{n+1}^i . Updating momenta, restraining function and position is data-parallel: every thread only needs to load data for one particle from the global memory. Hence it is also embarrassingly parallel. Therefore this is perfectly suited for the SIMT GPU approach.

The main difference between the CPU and the GPU algorithm is in the *Force update* step. This is the step which generally takes most time in a molecular dynamics simulation and it is thus not surprising that this is where we try to decrease computation times. We should take the following into account. As was discussed in the previous chapter, in CPU implementations it is common to employ Newton's third law ($\mathbf{f}_{ji} = f(\mathbf{q}_j - \mathbf{q}_i) = -f(\mathbf{q}_i - \mathbf{q}_j) = -\mathbf{f}_{ij}$) to reduce the number of pairwise force evaluations by 50%. This reduction is generally avoided on the GPU due to possible memory access conflicts. Imagine particle k having neighbors i and j and their corresponding work-items. These work-items cannot add \mathbf{f}_{ji} and \mathbf{f}_{ki} to $F[i]$ at the same time. Atomic operations could solve this, but as of now there are no proper *atomic_add* operations that work on double precision floating point values in OpenCL. There are workarounds for single precision floating point operations, but in order to keep our approach scalable and applicable to double precision values we will maintain a redundant computation strategy, where the interaction between two particles is computed by both their corresponding work-items. The disadvantage of this method is obviously in the 50% redundant computations, but it's advantageous in that it's data-parallel and does not need complex and time-consuming reduction operations [67].

Updating forces

Speedups reported for MD implementations on the GPU are usually due to a radical decrease in the time needed to update forces. One work-item is mapped to each particle and all forces acting on that particle are computed by going over all its interaction neighbors. The highly multi-threaded nature of GPUs implies that hundreds of forces can be computed in parallel. For bonded interactions, neighbor lists never require updating and thus can be generated on the CPU and then transferred to the GPU. In the non-bonded case, the set of neighbors usually depends on some cutoff distance r_c . As particles move, the neighbor list of each particle requires constant updating. It is this case where we will need an enhanced algorithm.

In [40], two algorithms to update forces in ARPS are explored. In the first algorithm, lists of 4-tuples $(a_i, a_j, \mathbf{r}_{ij}, \mathbf{f}_{ij})$ are updated and switched at every time step, where a_i and a_j are indexes of neighboring particles, \mathbf{r}_{ij} is the distance vector s.t. $\|\mathbf{r}_{ij}\| < r_c + \delta$ and \mathbf{f}_{ij} is the force that particle a_j exerts on particle a_i . This list is updated every few steps, whenever some particle has moved more than half the shell width

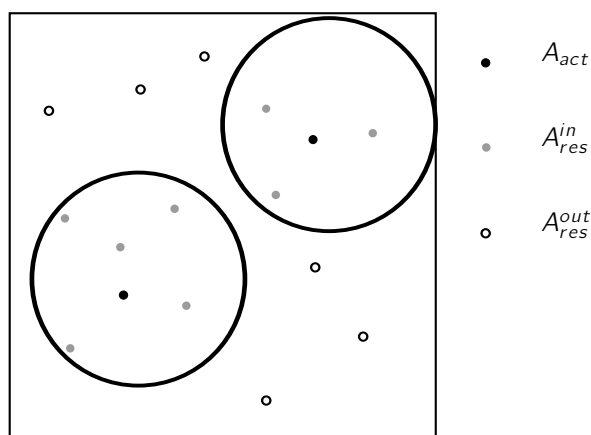


Figure 3.2: The set of active particles A_{act} and restrained particles A_{res} with subset A_{res}^{in} .

δ in Figure 3.10. This well known neighbor list approach is very popular and has turned out to be very useful in MD GPU simulations.

The second method in [40] employs the cell grid in Figure 2.4 and is a so-called cell list algorithm. At every time step, cell grids are built or updated and then used for evaluation of the pairwise interactions. The disadvantage is that there are a lot of redundant distance evaluations (i.e. the vertically shaded area in Figure 2.4). However, ARPS significantly reduces the number of redundant computations. It is this approach that we will use here. In Section 3.3 we further motivate the choice for cell lists in ARPS.

For the sake of our algorithm we split the set of particles A into three parts. We define the set of active particles A_{act} and the set of restrained particle A_{res} , such that $A = A_{act} \cup A_{res}$ is the complete set of particles. Then we further split A_{res} into A_{res}^{in} and A_{res}/A_{res}^{in} , where the particles in A_{res}^{in} have at least one active neighbor and the particles in A_{res}/A_{res}^{in} have no active neighbors, as in Figure 3.2. Before every *Force update* step the set of particles A is split into A_{act} and A_{res} . The *Force update* step (step 1 in Figure 3.1) at iteration n in ARPS on a GPU contains the following sub steps:

- 1.1 *Build grid*: map all particles in A_{act} to a grid according to their **old** positions \mathbf{q}_{n-1}
- 1.2 *Subtract old restrained forces*: subtract old forces of particles in A_{act} on particles in A_{res} using cell grid
- 1.3 *Build grid*: map all particles in A_{act} to a grid according to their **new** positions \mathbf{q}_n
- 1.4 *Add new restrained forces*: add new forces of particles in A_{act} on particles in A_{res}^{in} using cell grid
- 1.5 *Update grid*: add particles in A_{res}^{in} to cell grid
- 1.6 *Add new active forces*: add new forces of particles in $A_{res}^{in} \cup A_{act}$ on particles in A_{act} using cell grids

Figure 3.3 shows a four-particle system where two particles are active (closed dots) and two particles are restrained (open dots). We will now explain the sub steps in the algorithm based on this example. There are only three sub steps (1.2, 1.4 and 1.6) in which we actually compute forces. Sub steps 1.1, 1.3 and 1.5 act as auxiliary steps, which allow us to use a cell grid algorithm. In sub step 1.2 we subtract the old forces of active particles on restrained particles. Then we move the active particles to their new positions in the grid and add the new forces of these particles on the restrained particles. Now the total

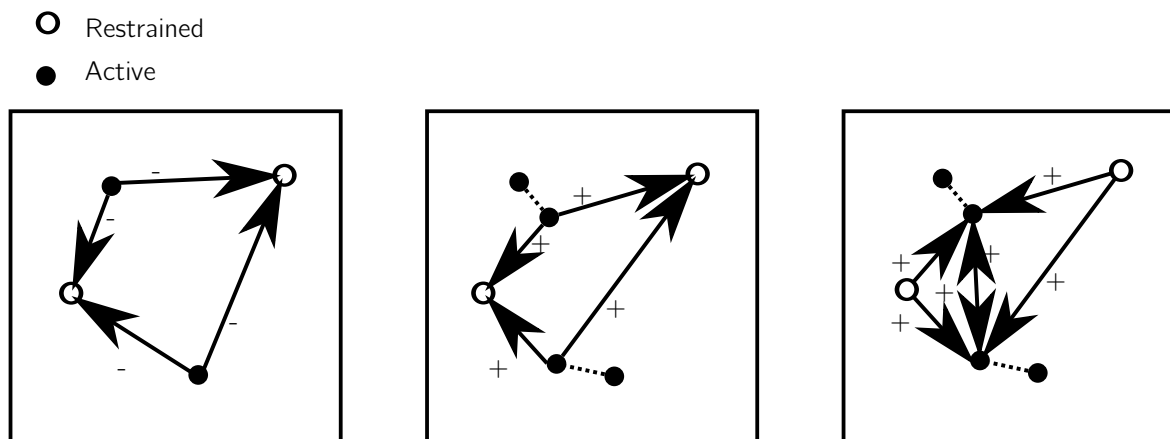
1.2 *Subtract old restrained forces*1.4 *Add new restrained forces*1.6 *Add new active forces*

Figure 3.3: Sub steps of the *Force update* step in Figure 3.1. Auxiliary steps 1.1, 1.3 and 1.5 are not shown here. In this example, the active particles (black dots) move, whereas the restrained particles do not move. In step 1.2 (*Subtract old restrained forces*) we subtract the old forces of active particles on restrained particles. Then, active particles are moved to their new positions. In step 1.4 (*Add new restrained forces*) we add the new forces of active particles on restrained particles. In step 1.6 (*Add new active forces*) we sum all forces on active particles based on the positions of the other particles (both restrained and active).

force on each restrained particle is the sum of forces of both restrained and active neighbors on that particle. Finally, in sub step 1.6 we compute all forces on each active particle. None of the forces on this particle is the same as in the previous time step, because its position has changed.

When we consecutively perform steps 1.1 through 1.6, the total force on each particle is the same as when we would sum the force of all particles on the particle. The speedup due to ARPS and reported on in [1] is caused by a reduction in the number of forces that are computed at every time step: in Figure 3.3 we do not compute forces between restrained particles. In Figure 3.4 we see how in ARPS the number of updated forces is a function of the number of active particles. For clarity, this figure shows long-range interactions, without a cutoff. For short-range interactions, only subsets of the lattices would be gray.

At this point, we can get an estimate on the number of forces to be computed in a full dynamics and an ARPS implementation on the GPU.

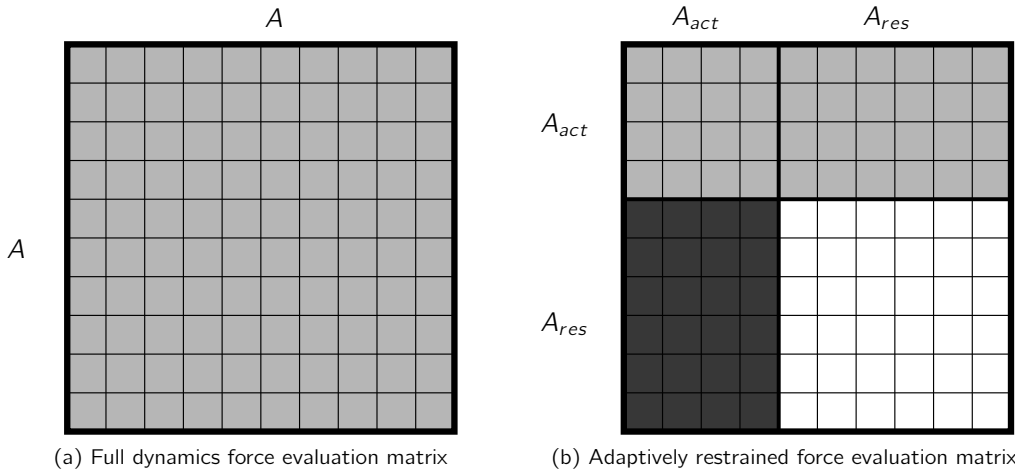


Figure 3.4: Forces to be evaluated in all-pairs MD (a) and adaptively restrained particle simulations (b). In ARPS on a GPU the white area is ignored, the light area is computed once and the dark area is computed twice. In this particular case, the number of force evaluations in ARPS on a GPU is only 12% smaller than in the full dynamics algorithm. Interestingly, if the number of active particles passes 50%, ARPS actually requires more force evaluations than a full dynamics algorithm.

Theoretical speedup

Before we look at any computational results, we analyze the theoretical speedups which might be reached by both ARPS on CPU and ARPS on a GPU, with respect to their full dynamics counterparts. For the CPU, we model according to the cell list algorithm in [40], which was also used to obtain results in [1].

We develop a simple model to count the number of force evaluations in a system of N particles, where a particle has on average N_n neighbors (for short-range forces, this is the number of neighbors within cutoff distance) and the ratio of active particles is given as $0 \leq \rho_{act} \leq 1$:

$$\text{CPU full dynamics: } N_F^{FD} = \frac{1}{2} NN_n.$$

$$\begin{aligned} \text{CPU ARPS: } N_F^{ARPS} &= \rho_{act} NN_n (1 - \rho_{act}) + \rho_{act} NN_n (1 - \rho_{act}) + \rho_{act} \frac{1}{2} NN_n \rho_{act} \\ &= \rho_{act} NN_n \left(2 - \frac{1}{2} \rho_{act} \right). \end{aligned}$$

The estimate for N_F^{FD} comes from the fact that we need to compute NN_n inter-atomic interactions in the full dynamics implementation, which is cut in half by Newton's third law. The estimate for N_F^{ARPS} is based on the fact that for every active particle, we first subtract the old, then add the new force exerted by that particle on its $(1 - \rho_{act})N_n$ restrained neighbors. Newton's third law is used to then add the negative of the new force to the active particle. Finally, we should compute $\frac{1}{2}\rho_{act}NN_n$ forces between an active particle and its restrained neighbors.

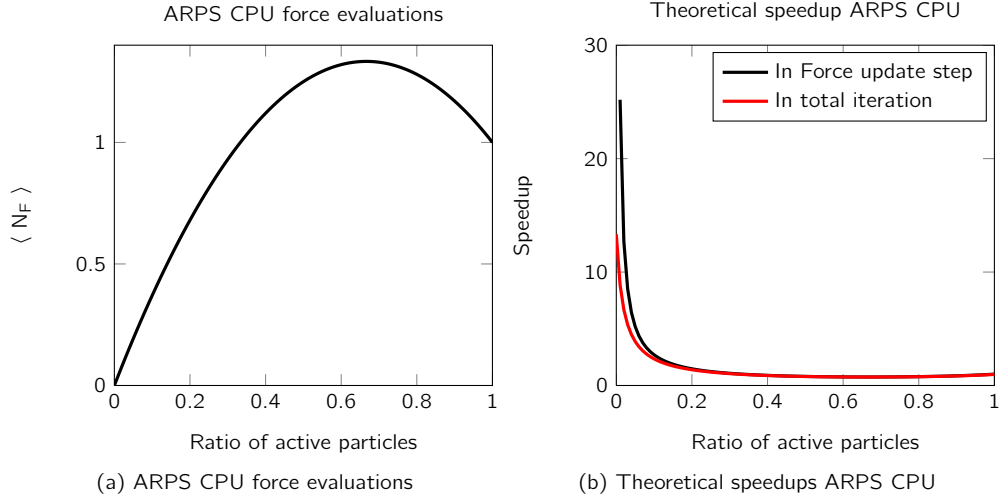


Figure 3.5: Number of force evaluations as a function of the ratio of active particles (a) and theoretical speedup in the *Force update* step and the overall iteration with $T_F = 0.9$ (b) in ARPS on CPU.

Then the ratio ρ_{CPU} of force evaluations in ARPS on CPU w.r.t. a full dynamics algorithm is given as

$$\rho_{CPU} = \frac{N_F^{ARPS}}{N_F^{FD}} = \frac{\rho_{act} N N_n (2 - \frac{1}{2} \rho_{act})}{\frac{1}{2} N N_n} = \rho_{act} (4 - 3\rho_{act}),$$

the curve of which is plotted in Figure 3.5a.

Speedup due to the reduced number of force evaluations is only noticeable in the *Force update* step in Figure 3.1. Because this step usually accounts for more than 90% of each iteration, it seems reasonable to estimate the speedup as $\alpha_{CPU} = \frac{1}{\rho_{CPU}}$, which is also plotted in Figure 3.5b. For a small number of active particles however, we see that the value of $\langle N_F \rangle$ gets very low. Then it might be better to take the other steps into account as well. We estimate total speedup as

$$\alpha_{CPU} = \frac{1}{T_F \rho_{CPU} + T_O}, \quad (3.4)$$

where T_F is the ratio of time that *Force update* takes in a full dynamics algorithm and T_O is the ratio of time that the other steps take, such that $T_F + T_O = 1$.

On the GPU, we do not use Newton's third law. Hence, we can estimate the number of force evaluations in the full dynamics algorithm as

$$\text{GPU full dynamics: } N_F^{FD} = N N_n,$$

where every particle computes one force for each of its neighbors. In ARPS on a GPU we evaluate $\rho_{act} N (1 - \rho_{act}) N_n$ forces in steps 1.2 and 1.4, and $\rho_{act} N N_n$ forces in step 1.6. This sums to

$$\text{ARPS on a GPU: } N_F^{ARPS} = \rho_{act} N N_n (3 - 2\rho_{act}),$$

and we get

$$\rho_{GPU} = \frac{N_F^{ARPS}}{N_F^{FD}} = \frac{\rho_{act} N N_n (3 - 2\rho_{act})}{N N_n} = \rho_{act} (3 - 2\rho_{act}).$$

The speedup is computed as in Eq. (3.4). From this, and Figures 3.5b and 3.6b, it is clear that the reduction in force evaluations on the GPU using ARPS is slightly more effective than on the CPU. This is especially noticeable when there is a small number of active particles, i.e. when ρ_{act} is small. This is due to the fact that on GPU, we cannot use Newton's third law. ARPS on a GPU adds less redundant force computations than on the CPU, where there is a penalty in computing some forces twice every time step.

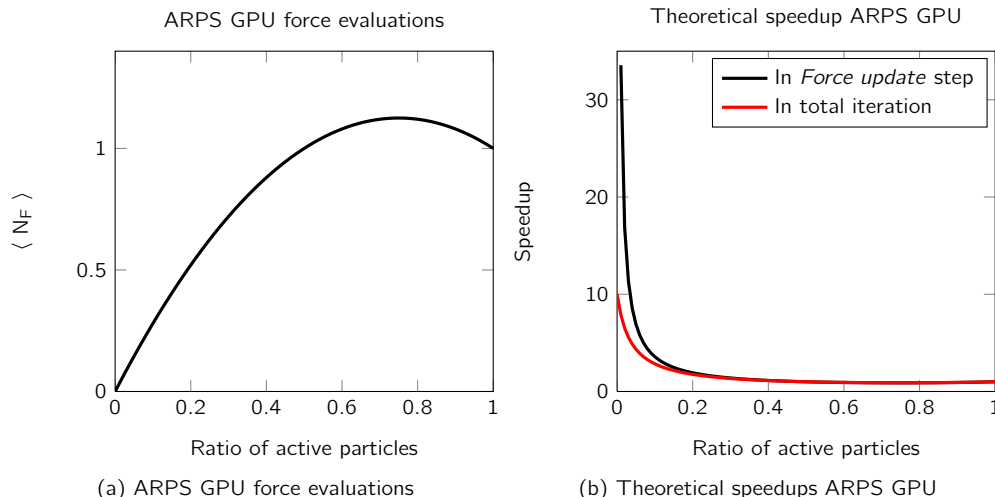


Figure 3.6: Number of force evaluations as a function of the ratio of active particles (a) and theoretical speedup in the *Force update* step and the overall iteration with $T_F = 0.9$ (b) in ARPS on a GPU.

3.3 Implementation

The ARPS on GPU algorithm was implemented in OpenCL with C bindings, according to the OpenCL 1.1 specification. Here, we will discuss some details which are specific to implementation on the GPU. First, we note that because GPUs are developed to deal with geometric graphics data, there is very good native behavior on 2-, 4-, 8- or 16-element vectors, where e.g. reading a `double4` vector requires just one operation, instead of an individual operation for each element. Also, native functions for vector multiplication or norm computation are available. We use these native data types whenever possible. Hence, forces, positions and momenta are represented by the `double4` type. This is preferable over the `double3` type, because the data aligns better in memory. In Listing 3.1 we see the force computation function, where the `double2` and `double4` native types are used along with native functions `dot`, `pow`, `pown` and vector multiplication to compute a switched Lennard Jones force (first three elements of return vector) and potential (fourth element of return vector).

In Listing 3.1 we use double precision floating point numbers, as we do throughout the whole implementation. This greatly increases accuracy and stability of the simulations, albeit at some additional computational cost. Nevertheless, it has been shown that single precision floating point numbers and operations can cause serious accuracy and reproducibility problems in MD on GPU implementations [68, 69]. Most modern GPUs support double precision operators and in OpenCL the double data type follows IEEE specifications. Also, the full dynamics GPU algorithm with which we compare ARPS on GPU was implemented using double precision numbers, hence the computational penalty is reflected in both implementations.

```

// Give Lennard-Jones force with cutoff distance r_c=8 and switch at r_s=7.5
2 double4 getForce(double4 drp)
  {
4   double r_2, r_6, r_8, r_14, c2, c4, pot ;
   double r2 = dot(drp, drp) ;
6   double kB = 0.00198721;
   double eta = 120*kB;
8   double s12 = pow(3.4,12);
   double s6 = pow(3.4,6);
10  double cutoff2 = 64 ;
   double switchDist2 = 56.25 ;
12  double c3 = 3*(cutoff2-switchDist2) ;
   double c0 = 1.0/(cutoff2 - switchDist2);
   double c1 = c0*c0*c0;
14  if(r2<=cutoff2 && r2 != 0)
16  {
   r_2 = 1.0/r2 ;
18  r_6 = pown(r_2,3) ;

```



```

20     r_8=r_6*r_2;
    r_14=r_6*r_8;

22     c2 = cutoff2 - r2;
    c4 = c2*(c3 - 2.0*c2);

24     double2 switchVal = (r2 > switchDist2) ? (double2){c2*c4*c1, 2*c1*(6*c2*
        c2-2*c2*c3)} : (double2){1.0, 0.0} ;

26     pot = 4*eta*r_6*s6*(s6*r_6-1.0);
28     double4 trap = {0.0, 0.0, 0.0, 1.0} ;
    return (trap*pot+(4*eta*(-12.0*r_14*s12+6.0*r_8*s6)*drp))*switchVal.x +
        pot*switchVal.y*drp ;
30 }
32 return (double4)(0.0) ;
}

```

Listing 3.1: OpenCL LJ force and potential computation function

Mapping of algorithm to OpenCL kernels

Generally, steps in the ARPS on GPU algorithm map to OpenCL kernels. However, as we discussed in Section 2.4, OpenCL only allows synchronization between kernels. Therefore, we have to spread step 1.3 (where we build the grid of active particles) over two kernels `clean_up` and `update_cells`, where we first reset the cell lists and then rebuild them.

On the other hand, sometimes we are able to put multiple steps of the algorithm in one kernel. We don't need global synchronization between steps 2, 3 and 4, so these steps can share a single kernel. Also because of this, we are able to build the grid of active particles in their old positions as part of the `update_positions` kernel. Hence the right column of Table 3.1 is slightly shifted in order to follow the flow of the algorithm. In our experiments, the grid is initialized with N active particles. In the `prefix_bins` and `assign_order` kernels we split the set of particles A into A_{act} and A_{res} . Then in the `add_new_restrained_forces` kernels the $A_{res}^{in} \subseteq A_{res}$ set is determined, after which the `update_cells_two` adds them to the existing cell grid.

We will now discuss some kernel-specific implementation details.

	Step	Kernel
1.1	<i>Build grid</i>	<code>update_positions</code>
		<code>prefix_bins</code>
		<code>assign_order</code>
1.2	<i>Subtract old restrained forces</i>	<code>subtract_old_restrained_forces</code>
1.3	<i>Build grid</i>	<code>clean_up</code>
		<code>update_cells</code>
1.4	<i>Add new restrained forces</i>	<code>add_new_restrained_forces</code>
1.5	<i>Update grid</i>	<code>update_cells_two</code>
1.6	<i>Add new active forces</i>	<code>update_active_forces</code>
		<code>clean_up</code>
2	<i>Momenta update</i>	<code>update_positions</code>
3	<i>Restraining function update</i>	
4	<i>Position update</i>	

Table 3.1: Mapping of algorithm steps to OpenCL kernels.

Prefix ordering

To update the order lists, which contain all particle indexes in A_{act} or all particle indexes in A_{res} , we perform a stream reduction on the complete list of particle indexes A . We perform a prefix scan on the list with 1 at entries to be taken into account and 0 elsewhere. By first computing the individual index of each work item's particle in the set of interest within a workgroup using reduction and then

combining the workgroup results, every relevant particle gets a unique index in the order list. The reduction within a workgroup is performed using a tree with $\log_2(\text{local_size})$ levels as in Figure 3.7, where `local_size` is 256 in our experiments. This is done at the end of the `update_positions` kernel. Then in the `prefix_bins` kernel, results of all workgroups are combined and in `assign_order` indexes are assigned to individual work items so that work items in workgroup $n + 1$ get indexes which extend those in workgroup n . In Figure 3.7 we see how a workgroup which contains 8 work items has 5 active particles and 3 restrained particles. Hence the first active particle in the next workgroup will get index 5 in the list of active particles and the first restrained particle in the next workgroup will get index 3 in the list of restrained particles. These indexes are used to build a list of active particles or restrained particles so that only the necessary number of work items need to be launched in the `subtract_old_forces`, `add_new_forces` and `update_active_forces` kernels.

Though prefix scans and summation are often the cleanest and most parallel approach, sometimes an implementation using global index assignment might be profitable. Here, one global counter assigns indexes to work items through an atomic add operation. For the prefix scan method we need $2\log_2(\text{local_size})$ steps, each with synchronization within workgroups. This is costly too and may cause warp divergence. Furthermore, with the advent of new GPU architectures, significant increases in the speed of atomic operations have been gained. In [70], it is shown that in some cases it is advantageous to use atomic operations over scans. Furthermore, using atomic functions reduces the length of the code by almost an order of magnitude. Finally, ARPS is intended to be used in cases where N_{act} is small. As the number of possible memory conflicts is bounded by N_{act} , this might be an alternative useful approach.

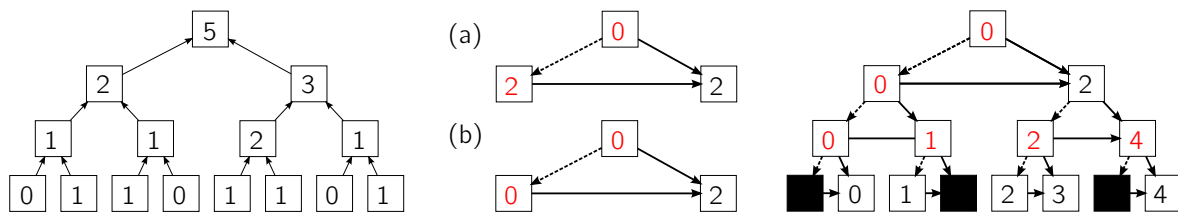


Figure 3.7: Index designation in $\log_2 n$ steps using reduction. First, values are summed in a bottom-up pass. Then at each level of the top-down pass, the left leaf value is passed to the right leaf (a) and the value of the node is passed to the left leaf and added to the right leaf (b). Finally, all nonzero elements have a unique index.

Linked cell lists

In Section 3.1 we have introduced cell grids, where the domain is split up in cells with width $\geq r_c$. In steps 1.1 and 1.3 of the algorithm these cell grids are built from scratch and in step 1.5 the cell grid built in step 1.3 is updated. The advantage of building the grid in two stages (steps 1.3 and 1.5) is clear if we look at Figure 3.8. We see how active particle k has many more relevant neighbors than restrained particle l . To reduce the number of random memory accesses in steps 1.2 and 1.4, we want the kernels to use a sparse cell grid representation, where the sparseness depends on N_{act} . Then in step 1.5, this sparse grid is extended to the full grid we see on the right side of Figure 3.8.

Several approaches to cell list or neighbor list keeping on the GPU have been proposed. Most of these methods allocate memory for each cell, bounded by the expected maximum number of cell members. Because the number of active or restrained particles per cell, as well as the actual number of particles per cell, might fluctuate strongly, setting a maximum cell capacity and then keeping a member array per cell would be both arbitrary and wasteful. To make optimal use of ARPS and reduce redundant random memory access, we want to rebuild the cell grid on a regular basis.

We propose to use linked cell lists to keep the cell grid [71]. In Figure 3.9 we see a small grid with four cells and ten particles. The goal is to obtain a linked list for each cell which contains the particles in that cell. In a linked list, every element contains a data field as well as a pointer to the next element in the list. The last element in the list, called the 'tail', points to a **null** value, in our implementation this value is -1 . In our case, the required data field is just the particle index. Hence, we create a linked list where every element contains a pointer to the index of the next element in the list. We keep two arrays, a `cells` array and a `particles` array. The `cells` array contains one element for every cell, the

`particles` array contains one element for every particle. Initially, every element in `cells` is set to -1 , the null value. Every element in `particles` is set to its index. Then the data structure is as follows:

	0	1	2	3	4	5	6	7	8	9
<code>cells</code>	-1	-1	-1	-1						
<code>particles</code>	0	1	2	3	4	5	6	7	8	9

Now we add every particle to its corresponding cell. First we add particle 0 to cell 0. We swap the value at `particles[0]` and the value at `cells[0]`. Now `cells[0]` points to particle 0 at `particles[0]`, which points to null. Cell 0 contains just particle 0. The data structure is:

	0	1	2	3	4	5	6	7	8	9
<code>cells</code>	0	-1	-1	-1						
<code>particles</code>	-1	1	2	3	4	5	6	7	8	9

If now we add particle 1 by swapping `particles[1]` and `cells[0]` and then add particle 5 by swapping `particles[5]` and `cells[0]` we get:

	0	1	2	3	4	5	6	7	8	9
<code>cells</code>	5	-1	-1	-1						
<code>particles</code>	-1	0	2	3	4	1	6	7	8	9

We can now iterate over all particles in cell 0: first we find that `cells[0] = 5`. Then we see that `particles[5] = 1`, `particles[1] = 0` and `particles[0] = -1`. We stop because we have found the tail of the list. The list contains particles 5, 1 and 0.

If we construct the lists for the other cells in the same way, we get:

	0	1	2	3	4	5	6	7	8	9
<code>cells</code>	5	4	9	8						
<code>particles</code>	-1	0	-1	-1	2	1	-1	6	7	3

Like the rest of the methods introduced here, we have implemented this algorithm in OpenCL. An interesting observation is that it does not matter whether we add e.g. particle 0 or particle 1 first to cell 0. The linked list will still contain the same items. Therefore, we let each particle add itself to the linked cell list. A work item i corresponding to particle i in cell c swaps values `particles[i]` and `cells[c]`. A swap consists of reading the old value at `cells[c]`, replacing that value with the one at `particles[i]` and then replacing the value at `particles[i]` with the old value at `cells[c]`. These three operations cannot be interrupted. If we let all work items swap at the same time, race conditions occur. Luckily, OpenCL offers a relatively fast atomic swap operations, which guarantees that all swaps will occur sequentially. In steps 1.1 and 1.3 we need $\rho_{act}N_c$ swaps per cell, where N_c is the mean cell occupancy and ρ_{act} is the ratio of active particles. In step 1.5 the number of swaps is determined by the probability of a restrained particle having an active neighbor. For denser systems, this is often 1 and hence the number of swaps in a cell is equal to $(1 - \rho_{act})N_c$.

On the use of cell lists

Over the past few years, there has been some discussion on the use of neighbor lists and cell lists for MD on the GPU for the evaluation of non-bonded short-range forces. In [16], it was suggested that cell lists are more suitable for GPU implementations due to hardware limitations. In [15] however, empirical results for cell and neighbor list are compared and the neighbor list is found to be 3.2 times faster on identical simulations. Other successful implementations of neighbor lists on the GPU, as well as a more theoretical analysis in [72] seem to support this idea and hence neighbor lists are the de facto standard in MD GPU implementations.

Updating a neighbor list comes at a cost, e.g. computing all distances between particles in adjacent cells. Because the sets of active particles A_{act} and restrained particles A_{res} change at every timestep, and we only wish to update the neighbor list every few steps, we cannot split one particle's neighbor list Ω into two separate lists for active particles and restrained particles. Hence these need to be contained within one list. Then in all three steps 1.2, 1.4 and 1.6 of the algorithm, we would need to iterate over

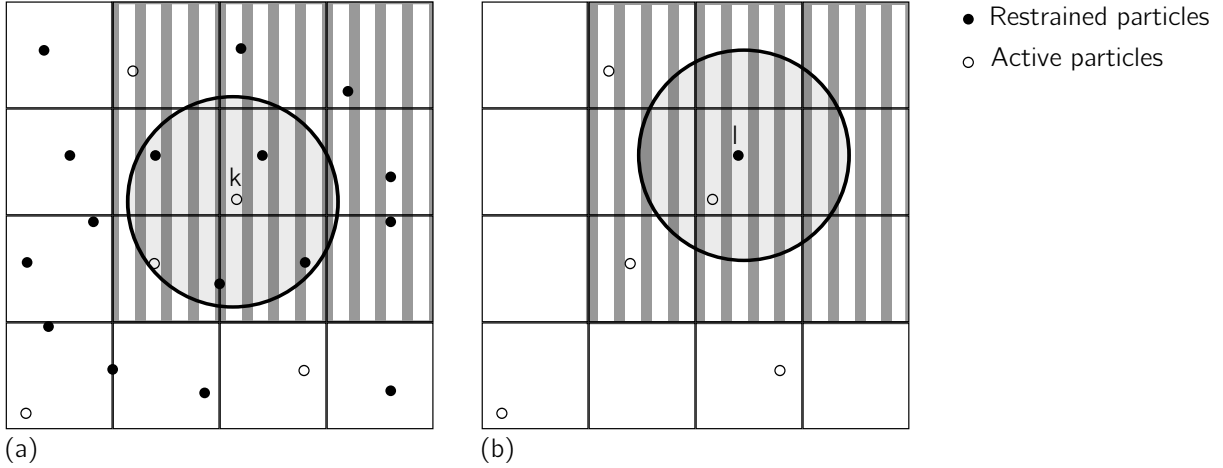


Figure 3.8: (a) Neighbor list evaluation for restrained particles, with particle k looking only at neighboring active particles. This corresponds to steps (1.2) and (1.4). (b) Same for active particles, with particle l looking at both restrained and active neighboring particles. This corresponds to step (1.6).

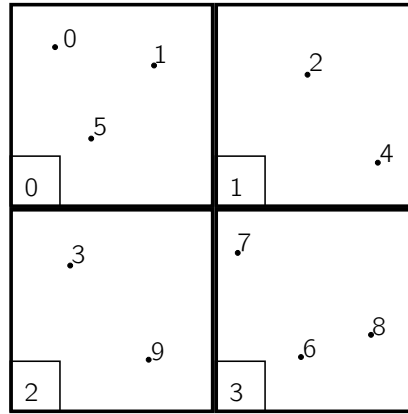


Figure 3.9: Ten particles in a four-cell grid. These are easily represented by linked cell lists.

all N_n particles in Ω . Using cell lists, we only iterate over $\rho_{act}27N_p$ (with N_p the average number of particles per cell) items in steps 1.2 and 1.4 and $(\rho_{act} + \rho_{res}^{in})27N_p$ items in step 1.6, where $\rho_{res}^{in} = \frac{|A_{res}^{in}|}{N}$, the ratio of restrained particles with an active neighbor. Nevertheless, neighbor lists might still be faster because $100(1 - \frac{4\pi}{27 \times 3}) \approx 85\%$ of distance computations is redundant in cell grids. So only when ρ_{act} is very low, which is also the range in which we see significant speedups for ARPS, can we say that cell lists are faster than neighbor lists.

To go into a little more detail: in order to keep a reliable neighbor list, we should update it every N_u steps, where N_u depends on the shell thickness δ used as in Figure 3.10. Using parameters derived from results in [18], we model according to $r_c = 2^{1/6}$ and $\delta = 0.6$, then $N_u = 12$. Mean cell occupancy is $N_p = 4.1$, the average value of N_n is $4.1 \times 27 \times 0.15 = 16.6$. Then, assuming ρ_{act} remains constant, between two neighbor list updates we compute $N_u((2(1 - \rho_{act}) + \rho_{act}N)NN_n) + NN_p * 27$ distances.

We can use the same parameters and estimate the number of particles over which a particle iterates in the cell list. With these parameters, the relation between ρ_{act} and ρ_{res}^{in} is

$$\rho_{res}^{in} = \begin{cases} 4.714\rho_{act}, & \text{if } 0 \leq \rho_{act} \leq 0.175, \\ 1 - \rho_{act}, & \text{if } 0.175 < \rho_{act} \leq 1. \end{cases} \quad (3.5)$$

Then in N_u steps we compute $N_u(2N(1 - \rho_{act})27N_p\rho_{act} + N\rho_{act}27N_p(\rho_{res}^{in} + \rho_{act}))$ distances. When we plot the relation between these numbers in Figure 3.11, we see that when 14% or less of all particles is active, the cell list approach is preferable. As we will see, we only get speedups for small values of ρ_{act} . Hence, in cases where ARPS might be used, its better to use a cell list approach.

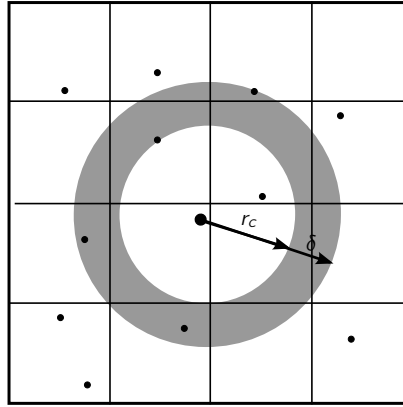


Figure 3.10: A particle's Verlet neighbor list contains all particles within cutoff distance r_c of that particle. Additionally, particles within some shell δ around the cutoff radius are included. Whenever any particle has moved more than $\delta/2$, the neighbor list needs to be updated.

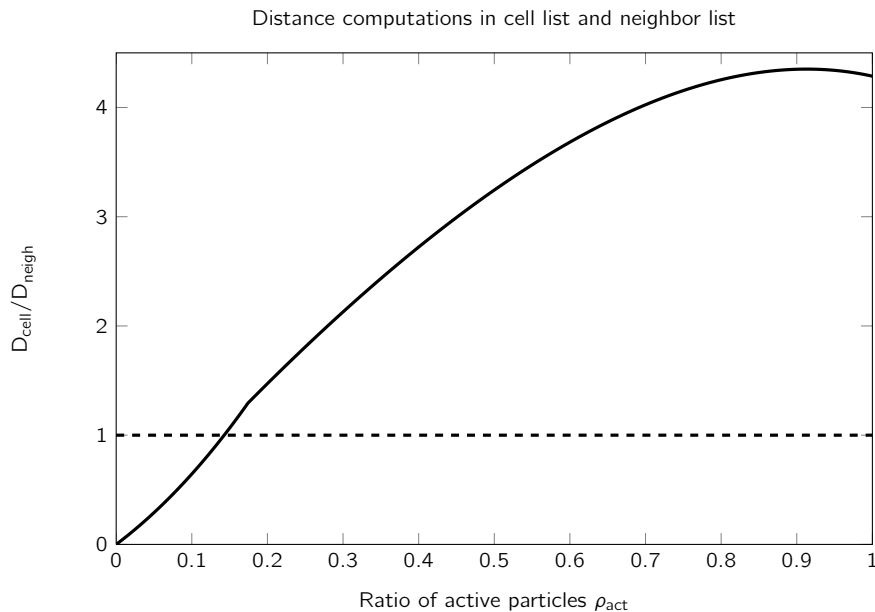


Figure 3.11: Number of distance computations D_{cell} in a cell list approach compared to that in a neighbor list approach D_{neigh} . When less than 14% of particles is active, we see that it's better to use a cell list approach to keep the number of distance computations down.

Langevin integration

In Benchmark 4, we use a Langevin thermostat to simulate in the NVT ensemble. The integration scheme is the same as in Eq. (2.5), but with the Hamiltonian $H(\mathbf{q}, \mathbf{p})$ replaced by the adaptive Hamiltonian $H_{AR}(\mathbf{q}, \mathbf{p})$. As discussed in Section 2, the Langevin thermostat relies on a random force \mathbf{f}' , sampled from a Gaussian distribution. (Pseudo)random number generation (PRNG) on the GPU remains an interesting topic, with recent progress in implementation of less (e.g. linear congruential) or more complex (e.g. Mersenne twister) random number generators [73] closely following advances in RNG research. In the Langevin momenta update step, three uniformly distributed numbers are required, which are then transformed to normally distributed numbers using a polar Box-Muller transform.

In our implementation, we initially seed every particle with its own index. Then at the beginning of the Langevin momenta update step, the state of the particle's generator x_n is read and random numbers are generated from it using a linear congruential RNG $x_{i+1} = (ax_i + c) \bmod m$. When a number is rejected in the Box-Muller transform, a new number is generated. Then at the end of the Langevin momenta update step, the new state is stored in global memory. Hence, we have N RNGs generating i.i.d. sequences of

numbers. In [74], this is called *one-PRNG-per-thread*, as opposed to *one-PRNG-for-all-threads*. We set $a = 1664525$, $c = 1013904223$ and $m = 2^{32}$. This period is rather small. However, storing states per thread might cause memory issues when more complex RNGs with much longer periods are used. In [74], global memory is avoided altogether by hashing a seed from particle-specific data such as its position and momentum and using this seed only within the kernel, discarding the RNG when it's no longer needed. This is a very interesting approach which might be used to optimize performance of ARPS on GPU.

To show that we can get statistics using ARPS on GPU, we want to obtain a radial distribution function (RDF)

$$g(r) = \lim_{dr \rightarrow 0} \frac{\rho(r)}{4\pi(N/V)r^2 dr}, \quad (3.6)$$

with $g(r)$ the value of the RDF at distance r , dr the width of a shell at distance r , (N/V) the number density. For each particle, we count the number of particles at a distance between r and $r + dr$ of the particle's position. Then we use an array of integers as a histogram. Whenever a particle finds another particle at distance r of itself, it increases the number in bin $\lfloor \frac{r}{dr} \rfloor$ by 1 using an `atomic_add` operation. Other methods are available, using e.g. individual histograms for all particles or shared histograms for groups of particles, but this global shared histogram has been found to be the most sensible choice [75].

3.4 Results

Four benchmarks were run using homogeneous particle systems modeled after liquid argon. The first three benchmarks were run in the NVE ensemble. Throughout the simulations, the number of particles N , the volume of the system V and the energy levels E were kept constant. For this we used a partitioned Euler splitting algorithm as introduced in Section 2.2. Non-bonded interactions were modeled with a 6-12 Lennard-Jones potential, with a switching function as in Figure 2.1 between $r_s = 7.5\text{\AA}$ and $r_c = 8\text{\AA}$. Other parameters were $\epsilon = 120K_b$ and $\sigma = 3.4\text{\AA}$. Benchmark 3 was run using bonded forces. Here, we used springs to model the bond between atoms, with parameters $K_r = 600$ and $r_{eq} = 3.7515\text{\AA}$. The fourth benchmark was run in the NVT ensemble, with the splitting scheme: a half step for the Langevin part of the equations, a full step for the Hamiltonian part, a half step for the Langevin part.

All benchmarks were implemented in ARPS on GPU with the before-mentioned implementation details. While it is interesting to look at the speedup of GPU algorithms with respect to CPU algorithms, this is not the main purpose of this work. It should not be surprising that the GPU implementation is faster than a CPU implementation in most cases. The exact speedup is always hard to determine, as there are some common caveats in comparing CPU and GPU performance:

- Comparing an out-of-date CPU with a state of the art GPU
- Comparing highly optimized GPU code with non-optimized CPU code
- Comparing single-thread performance on a multi-core CPU with multi-threaded GPU performance
- Insufficient reporting on memory transfer times on the GPU [76]

For these reasons, it makes more sense to compare the performance of ARPS on GPU with a basic MD on GPU implementation, called MDGPU. We do not claim that the MDGPU implementation is optimal, but then again neither is the ARPS on GPU implementation. The fact we want to verify is that ARPS can cause speedups as well on the GPU as it does on the CPU. Therefore, the implementation details for MDGPU and ARPS on GPU are very similar, e.g. both use the same inline functions for force evaluations.

The three main benchmarks were run in the NVE ensemble, even though ARPS on GPU does of course not have to be restricted to this ensemble. Simulations in the NVT ensemble should show equal speedups, as integration of the equations of motion (which is slightly more time-consuming for NVT) is only a minor part of every time step compared to evaluation of the pairwise forces. Successful NVT simulations using e.g. a Langevin thermostat on the GPU have been shown to be viable [77] and ARPS has been shown to speed up NVT as well as NVE simulations on the CPU [1]. To show that we can indeed obtain statistics in the NVT ensemble in ARPS on GPU, we have run a fourth benchmark using a Langevin thermostat to obtain radial distribution functions of argon.

The NVE benchmarks were run on AMD's HD7970 graphics card, which has 32 compute units with 64 processors each, for a total of 2048 processors and 3GB of memory. This device was hosted by Computer

1 (one Intel Core2 quad-core 2.40 GHz processor with 4GB of RAM on a Windows Vista 32-bit operating system). The NVT benchmark was run on a NVIDIA GTS450 graphics card, which features 192 CUDA cores and 1GB of memory. This card was placed in Computer 2 (one AMD Phenom(tm) II X4 965 Processor quad-core processor with 8GB of RAM, on a Ubuntu 12.10 64-bit operating system).

Benchmark 1: Particle NVE argon Liquid

We have simulated periodic 3-D boxes with either 21952, 125,000 or 343,000 argon particles. Interactions were modeled with the 12-6 Lennard-Jones potential. Particle mass was 39.95 g/mol. In [1] this test was run with 21952 particles to show that the adaptive energy (the value of the adaptive Hamiltonian H_{AR}) is stable over long simulations. Here, we want to replicate these results. Box size is 99.3048Å. We run ARPS on GPU with 5,000,000 time steps of 0.488fs for a total of 2.44ns. We use two sets of thresholds ($\epsilon_r = 1.5; \epsilon_f = 2.5$) and ($\epsilon_r = 9; \epsilon_f = 10$).

Accuracy and stability

In Figure 3.12a we see how the the total energy shows a drift throughout the simulation for ($\epsilon_r = 1.5; \epsilon_f = 2.5$). In the original CPU implementation, this drift was not present. In [68] and [69] the issue of accuracy and reproducibility of MD simulations on the GPU is addressed. This is not a GPU-specific problem, but a problem for parallel applications in general, as is pointed out in [78]. The authors of [69] note that even though operations on double precision floating point numbers are less prone to errors than those on single precision numbers, drifts still occur in long simulations as small errors accumulate rapidly. Often these errors are caused by rounding errors when accumulating distributed results, e.g. across work groups.

In Figure 3.12a we also see how the energy plot for the higher thresholds seems to contain more noise. This is due to the sudden release of momenta when a particle becomes active in a simulation with high thresholds. In ARPS, the amount of momenta used in the position update depends on a switch function which runs from 0 to 1 between ϵ_r and ϵ_f . If the gap between ϵ_r and ϵ_f is small, momentum is released rapidly and this might cause inaccurate position updates on the GPU. For ($\epsilon_r = 9; \epsilon_f = 10$) the gap is relatively smaller than for ($\epsilon_r = 1.5; \epsilon_f = 2.5$), hence the energy plot is noisier. If we broaden the gap to ($\epsilon_r = 9; \epsilon_f = 12$), the variance in H_{AR} is reduced from 0.3964 kcal/mol to 0.0608 kcal/mol. As a reference, the variance in a 5,000,000 step full dynamics GPU simulation is 0.0092 kcal/mol.

Nevertheless, these results show a strong overlap with those obtained on the CPU. Furthermore, the number of active particles stays stable throughout the simulation (4120 ± 58 for ($\epsilon_r = 1.5; \epsilon = 2.5$), 435 ± 21 for ($\epsilon_r = 9; \epsilon = 10$)).

Reproducibility

In this implementation of MD on GPU, and hence also in ARPS on GPU, different runs are not identical as linked cell lists do not have the exact same order in every simulation. The order of the elements in a linked cell list depends on the way the atomic swap operations are scheduled to avoid race conditions and this happens ad hoc. Therefore, if we compare two independent simulations, the forces on particle i in step n are most likely computed in a different order. In Figure 3.12b we see how rounding errors accumulate to a small variance between six different runs of the same simulation, even though the total energy in an individual run is very stable (Figure 3.12a, red line). In cases where reproducibility is an issue, runs might be made more similar by using alternative cell list or neighbor list approaches such as in [16] or [18].

Computational performance

Figure 3.13 shows the percentage of active particles ρ_{act} as a function of ϵ_r , as determined experimentally. Data for 21952, 125,000 and 343,000 particle systems is aggregated and shows a very strong overlap. Not surprisingly, this function does not depend on the size of the system, but rather on other parameters such as density, potential function and initial energy.

In Figure 3.14 we see how the number of forces computed in benchmarks matches the predicted number of computed forces. The black line in Figure 3.14 is equal to the black line in Figure 3.6. The colored scatter plots correspond to several different values for (ϵ_r, ϵ_f). Values on the y -axis are scaled

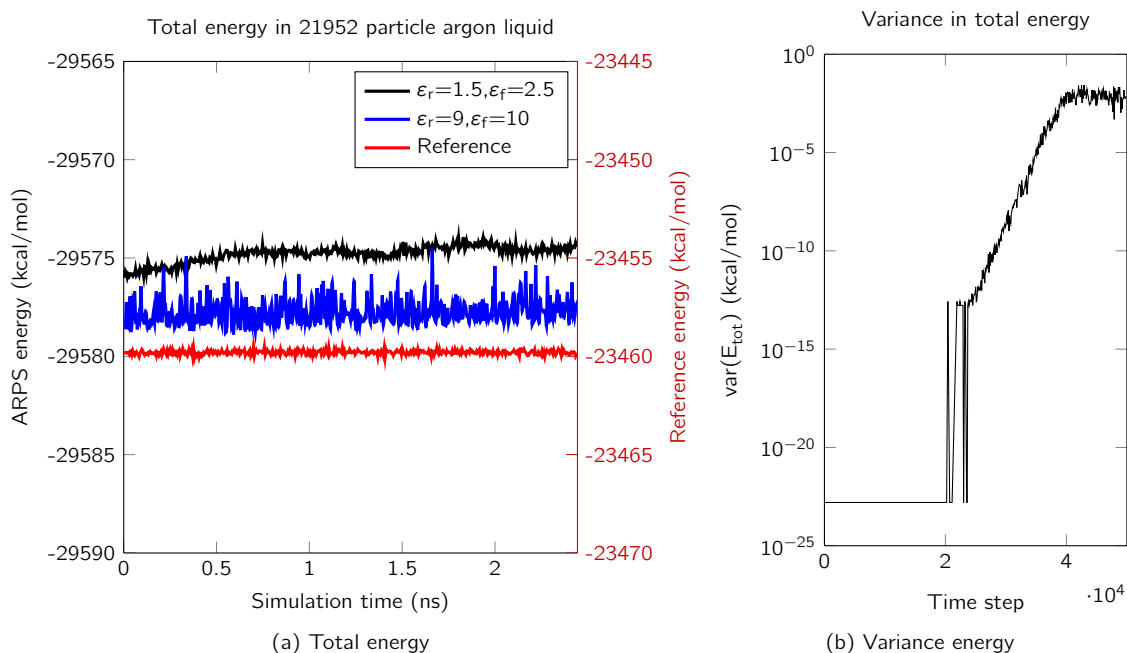


Figure 3.12: (a) Total adaptive energy $H_{AR}(\mathbf{q}, \mathbf{p})$ in system for two sets of thresholds and reference energy $H(\mathbf{q}, \mathbf{p})$ for a full dynamics simulation. Here, 5,000,000 time steps were run for a total 2.4ns simulated time. (b) Variance in total energy E_{tot} for 6 runs of a 21952 full dynamics simulation.

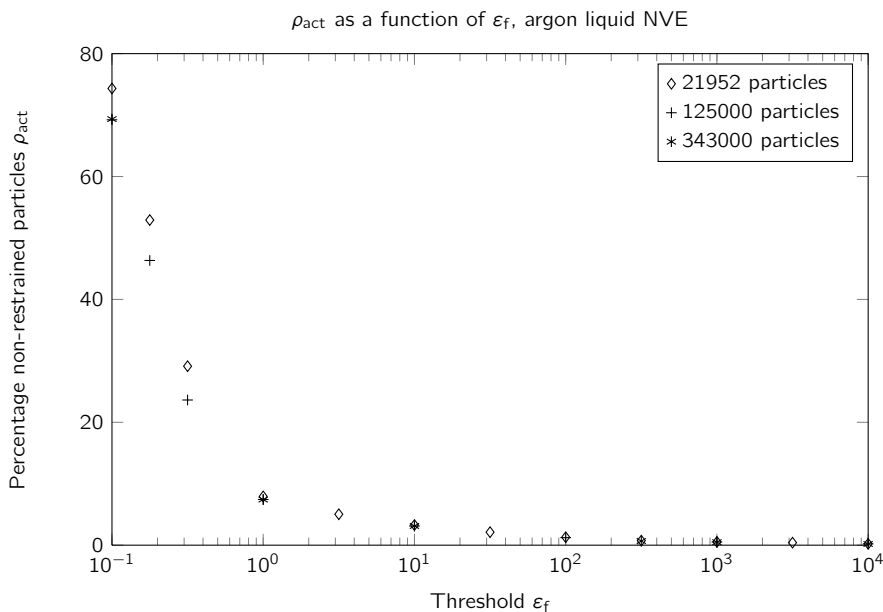


Figure 3.13: Ratio of active particles ρ_{act} as a function of threshold ϵ_f in argon liquid NVE. Aggregated data for 21952, 125000 and 343000 particle systems. In all cases, $\epsilon_f = 1.1\epsilon_r$.

according to the full dynamics condition. We verify that when $\rho_{act} > 0.5$, it is actually better to use a full dynamics approach because ARPS on GPU only adds force computations.

We have used OpenCL's profiling options to gain insight in the distribution of the computing time over the different kernels. From Section 3.3 we know that there are ten kernels. In Figure 3.15 we see how the average wall clock time of one iteration is distributed over these kernels. As we might expect, the force evaluation kernels consume by far the most time. Also, we see that subtracting and adding forces to the restrained particles take almost identical amounts of time. Furthermore, we see that when the threshold increases and the number of active particles decreases, a relatively larger share of the time is

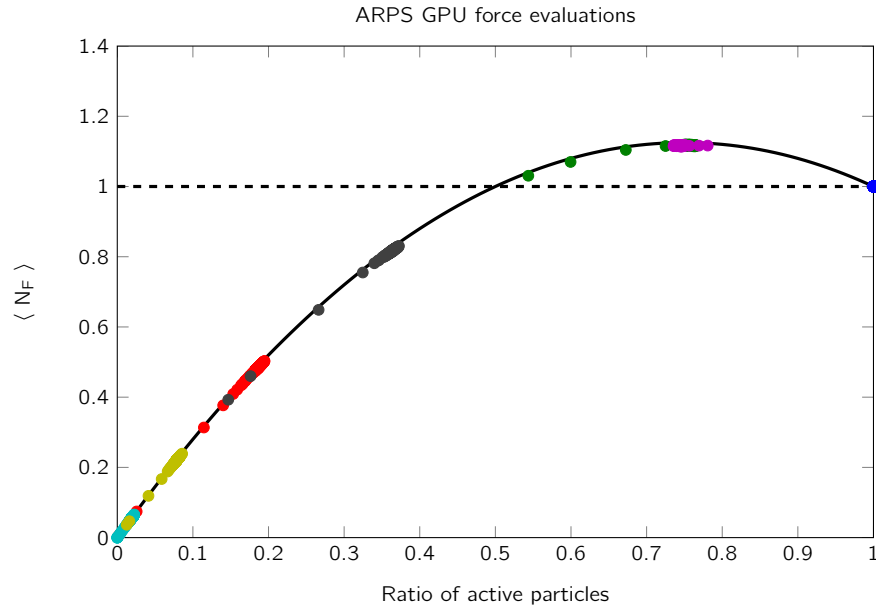


Figure 3.14: Number of forces computed per iteration for several values of ϵ_f and their corresponding levels of activity.

devoted to force computation on the active particles. From Section 3.1 we know that theoretically steps 1.2 and 1.4 take $\rho_{act} N N_n (1 - \rho_{act})$ force evaluations each and step 1.6 takes $\rho_{act} N N_n$ force evaluations. Hence, we can say that steps 1.2 and 1.4 take $1 - \rho_{act}$ times the time that step 1.6 takes: step 1.6 always takes more time. We see this reflected in Figure 3.15. For $\epsilon_r = 0.1$ this would theoretically be 0.3 while experimentally it is 0.35, for $\epsilon_r = 0.18$ it would be 0.54 while experimentally it is 0.58.

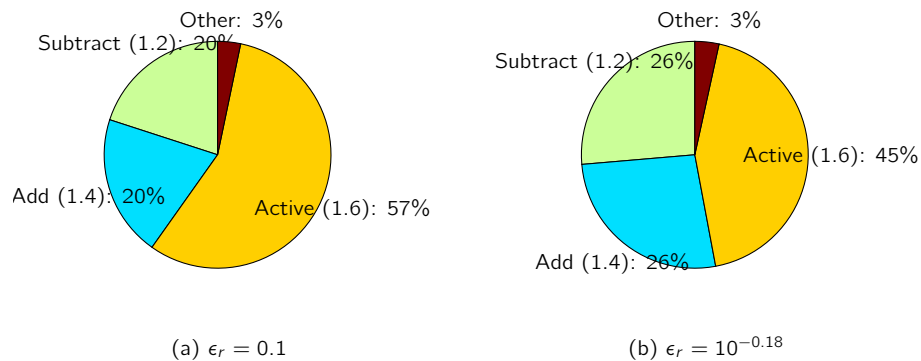


Figure 3.15: Split up of average iteration computation time in 343,000 particle argon liquid NVE ensemble for threshold (a) $\epsilon_r = 0.1$, $\rho_{act} = 0.6936$ and threshold (b) $\epsilon_r = 0.18$, $\rho_{act} = 0.4636$.

In Figure 3.16 we compare the theoretical speedup with the actual experimental speedup we reach for a system of 125,000 particles. The ARPS implementation reaches a factor 25 speedup over the full dynamics CPU implementation in [1]. We see that we achieve a speedup of approximately 5 with respect to the full dynamics GPU implementation for a very low number of active particles. This falls short of the 20+ speedup we could theoretically expect. However, in Figure 3.14 we have seen that the number of computed forces matches the predicted number of forces almost perfectly. Hence, computing time does not appear to scale linearly with the number of forces, as it does in a single-threaded CPU implementation. This phenomenon has been observed in a range of other papers on MD in GPU. Usually claims about GPGPU speedups are based on large systems. Results in [72, 18, 16, 15, 79] independently show how GPU performance increases with system size, only to reach a maximum when the GPU is satisfied, e.g. at around 200000 particles [72] or 50000 particles [15]. When ρ_{act} is very small, the

number of active particles is very small and steps 1.2, 1.4 and 1.6 of the algorithm basically represent three force evaluation steps with very small N .

In Figure 3.17 we see how the required wall clock time for the `update_active_forces` kernel does not increase linearly with the number of active particles, but rather as a step function, which increases every ± 8000 particles and which we show as a black line. In fact, this number seems to correspond with 8192, the number of stream processors in the HD7970 (32) multiplied with the size of workgroups we used (256). The increase in computing time at both jumps seems to be approximately the same. Again, this makes sense if we look at the problem at hand: irrespective of the number of active particles, every work item corresponding to an active particle needs to process N_n force computations, one for each neighbor of the active particle, and this time is fixed. Hence we might say that the GPU launches batches of 8192 work items, and the number of batches depends on the number of active particles. There is no difference between 100 active particles and 8000 active particles in the evaluation time of the `update_active_forces` kernel. For the other two kernels, this is different, as the time one thread takes depends on the number of active neighbor particles of an active particle. This is also reflected in e.g. the first three green/blue data points in Figure 3.17. Interestingly, the time for these kernels decreases as the number of active particles increases. This is an indication that these kernels are also launched in batches of 8192 work items. As this particular behavior is not present in the theoretical model which we developed in Section 3.1, the speedup we see in Figure 3.16 is lower than expected. In Figure 3.16 we already see the difference between a 21952 particle and a 125,000 particle system. We might expect that the experimental results better match the theoretical prediction as the number of particles increases further.

If we change the model so that it contains the step function in Figure 3.17, we see that the speedup depends on the number of particles (Figure 3.18). Our expectations are confirmed when we run some experiments with 1,000,000 particles. The activity levels and corresponding speedups are shown in red in Figure 3.16. Indeed, as the number of particles increases, the experiment results better match the predicted speedup.

If we make a quick comparison between performance in [40] and this GPU approach, we see that the full dynamics GPU program has a speedup of approximately 40 over the full dynamics CPU implementation. Thence, for low levels of activity, ARPS on GPU might be more than 500 times faster than the standard full dynamics CPU implementation. In the ± 30 minutes it takes to let the full dynamics CPU implementation simulate 10,000 steps for 21,952 particles, we can also run 100,000 steps with ARPS on GPU for a one million particle system and appropriate thresholds.

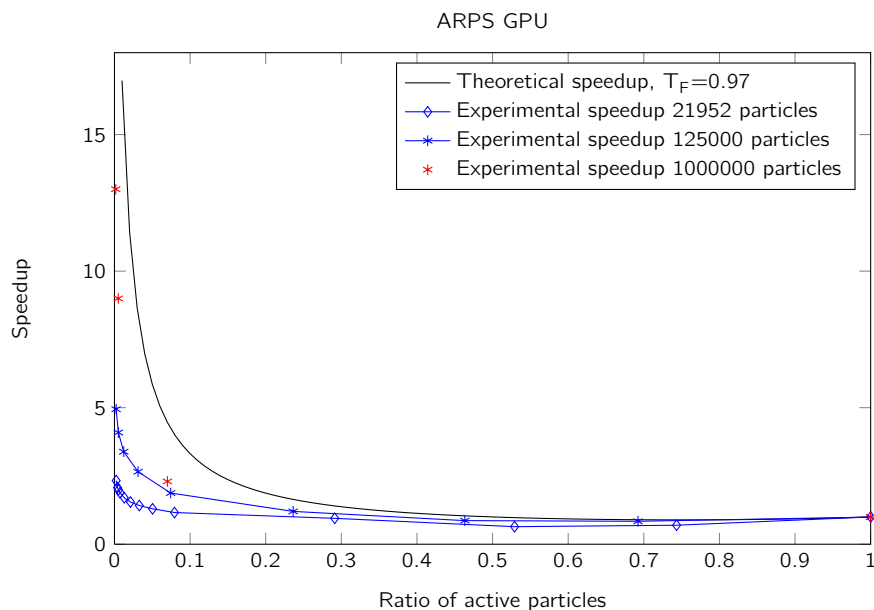


Figure 3.16: Experimental speedup for systems of 21952, 125,000 and 1,000,000 particles, compared to theoretical speedup according to the model developed in Section 3.1. Experimental speedup is lower than predicted for small values of ρ_{act} due to hardware constraints. Here, $T_F = 0.97$ and $T_O = 0.03$, as in Figure 3.15.

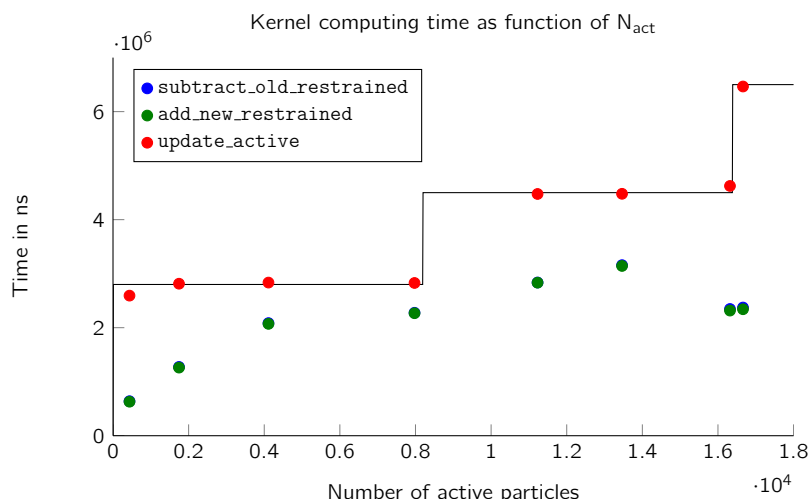


Figure 3.17: Computation time for the force evaluation kernels does not increase linearly with the number of active particles, but rather as a step function. This is reflected in the speedup for small numbers of active particles.

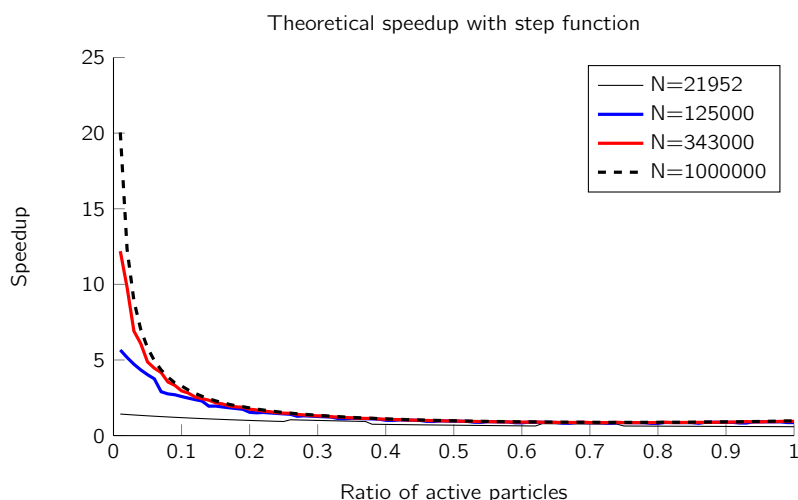


Figure 3.18: Theoretical speedup with force computation time modeled as step function. As the system size increases, the curve becomes smoother and matches the curve in Figure 3.6 better.

Benchmark 2: Collision cascade

In [1] and [40], several collision cascade simulations are examined. It is shown that even when only a very small percentage of the particles is active, ARPS is able to preserve features of the simulation very well. The simulation consists of particles in an equilibrated 2-D or 3-D grid and one particle which is launched at this grid with some initial momentum. In [1], a 5930 particle 2-D system is used to demonstrate the effects of ARPS. Figure 3.19 shows three snapshots from a collision cascade in the 5930 particle system. In the previous benchmark we have seen that for small numbers of particles, ARPS may have little or even negative effects on computation times. Therefore, we concentrate on an equilibrated system of 138,916 particles in 3-D at which we launch one particle with some momenta. The time step size is 0.0000488fs and 100,000 steps are simulated for a total simulated time of 4.88fs. To verify the ARPS on GPU implementation, we have compared the final configuration of a 5930 particle 2-D system with that obtained on the CPU and found that they are identical.

We use the same potential function and parameters as in the previous benchmark, with the exception of particle mass, which is now 1 g/mol. In Figure 3.20 we see that the number of active particles is not constant like it is in the argon liquid benchmark, but increases as the simulation runs and more particles

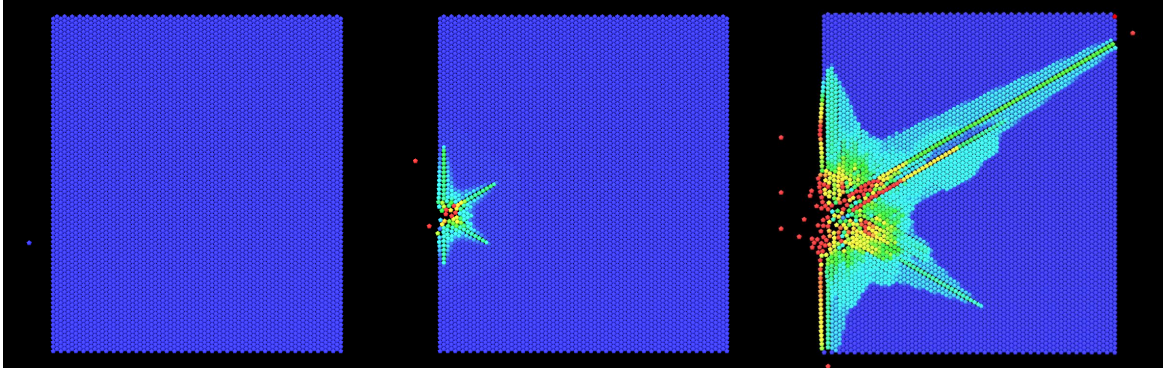


Figure 3.19: 5930 particle 2-D collision cascade after 0, 700 and 7000 time steps. One particle is launched at an equilibrated lattice and causes a shock, which is propagated through the lattice.

are affected by the propagating collision wave. For high thresholds ($\epsilon_r = 99, \epsilon_f = 100$) at most 0.4% of all particles is active. As we might expect, this is reflected in reduced time for force evaluation. In Figure 3.21 we see how much faster ARPS on GPU is with these thresholds, per time step and compared with the full dynamics algorithm. We see that at the start of the simulation, when almost no particles are active, the speedup is almost 20, then rapidly decreases to around 4 and then slowly increases as the number of active particles decreases again. For the lower threshold set ($\epsilon_r = 0.01, \epsilon_f = 0.02$) the number of active particles increases throughout the simulation and hence the speedup per time step only decreases. Overall, the low threshold simulation is 3.05 (vs. 10 on CPU) times faster and the high threshold simulation is 5.16 (vs. 25 on CPU) times faster than the full dynamics simulation.

The difference between this benchmark and the first benchmark is that activity is very localized. At the start of the run, we get impressive speedups. This is not only due to the fact that there is a small number of active particles, but also because all these active particles are in the same area. Hence, there will be a lot of work groups in the `subtract_restrained_forces` and `add_new_restrained_forces` kernels which will not contain particles with active neighbors. Hence, these kernels finish very quickly. Even near the end of the simulation, only a small set of restrained particles has an active neighbor (Figure 3.22). This in contrast to the argon liquid benchmark, where activity is spread more evenly over the domain.

Percentage of active particles in 138,916 particle 3-D collision cascade for different thresholds

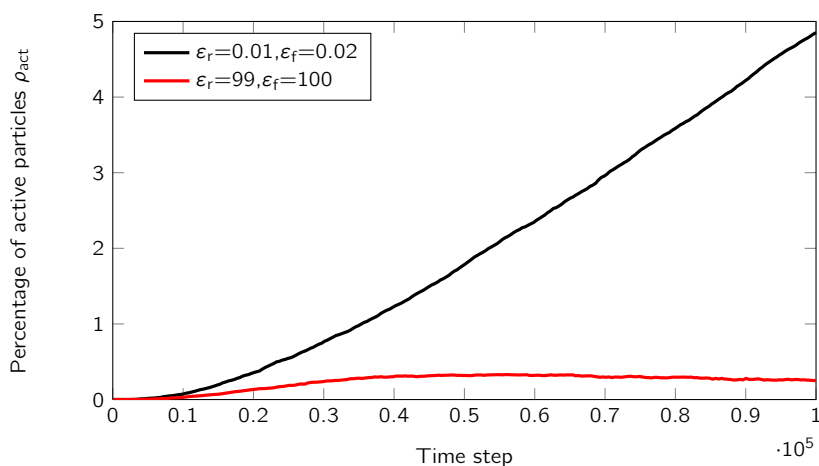


Figure 3.20: Percentage of active particles for two threshold sets in a 138,916 particle simulation. For the low threshold, the number of active particles increases throughout the simulation. For the high threshold, the number of active particles peaks after some 50,000 iterations, then decreases.

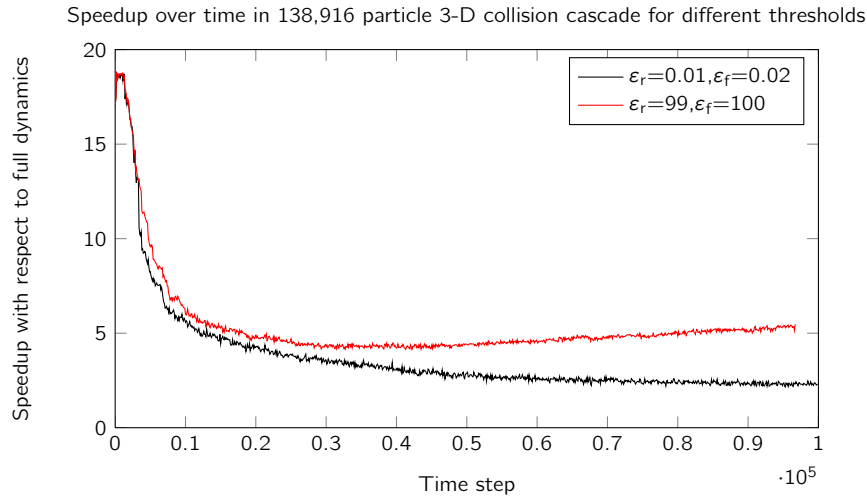


Figure 3.21: Speedup with respect to full dynamics GPU simulation. As the number of active particles increases (Figure 3.20), speedup decreases. Then when the number of active particles decreases, speedup increases again. Run with 138,916 particles, $\varepsilon_r = 99, \varepsilon_f = 100$.

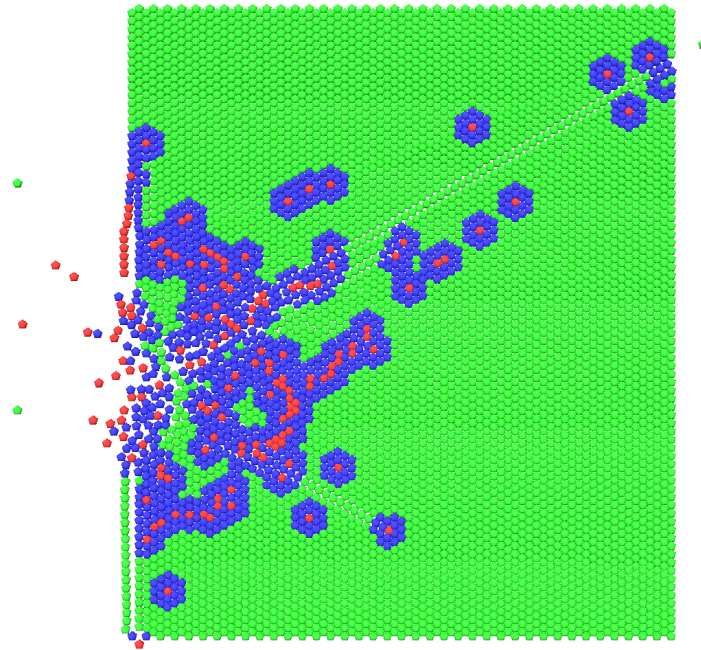


Figure 3.22: Three sets of particles: active particles A_{act} in red, restrained particles within cutoff of active particles A_{res}^{in} in blue and other restrained particles A_{res}^{out} in green.

Benchmark 3: Bonded Interactions

Benchmarks 1 and 2 model non-bonded short-range interactions. To see if ARPS on GPU performs equally well on bonded interactions, we run a third benchmark with a 2-D system of bonded particles, very similar to the collision cascade. A particle connected by spring-like bonds to an equilibrated lattice is pulled back and released. This causes a wave propagation. Computing only bonded forces means that we already now which particles are neighbors, as in Figure 3.23, and these neighbor lists do not require updating throughout the simulation. This greatly reduces the complexity of the algorithm, though the core is still the same. Again, we might expect that for very small numbers of particles, ARPS is not beneficial. Therefore, we run this benchmark with 200,000 particles in 2-D.

We notice that the speedup due to ARPS is negligible, even when we set the thresholds very high. The reason for this is that a particle now only has 6 neighbors, approximately 8 times less than in the

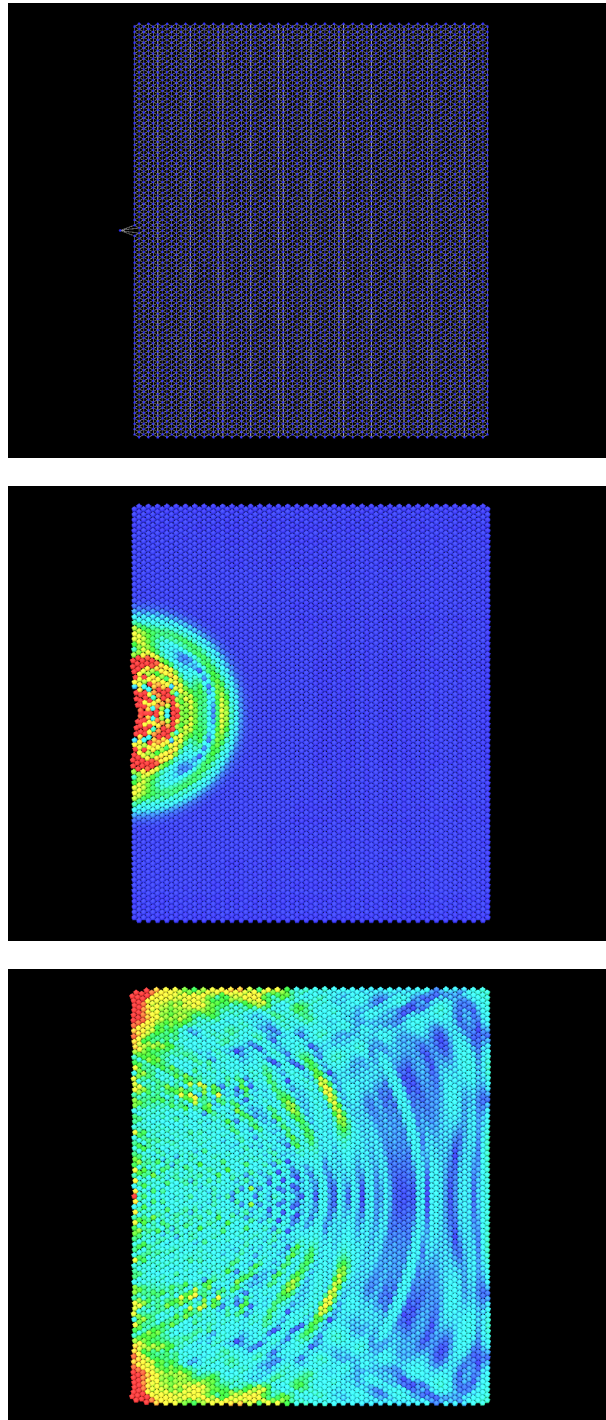


Figure 3.23: Bonded system after 0, 700 and 3000 time steps (bonds omitted in latter two for clarity).

non-bonded case. Therefore a work item is able to compute the forces on its corresponding particle 8 times faster. Relatively, the overhead kernels take a lot of computing time. When we look at T_F , the share that the actual force computation takes, we see that in the full dynamics implementation, approximately 68% of each iteration is spent on force computing. In ARPS on GPU, this depends on the number of active particles, as we see in Figure 3.24. With the simple spring model which we use for bonded forces, more than 50% of the particles has to be active to let 40% of computing time be occupied by force computations.

If we increase the cost of the force computation kernels by artificially making the force function more complex, as in Listing 3.2, we see that the time spent on force computation increases in both full dynamics and ARPS on GPU. In full dynamics, it becomes around 95% and in ARPS on GPU it depends on the number of active particles as in Figure 3.24. Then the number of forces determines the running time of the algorithm and we actually get a speedup due to ARPS, like in Figure 3.25. This shows that ARPS on GPU is capable of reducing evaluation times for bonded as well as non-bonded particle interactions, as long as there are enough neighbors or more complex force fields are used. These figures also very clearly show the stepping behavior which was observed in Figure 3.17.

```

1  int complexity = 100 ;
2  int neighbor = neighborlist[j] ;
   double4 r = positions[neighbor]-myposition ;
4  double rnorm ;
   for(int i = 0 ; i < complexity ; i++)
6     rnorm = distance(positions[neigh].xyz, myposition) ;
   double4 force += kSpr*(rnorm-d0Spr)*r/rnorm ;

```

Listing 3.2: OpenCL bonded force computation. The `complexity` parameter artificially makes the force computation more time consuming.

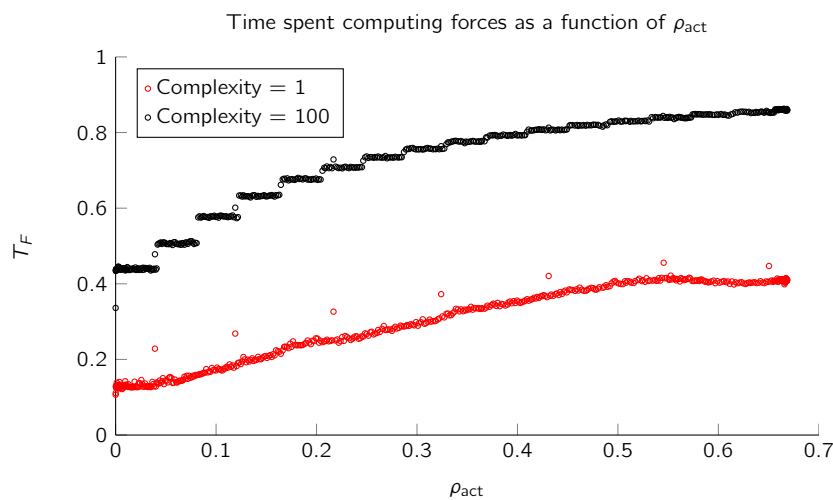


Figure 3.24: Computing time per iteration spent on force computation, as a function of the ratio of active particles ρ_{act} . Line plotted for a very simple force computation with `complexity=1` as in Listing 3.2 and a more complex computation with `complexity=100`.

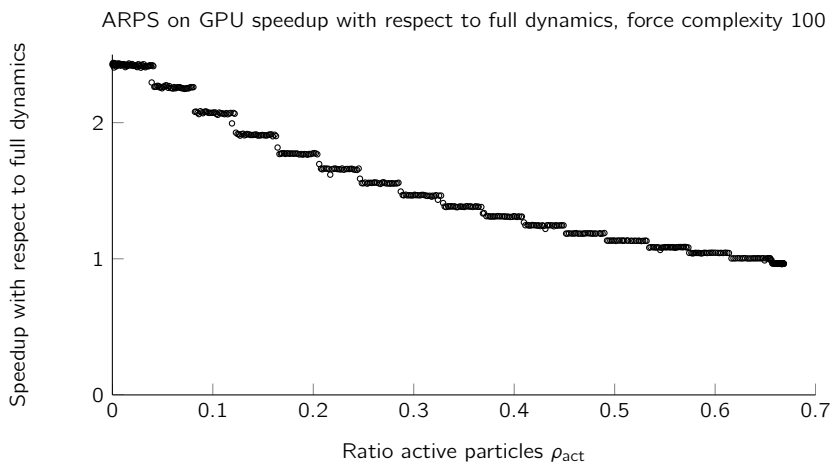


Figure 3.25: Speedup of ARPS on GPU with respect to full dynamics implementation. Force computation is made more complex and now we see a speedup for low values of ρ_{act} .

Benchmark 4: Obtaining statistics in the NVT ensemble

To show that we can obtain statistical properties using ARPS on the GPU as well as on the CPU, we run a 343 argon particle NVT simulation in a $25.56 \times 25.56 \times 25.56 \text{ \AA}$ 3-D box. The mass of each particle is 39.95 g/mol. We use the following Langevin thermostat: a half step for the Langevin step of the equation, one step for the Hamiltonian part and a half step for the Langevin step. Because this scheme is implicit, we use a fixed-point method to solve the non-linear equations for momenta. On average, 3 iterations were needed for the method to converge. We use a Langevin friction coefficient $\gamma = 1$ and set temperature to $T = 94.4$. All parameters are equal to those in [1]. We run 150.000 0.488fs steps for a total of 73.2ps simulated time. After 50.000 equilibration steps, we start measuring the radial distribution function.

We obtain radial distribution functions (RDFs) of argon in MDGPU and ARPS on GPU. Because we use periodic boundary conditions, the expected number of particles $\rho(r)$ at some distance r from a particle does not increase quadratically, but drops at half the width of the box, $L/2 = 12.78 \text{ \AA}$. In the obtained RDF diagrams, we apply a cutoff at this point. Figure 3.26 shows the obtained radial distribution functions. We see that the RDFs coincide, even though in ARPS only $3.3 \pm 1.2\%$ of particles is active on average. This is similar to the result obtained in [1]. In terms of computational speedup, results obtained using NVT are very similar to those in NVE: the only difference is in the more complex integration step, which takes a comparable amount of time in both full dynamics and ARPS implementations.

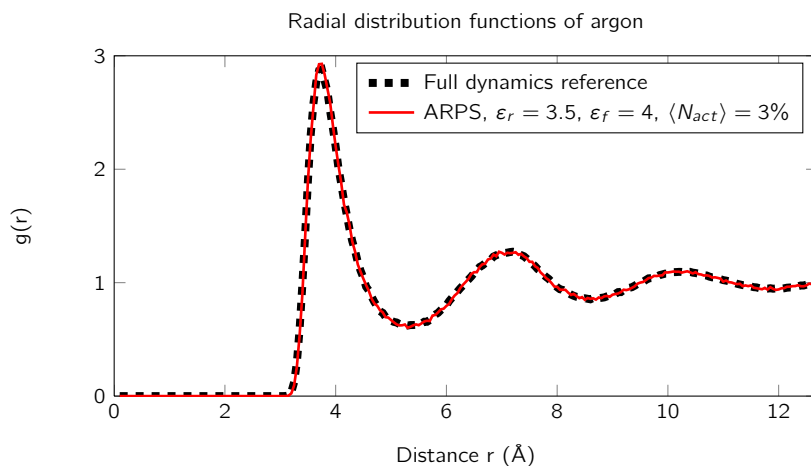


Figure 3.26: Radial distribution functions for argon, obtained on the GPU with a full dynamics implementation and an ARPS implementation. The obtained functions are very similar, even though only 3% of the particles is active in the ARPS simulation.

3.5 Conclusion and discussion

ARPS has shown to be a valuable addition to the arsenal of coarse-graining and other adaptive molecular dynamics algorithms on the CPU. Here we have shown that ARPS can reduce computing times on the GPU too, by explicitly reducing the number of forces to be computed. We have presented a data-parallel implementation using linked cell lists which shows reduced computing times for both bonded and non-bonded pair interactions depending on the number of active particles. This implementation was compared with a very similar non-adaptive full dynamics implementation.

On the GPU, ARPS might reach speedups comparable to those on the CPU and beyond, but there are some caveats. We have seen how the speedup not only depends on the ratio of active particles ρ_{act} , but also on the number of particles N . We get better speedups in the low ρ_{act} range when N is bigger. For a one million particle NVE simulation, we were able to reach a speedup of 12 over the standard GPU implementation, with only a small fraction of particles active. Nevertheless, we have shown that on the GPU we can obtain reliable statistics using ARPS in both the NVE and NVT ensemble, even when only a very small fraction of particles is active. ARPS on the GPU functions best if force computation is a very significant part of each iteration. In the case where there are only a few forces to be computed, e.g. in Benchmark 3 with just six neighbors per particle, overhead caused by ARPS' extra kernels might reduce efficiency.

ARPS is able to enhance the speedup obtained by GPU-powered MD simulations by some extra factor. This might be a huge benefit to work in computational biology, chemistry and physics. Compared to a very similar full dynamics CPU implementation, ARPS on GPU performed up to 500 times faster. There still are some caveats in terms of stability and reproducibility. This is a problem across MD simulations on the GPU. In future implementations, more care could be taken of this, based on recently published solutions.

Interactive Manipulation of Large Macromolecules

Contents

4.1	Introduction	43
4.2	Related work	44
	Skinning	44
4.3	Methods	47
	Quadratic program	48
	Laplacian matrix	48
	Mass matrix	49
	Shape preservation	49
4.4	Implementation	50
4.5	Results	50
	Problem 1: 1-dimensional chain	50
	Problem 2: 2-dimensional ring	51
	Problem 3: Graphene	52
	Problem 4: Molecule manipulation	53
	Problem 5: Simulation manipulation	53
4.6	Conclusion and discussion	54

4.1 Introduction

In Chapters 1 and 2 we have introduced software tools like YASARA [31], PYMOL [32], Abalone [44], AMBER [36], GROMACS [47] and LAMMPS [49, 50]. Other noticeable tools are MOLDEN [80], NAMD [51] and VMD [81]. These tools all perform a subset of the following tasks: visualization, MD simulation and (protein) modeling. Some frameworks have been presented which perform all three of these tasks. In [82] interactive molecular dynamics (IMD) are introduced, where NAMD provides a back-end running MD simulations, VMD visualizes the simulation and the user is able to apply forces to atoms through a haptic device. These forces are then scaled and integrated into the equations of motion. Over the past ten years, IMD has been further developed to employ GPU hardware, virtual reality and commodity haptic devices [83]. Quite similar to IMD, but different in the sense that it's aimed more specifically at proteins, is ProteinShop [33].

The focus of the work here presented is on developing an algorithm which is capable of manipulating very large molecules (not necessarily proteins). As with ProteinShop [33], the utility will be twofold: the user should be able to generate starting configurations for MD simulations or energy minimization and the user should be able to intervene in running simulations or energy minimizations by applying transformations on particles or groups of particles. We want the newly developed method to take into

account forces at an atomic level. The user should be able to apply rigid body transformations to individual atoms or secondary structures in a protein.

As will be described in the next section, the method proposed in this chapter leans heavily on computer graphics methods for deformation. We find Laplacian energy minimizing functions on molecular graphs based on inter-atomic forces and then apply a weighted combination of rigid body transformations to each particle. This method is different from IMD and ProteinShop in the sense that atomic interactions are approximated more coarsely and updated less frequently. This does however allow us to sculpt large molecules interactively at an acceptable loss in accuracy.

4.2 Related work

Realistic deformation of objects, i.e. finding a map from a given shape to a new one, has been a challenge in computer graphics for decades. With games and animated movies getting more and more complex, there is always a need for efficient algorithms which are able to realistically deform a model. Furthermore, there is an increasing need for high-quality interactive editing deformation of shapes, e.g. for CAD design and medical simulation tools. In theory, it is possible to use existing techniques for physical simulation such as finite element methods to very accurately deform a model. For interactive deformation however, finite element methods are much too computationally demanding and a trade off is often made between accuracy and speed.

One class of algorithms deforms a model by deforming its surface, which might be represented by a set of shells. For solid models, this is less appropriate, as mere surface deformation might lead to a loss of volume. Therefore, volume-preserving methods have been developed, such as the adaptive space deformation method proposed in [84], where space is discretized into a set of three-dimensional arbitrarily shaped cells, time is discretized in time steps and a system of equations is solved at each time step to minimize elastic energy of the system and find rigid transformations, which are then applied to the cells. Though this method excels in shape deformation with volume preservation, it loses interactivity for larger models because the time needed to solve the system of equations corresponds to the system size.

Another challenge is to model deformation of objects with varying stiffness. In modeling a protein or macromolecule, we expect parts of it to be more rigid than others. For example, we expect the bonds on the protein backbone to be more rigid than the non-bonded Vanderwaals interactions between atoms in adjacent side chains. The method proposed in [85] represents varying stiffness by so-called *composite elements* and yields physically plausible results, though it's aimed more at simulations and hence does not allow the user to interactively manipulate an object. Alternative methods such as [86] or [87] do not rely on a mesh or grid, but rather on points. These points represent physical volume elements (phexels) or surface elements (surfels) with different volumes, areas and mass. Setting the number of points relatively low allows interactive manipulation, but allows less possibilities in terms of locally varying stiffness.

The disadvantage of all these methods is that they require implicit solving of a system of equations at runtime. Typically, in deformation methods a distinction is made between **bind time**, where the system is initialized, and **pose time**, where the actual deformation occurs [2]. In that respect, methods like [84, 85] have relatively small bind time but large pose time. In fact, we would prefer explicit computation of new positions within the object at pose time, with some acceptable bind time. In [88] Green coordinates for cage deformations are introduced, where an object is captured within a (partial) grid and the position of a point within the object is based on the coordinates of the cage vertices and faces. Alternatively, mean value coordinate or harmonic coordinate methods do not consider face normals and offer worse shape preservation, especially of surface details [89, 90]. Because in these methods for every point within the object, we only compute a linear combination of coordinates, this is a very fast deformation algorithm which is easily moved to the GPU for even better performance.

Skinning

Cage-based methods like the ones in [88, 89, 90] are very close to the concept of linear blend skinning (LBS). This method is known under a variety of names, e.g. enveloping or skeleton-subspace deformation [91]. The algorithm is unpublished, but ubiquitous in computer graphics. Though this method is known to produce some notorious artifacts, it's popular because of its simplicity and the fact that it's easy to implement on a GPU. Given a shape \mathcal{S} , a set of handles H is defined by the user as in Figure 4.1a. These handles can be either bones, point or cage vertices. In our implementation, they will be graph

vertices. On manipulation, transformations are applied to these handles. Every handle H_j has its own weight function w_j on the shape \mathcal{S} . In Figure 4.1b we show an example weight function w_3 for handle H_3 . The domain of this weight function is not necessarily restricted to \mathcal{S} . We define $\Omega \subset \mathbb{R}^2$ or \mathbb{R}^3 as the volumetric domain enclosed by the union of the given shape \mathcal{S} and cage controls. For every point $\mathbf{p} \in \Omega$, a linear combination of handle transformations is computed based on the weight functions: $T_{\mathbf{p}} = \sum_{j=1}^m w_j(\mathbf{p})T_j$. Then, this transformation is applied to \mathbf{p} to obtain its new position \mathbf{p}' .

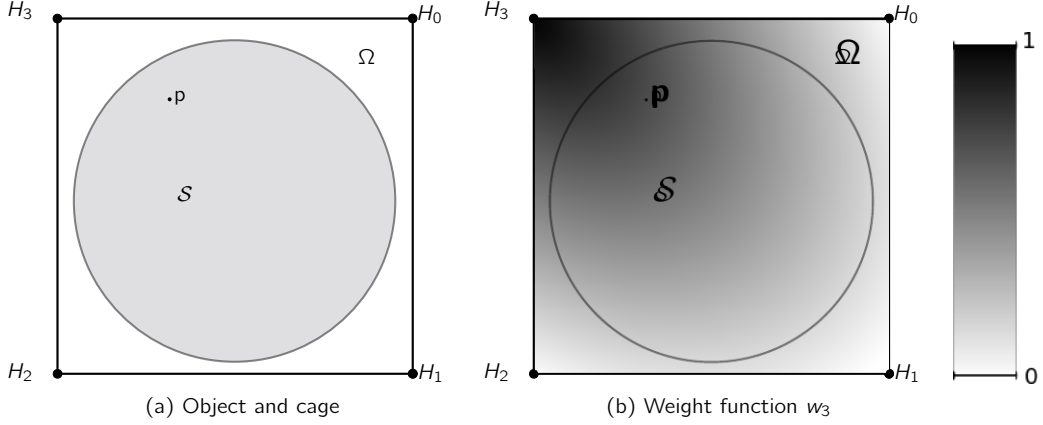


Figure 4.1: (a) The circular shape \mathcal{S} is enclosed by a rectangular cage, defined by the set of handles $\{H_0, H_1, H_2, H_3\}$. This cage encloses the domain Ω , including some point $\mathbf{p} \in \mathcal{S} \subset \Omega$. (b) Weight function w_3 of handle H_3 on Ω , running from 1 at H_3 to 0 at the other handles. Here, $0 < w_3(\mathbf{p}) < 1$.

In [2], a set of weight functions over \mathcal{S} is found, after which these weights are used for linear blend skinning. The authors call these bounded biharmonic weights (BBW), as a biharmonic boundary problem is solved over Ω . These weights have some attractive properties and allow for cage-based, skeleton-based and point-based deformation. Also, areas of the object can be rigidified during deformation. Here, we propose to extend this method to deformation of macromolecules. In [2], weights are found as minimizers of the problem

$$\begin{aligned}
 & \arg \min_{w_j, j=1, \dots, m} \sum_{j=1}^m \frac{1}{2} \|\nabla w_j\|^2 dV \\
 & \text{subject to: } w_j|_{H_k} = \delta_{jk} \\
 & \quad w_j|_F \text{ is linear} \quad \forall F \in \mathcal{F}_C \\
 & \quad \sum_{j=1}^m w_j(\mathbf{p}) = 1 \quad \forall \mathbf{p} \in \Omega \\
 & \quad 0 \leq w_j(\mathbf{p}) \leq 1, j = 1, \dots, m \quad \forall \mathbf{p} \in \Omega
 \end{aligned} \tag{4.1}$$

where \mathcal{F}_C is the set of all cage faces (e.g. the face running from H_0 to H_1 in Figure 4.1), δ_{jk} is Kronecker's delta, $w_j(\mathbf{p})$ is the weight at a point p , H_k is the k -th handle, $\Omega \subset \mathbb{R}^2$ or \mathbb{R}^3 is the volumetric domain enclosed by the union of the given shape \mathcal{S} and cage controls and V is the Laplacian energy function. In the next section we show how we translate this to a macromolecule.

In computer graphics, affine transformations are usually expressed in homogeneous coordinates. This means that a 3-D point (p_x, p_y, p_z) is represented by the 4-vector $\mathbf{p} = (p_x, p_y, p_z, W)$, where W is typically 1. A transformation is represented by a 4×4 -matrix T like the ones in Eq. (4.2), where the fourth row and column make sure that the new point is returned in homogeneous coordinates. The advantage is that rotation and scaling, as well as translation can be applied using a simple matrix-vector multiplication $\mathbf{p}' = T\mathbf{p}$. This allows the concatenation of transformations, e.g. apply a rotation and a translation consecutively [92].

In linear blend skinning (LBS), a transformation is represented by an affine transformation matrix with homogeneous coordinates T . A point $\mathbf{p} \in \Omega$ is then transformed according to a linear combination of the transformations on handles as

$$\mathbf{p}' = \sum_{j=1}^m w_j(\mathbf{p}) T_j \mathbf{p} = \left(\sum_{j=1}^m w_j(\mathbf{p}) T_j \right) \mathbf{p}, \quad (4.2)$$

where T_j is the transformation matrix of the j -th handle.

Linear blend skinning is a very popular fast and straightforward skinning algorithm, which unfortunately suffers from some unwanted artifacts, e.g. the candy-wrapper effect in Figure 4.2. This is a schematic drawing on an arm with two joints or handles, H^1 and H^2 , connected by a bone. Also, there is a bone connected to handle H^2 . We might think of handle H^1 being a shoulder joint and H^2 being an elbow joint. The joints are both on the x -axis. Naturally, the outer layer (skin) of the arm is at some distance from the joints. On this skin there is a point \mathbf{p} , with x -coordinate equal to that of handle H^2 . Point \mathbf{p} has weights $w_1(\mathbf{p}) = w_2(\mathbf{p}) = 0.5$.

In this example handle H^2 is rotated 180° around the x -axis: the elbow is twisted 180° . No transformation is applied to H^1 , hence we can express its transformation as $T_{H^1} = I$. The following homogeneous matrix multiplication is used to represent the rotation of H^2

$$H^{2'} = T_{H^2} H^2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} H_x^2 \\ H_y^2 \\ H_z^2 \\ 1 \end{pmatrix} = \begin{pmatrix} H_x^2 \\ -H_y^2 \\ -H_z^2 \\ 1 \end{pmatrix} = H^2,$$

because $H_y^2 = H_z^2 = 0$. The transformation matrix for point \mathbf{p} becomes

$$T_p = 0.5 \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} + 0.5 \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

and applying this to point \mathbf{p} gives

$$\mathbf{p}' = T_p \mathbf{p} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} p_x \\ 0 \\ 0 \\ 1 \end{pmatrix}.$$

Now $\mathbf{p}' = (p_x, 0, 0) = (H_x^2, 0, 0) = H^2$: point \mathbf{p} has collapsed onto H^2 . There are many other unfavorable artifacts, which are described in more detail in [91] and [3].

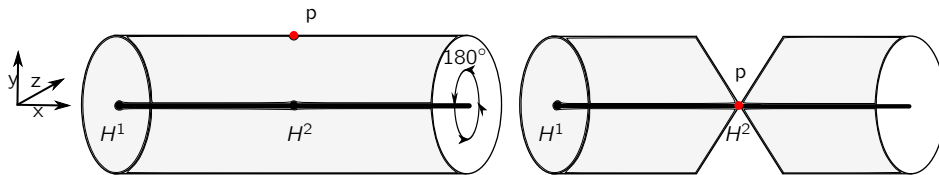


Figure 4.2: Candy wrapper effect in LBS. Point \mathbf{p} has weights $w_1(\mathbf{p}) = w_2(\mathbf{p}) = 0.5$. No transformation is applied to handle H^1 . Handle H^2 is rotated 180° around the x -axis and stays in place. Point \mathbf{p} however collapses onto H^2 .

Recently, LBS has gotten competition from more advanced methods, which are able to avoid artifacts such as the candy-wrapper effect. Some of these methods are almost artifact-free, but at a considerable cost: these methods are often iterative and slow for large systems. In [3], Dual quaternion Linear Blending (DLB) is introduced. This method is able to achieve high-quality skin deformation, but with computational complexity comparable to that of LBS. For a solid introduction to dual quaternions, rigid body motions and transformations the reader is referred to [93]. Here it suffices to say that a quaternion is a linear combination $a_0 + a_1i + a_2j + a_3k$ of the basis elements $1, i, j$ and k . A dual quaternion is an extension of a quaternion with a dual component $\epsilon(b_0 + b_1i + b_2j + b_3k)$. Addition, division and multiplication of dual quaternions are defined, as well as the conjugate and norm. We can represent a point in 2-D or 3-D in the dual components, define a rotation or translation (or a combination of both)

as a dual quaternion and then use dual quaternion multiplication to apply the transformation. In the candy-wrapper example, the transformation on H_2 is given as $\hat{\mathbf{q}}_2 = (0 + i)$. The transformation on H_1 is $\hat{\mathbf{q}}_1 = 1$. In [3], DLB is defined as

$$DLB(w_1, w_2; \hat{\mathbf{q}}_1, \hat{\mathbf{q}}_2) = \frac{w_1 \hat{\mathbf{q}}_1 + w_2 \hat{\mathbf{q}}_2}{\|w_1 \hat{\mathbf{q}}_1 + w_2 \hat{\mathbf{q}}_2\|}.$$

For point \mathbf{p} this would become

$$T_{\mathbf{p}} = \frac{0.5 + 0.5i}{\|0.5 + 0.5i\|} = \sqrt{0.5} + \sqrt{0.5}i.$$

This rotation is applied to point \mathbf{p} as

$$\begin{aligned} \mathbf{p}' &= T_{\mathbf{p}} \mathbf{p} T_{\mathbf{p}}^* = \left(\sqrt{0.5} + \sqrt{0.5}i \right) (1 + \mathbf{p}_x i \epsilon + \mathbf{p}_y j \epsilon + \mathbf{p}_z k \epsilon) \left(\sqrt{0.5} - \sqrt{0.5}i \right) \\ &= (1 + \mathbf{p}_x i \epsilon - \mathbf{p}_z j \epsilon + \mathbf{p}_y k \epsilon), \end{aligned}$$

where $T_{\mathbf{p}}^*$ is the conjugate of dual quaternion $T_{\mathbf{p}}$. Expressing the dual quaternion result in Euclidean coordinates gives $\mathbf{p}' = (\mathbf{p}_x, -\mathbf{p}_z, \mathbf{p}_y) \neq H_2$. The point \mathbf{p} has made a rotation of 90° around the x-axis, half that of H_2 . Hence, the candy wrapper artifact is not present in DLB. In [3], more examples are shown where LBS artifacts are present, but DLB performs as expected. In terms of memory requirements, dual quaternions only require 8 numbers, where matrices require 12 (the bottom row is always the same). Applying the transformation takes slightly more instructions than LBS (43 vs. 36). The generalized DLB equation with convex weights is

$$DLB(\mathbf{w}; \hat{\mathbf{q}}_1, \dots, \hat{\mathbf{q}}_n) = \frac{\sum_{j=1}^m w_j \hat{\mathbf{q}}_j}{\left\| \sum_{j=1}^m w_j \hat{\mathbf{q}}_j \right\|}.$$

We propose to generate convex weight sets for every particle in a macromolecule and use DLB to blend rigid transformations on a set of handles H into transformations on the set of particles.

4.3 Methods

The algorithm we propose for interactive molecular deformation contains two major steps:

1. *Bind time*: find weight functions on graph so that every particle has a convex combination of weights.
2. *Pose time*: apply rigid body transformations to particles according to their weights and transformations on a finite set of handles.

In the simplest case, we apply both steps only once. If we want to apply more complex transformations, we can apply step 2 several times. To keep the system up to date, we might have to apply step 1 after we apply step 2 several times.

In the previous section, we have motivated our choice for DLB as a means of deformation in step 2. Here, we explain how we obtain a molecular graph, apply a force field to it and obtain weights for deformation in step 1. For now, we propose a very simple forcefield with either covalent bonds or non-bonded Vanderwaals interactions between particles. In future implementations, this could be extended to include dihedrals and atomic angles. We use the GraphTreeLib library from [94] to parse a PDB file into an undirected molecular graph, where the set of vertices V represents atoms and edges in E represent covalent bonds between atoms. Short range non-bonded interactions are then modeled by edges connecting particle pairs within cutoff distance. We find these edges using a cell grid. By applying an appropriate parameter set such as the one in CHARMM22, we turn this into a weighted graph. Considering our simple forcefield, we might model covalent bonds using Hooke's law and non-bonded short-range interactions using a Lennard-Jones potential.

There are m handles, denoted by $H_j \in V$, $j = 1, \dots, m$. To apply blending of rigid transformations on all atoms/vertices in the molecular graph $G = (V, E)$, we need to set a weight $w_j(v_i)$ for each vertex/atom v_i and handle j , so that every vertex is transformed according to Eq. (4.2).

In [2], some favorable properties of the weight functions are listed. Of these, the following apply to our problem:

Non-negativity We impose that $0 \leq w_j(v_i) \leq 1$, $j = 1, \dots, m$, $\forall v_i \in V$. Behavior should be intuitive. In graphics negative weights have shown to produce unintuitive handle influences, causing regions with negative weights to move in the opposite direction to the handle transformation. In [2] and [3], transformations on points are convex combinations of transformations on handles, i.e. $\sum_{j=1}^m w_j(v_i) = 1$, $\forall v_i \in V$.

Shape-awareness Ultimately, the weight distribution should take into account primary and secondary structures of a protein, without the need to evaluate these beforehand, e.g. using DSSP [95].

Partition of unity If the same transformation T is applied to all handles, T should be applied to the molecule as a whole. So the weights on each particle should sum up to 1.

Locality and sparsity The set of handles influencing a particular vertex should be as small as possible, to allow for rapid real time blending in pose time. If a handle occludes the path from another handle H_j to some vertex v_i , then $w_j(v_i)$ should be zero. A good standard is to have at most 4 influencing handles per atom: hence in the weight matrix $W \in \mathbb{R}^{n \times m}$, only $\frac{4}{m} \cdot 100\%$ of entries is non-zero.

No local maxima Each w_j distribution should obtain its maximum at $w(H_j) = 1$ and decrease monotonically away from H_j . Only weight function w_k has a non-zero value at the vertex corresponding to handle H_k , that is $w_k(H_k) = 1$: hence $w_j|_{H_k} = \delta_{jk}$. Because we are working on a graph and not in \mathbb{R}^3 , this means that w_j should decrease as the weighted shortest path distance to H_j increases.

Quadratic program

In Eq 4.1, the weights w_j are defined as minimizers of a higher-order shape-aware smoothness functional, the Laplacian energy, subject to the aforementioned constraints. The problem is discretized in [2] using linear finite elements, where the domain Ω is meshed so that there is a set of vertices V which define a triangle/tetrahedral mesh \mathcal{M} . This finite element discretization corresponds very well to the molecular graphs that we use, where we already have one vertice for each particle. A weight function w_j is then a piecewise linear function, where we are interested in the values at the vertices. Hence, we can write the weight function optimization problem as:

$$\begin{aligned} \sum_{j=1}^m \frac{1}{2} \|\nabla w_j\|^2 dV &\approx \sum_{j=1}^m \frac{1}{2} (M^{-1}L\mathbf{w}_j)^T M (M^{-1}L\mathbf{w}_j) \\ &= \frac{1}{2} \sum_{j=1}^m \mathbf{w}_j^T (LM^{-1}L) \mathbf{w}_j, \end{aligned}$$

where \mathbf{w}_j , $j = 1, \dots, m$ are column vectors corresponding to weight functions of handles, M is a diagonal mass matrix and L is a symmetric stiffness matrix, i.e. the Laplacian matrix. In [2], the stiffness of the object is assumed to be homogeneous and therefore the cotangent Laplacian matrix is used for L together with a lumped mass matrix M , where the Voronoi area/volume of each vertex determines its diagonal entry. Now we derive matrices L and M for the molecular graph.

Laplacian matrix

A definition of the Laplacian of a function w_j on a non-weighted graph $G = (V, E)$ is in [96]:

$$(\Delta \mathbf{w}_j)(v_i) = \sum_{k:(v_i, v_k) \in E} [w_j(v_k) - w_j(v_i)],$$

where $\Delta \mathbf{w}_j$ is a vector containing the Laplacian values on all vertices, and v_i and v_k are vertices in V .

We would like the weights in strongly bonded atom pairs to be more similar than in weakly bonded atom pairs, in order to get shape-aware weight distributions. Therefore, we should take the strength of the bond in atom pairs into account. We define a weight function $\gamma(v_i, v_k)$ which applies a weight to every edge $(v_i, v_k) = E$. The value of $\gamma(v_i, v_k)$ represents the strength of the bond between atoms v_i and v_k . The weight function is symmetric: $\gamma(v_i, v_k) = \gamma(v_k, v_i)$.

Now the weighted Laplacian is

$$\begin{aligned}
(\Delta \mathbf{w}_j)(v_i) &= \sum_{k:(v_i, v_k) \in E} \gamma(v_i, v_k) [w_j(v_k) - w_j(v_i)] \\
&= \left[\sum_{k:(v_i, v_k) \in E} \gamma(v_i, v_k) w_j(v_k) \right] - \left(\sum_{k:(v_i, v_k) \in E} \gamma(v_i, v_k) \right) w_j(v_i).
\end{aligned}$$

We can write this as a matrix-vector multiplication. First we define the weighted Laplacian matrix L which has entries

$$\ell_{i,j} := \begin{cases} \sum_{k:(v_i, v_k) \in E} \gamma(v_i, v_k) & \text{if } i = j, \\ -\gamma(v_i, v_j) & \text{if } i \neq j \text{ and } (v_i, v_j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

The Laplacian matrix is also written as $L = D - A$ where D is called the diagonal degree matrix of G and A is the weighted adjacency matrix. Now we see that

$$\Delta \mathbf{w}_j = L \mathbf{w}_j.$$

Note that L is symmetric, $L^T = L$. The degree of the vertices on the graph could depend on a threshold on the strength of the atom-atom bond. Thus, the higher we set this parameter, the sparser the matrix L .

Mass matrix

In [2], the mass matrix is based on the Voronoi area/volume of vertices. In our case, this would not make sense, as it would represent separated atoms as heavy, while it would represent atoms in dense areas as light. Instead, we might just use each atom's mass. Then the mass matrix becomes a sparse $n \times n$ matrix with mass m_i of every particle on the diagonal. Also, we can let M be the identity matrix I and only worry about bonds between particles through the Laplacian.

Shape preservation

The user might choose to rigidify certain structures, either at a low level (side chains) or at a higher level (α -helices, β -strands etc.). In this case, all atoms/vertices in the rigidified structure should have (approximately) the same weights so that they are transformed together. In order to achieve this, the weights should be forced to take on the same values, i.e. minimize the variation in the weights by solving a least-squares problem. In [2], a brush is proposed which is used to give a rigidity value to every vertex in the mesh. Here, we propose to set, for every rigidified structure, the value of the brush to 1, and 0 elsewhere. Thus, for every rigidified structure π we define a subgraph $G^\pi = (V^\pi \subseteq V, E^\pi \subseteq E)$. We want to minimize the differences in weight values within the rigidified structure, hence we try to minimize:

$$\sum_{j=1}^m \frac{1}{2} \left(\sum_{v_i \in V^\pi} \sum_{k:(v_i, v_k) \in E^\pi} \|w_j(v_i) - w_j(v_k)\|^2 \right) = \sum_{j=1}^m \mathbf{w}_j^T L_\pi \mathbf{w}_j,$$

where L_π has the shape

$$\ell_{i,j}^\pi := \begin{cases} |\{k : (v_i, v_k) \in E^\pi, v_k \in V^\pi\}| & \text{if } i = j \text{ and } v_i \in V^\pi, \\ -1 & \text{if } i \neq j \text{ and } (v_i, v_j) \in E^\pi \text{ and } \{v_i, v_j\} \subseteq V^\pi, \\ 0 & \text{otherwise.} \end{cases}$$

Hence, L_π is just the non-weighted Laplacian matrix of the subgraph π . The matrix L_π is a sparse matrix which, when added to L , introduces no new non-zero entries. It is thus safe to add, for each rigidified structure π , a sparse matrix to the problem. We can introduce a parameter β_π which weighs the rigidity of structure π : the higher β_π , the more importance we put on rigidity in π . The resulting quadratic program to be solved at bind time then is

$$\begin{aligned}
& \arg \min_{w_j, j=1, \dots, m} \frac{1}{2} \sum_{j=1}^m \left(\mathbf{w}_j^T \left(LM^{-1}L + \sum_{\pi \in \Pi} \beta_\pi L_\pi \right) \mathbf{w}_j \right) \\
& \text{subject to: } w_j|_{H_k} = \delta_{jk} \\
& \sum_{j=1}^m w_j(v_i) = 1 \quad \forall v_i \in V \\
& 0 \leq w_j(v_i) \leq 1, j = 1, \dots, m \quad \forall v_i \in V, \quad (4.3)
\end{aligned}$$

with Π the set of rigidified areas. We define L^* as the sum of matrices $LM^{-1}L + \sum_{\pi \in \Pi} \beta_\pi L_\pi$.

Though this enforces similarity on vertices within a rigidified structure, it does not guarantee equality. Alternatively, a coarse graining approach such as in [61] might be used, where one bead represents a small set of particles and an amino acid or secondary structure is represented by one or more beads. This might affect smoothness of the weight function on the graph.

4.4 Implementation

Problem 4.3 is quadratic in the unknowns w_j and convex. The constraints are linear equality and inequality constraints. Because for an appropriate threshold on the bond strength the Laplacian matrices will be very sparse, we can employ a sparse quadratic programming solver to find appropriate weights. By setting a block matrix with L^* m times along the diagonal we can solve for the whole vector $\mathbf{w} = [\mathbf{w}_1^T \dots \mathbf{w}_m^T]^T$. However, the time needed to solve this problem scales superlinearly in the number of unknowns. For larger molecules the number of unknowns might be in the tens of thousands.

In [2] it is suggested to avoid the superlinear scaling of the quadratic solver by splitting the problem up in several subproblems (one for each handle) and then normalizing the weights per vertex/atom. For 46000 vertices and 7 handles, this is 50 times faster than solving one big problem. Though it imposes some errors in the weight functions, especially further away from the handle positions, the resulting weight functions still possess the required properties.

In the next section we present several problems and apply the current algorithm to them. Some of these problems are small toy problems, showing functionality on e.g. chains and rings of beads, other examples use real-world molecules like the BPTI protein, graphene and a carbon nanotube. We have implemented two versions of the algorithm, both in MATLAB [97] and in C++, where the MATLAB version was mostly used for the toy problems.

For the toy problems implemented in MATLAB, quadratic problems were solved with the interior point method of MATLAB's native `quadprog` function of the Optimization Toolbox unless stated otherwise. The macromolecule manipulation program `BlendProt` was written in C++. For quadratic optimization, the Python `CVXOPT` package was used [98].

4.5 Results

In this section we assess the functionality of our approach on 5 different problems. In problem 1, we manipulate a small chain of beads with one or two weaker parts. We show how the weight functions are distributed according to the weights on the edges and how transformations are applied. In problem 2, we demonstrate this on a ring. In problem 3 we show how the weight functions spread on a sheet of graphene weighted by a very simple force field based on CHARMM22 parameters. Problem 4 shows two examples where we use `BlendProt` to manipulate larger molecules, i.e. the bovine pancreatic trypsin inhibitor (BPTI) protein and a single-walled carbon nanotube which we unzip, thereby breaking non bonded connections and updating the graph interactively. In problem 5 we show how we can use `BlendProt` to intervene in a running MD simulation without causing extreme disturbances in the system statistics. We analyze the choice of mass matrix and skinning operators.

Problem 1: 1-dimensional chain

In the first example, the chain consists of two strongly linked parts, connected by a weak link. The link between two weakly connected vertices is 5 times weaker than the other links. Handles are applied at the

particles at both ends of the chain. Weights are computed and then a translation of 4 in the y -dimension is applied to handle 2 (the top atom). Throughout this manipulation, the position of the bottom particle is fixed. The other atoms move according to their weights. In this case, it does not matter whether we use LBS or DLB skinning to apply the transformations. We use uniform masses: $M = I$.

The weight functions in Figure 4.3 show a pattern for handle H_1 : the handle has a lot of influence on vertices which are in the component of which H_1 is a part, with a steep drop between vertices 5 and 6, due to the weak link between the vertices. Then in the second component, the values of w_1 are much smaller. The weights decrease monotonically as the distance to the handle increases. The weight function w_2 of H_2 shows the inverse pattern.

If instead we choose to use three components connected by two weak links, we get the situation in Figure 4.4. In the weights diagram, we clearly see three groups of vertices. This is reflected in the translation figure.

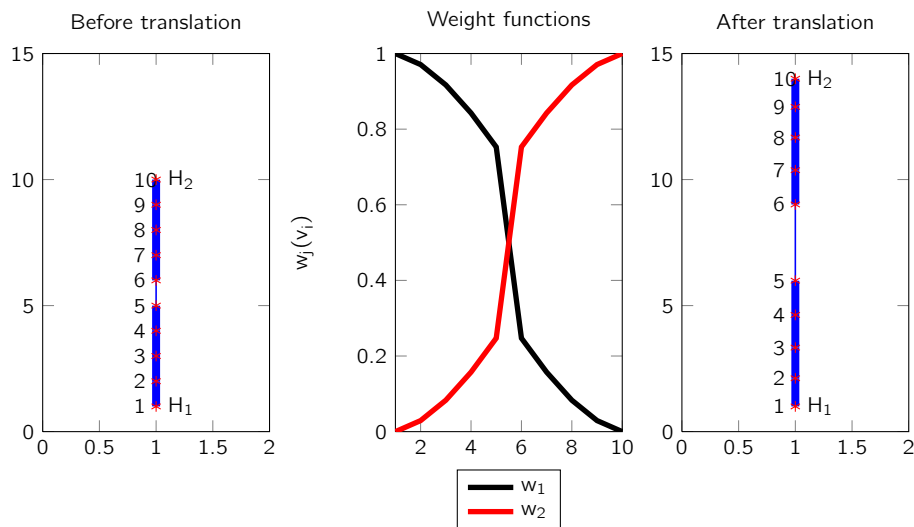


Figure 4.3: 10-vertex chain with weak link between vertex 5 and 6. Uniform mass.

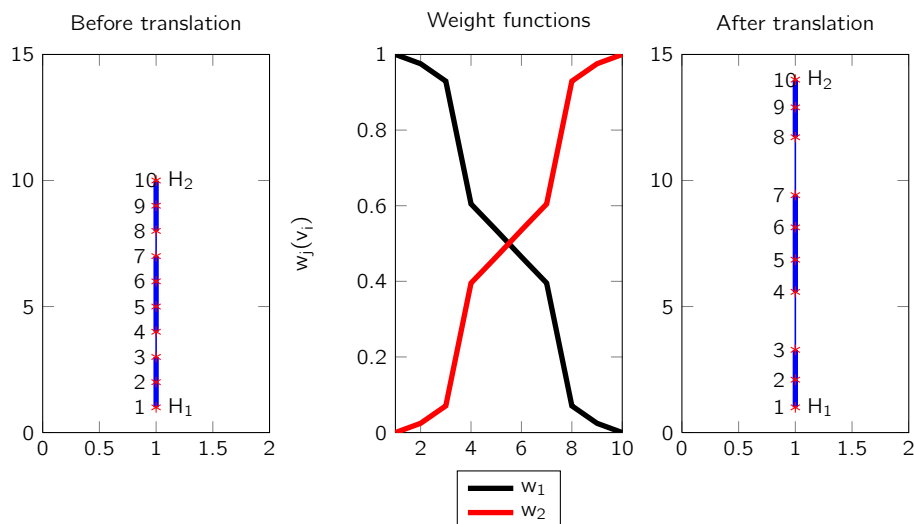


Figure 4.4: 10-vertex chain with weak links between vertex 3 and 4 and vertex 7 and 8. Uniform mass.

Problem 2: 2-dimensional ring

In this problem there are ten atoms/vertices, this time ordered in a ring with two side chains. All connections have the same strength. We try to rigidify the ring as proposed in Section 4.3. We define a

structure π containing vertices $\{3, 4, 5, 6, 7, 8\}$ in the ring and its corresponding Laplacian matrix. When its rigidity parameter β_π is set to zero, we get the result in the center of Figure 4.5. Also in Figure 4.5, we see how the value $\sum_{j=1}^m \frac{1}{2} \left(\sum_{i \in \pi} \sum_{k: (i,k) \in E} \|w_j(v_i) - w_j(v_k)\|^2 \right) = \sum_{j=1}^m \mathbf{w}_j^T L_\pi \mathbf{w}_j$ decreases as we increase the value of β_π . We can thus successfully impose that all weights within a rigidified structure are very similar.

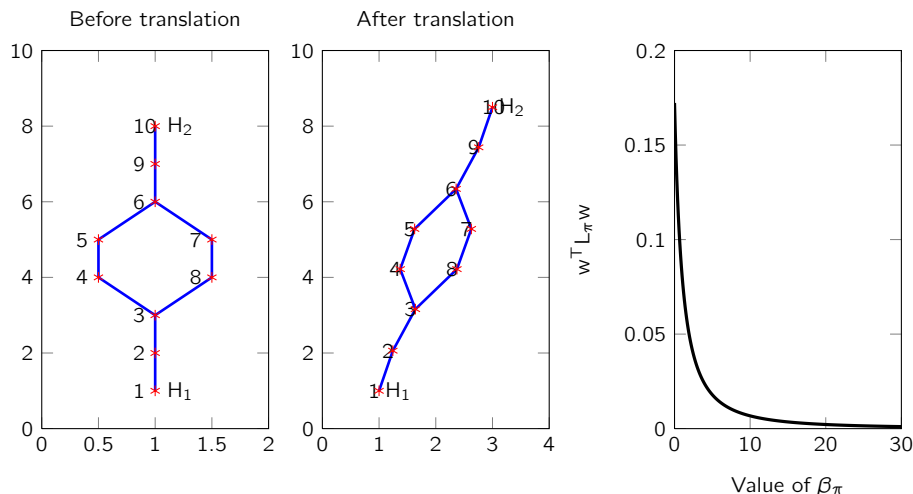


Figure 4.5: Manipulating a chain with a ring. Weights on the edges between vertices are homogeneous. When we impose no rigidity on the ring, it is stretched as in the figure in the middle. If instead we impose rigidity through β_π , we see how the difference between weights in the ring decreases as β_π increases.

Problem 3: Graphene

Now we take a 350-atom sheet of graphene and parse it into a graph such that every covalent bond in the graphene is an edge in the graph. On these edges, we put weights as

$$\gamma_{v_i, v_k} = k_b \left(\frac{b}{b_0} \right), \quad (4.4)$$

where k_b is the spring constant, b_0 is the equilibrium bond length and b is the bond length. Parameters were taken from CHARMM22 as $k_b = 600 \text{ Kcal mol}^{-1} \text{ \AA}^{-2}$ and $b_0 = 0.142 \text{ \AA}$. If a bond length is smaller than equilibrium, the weight is smaller so that atoms move more apart. If it is above equilibrium, the weight is larger so that atoms move more alike and hence stay closer.

We place either 2, 3, 4 or 5 handles on the lattice and solve the quadratic program in Eq. (4.3). Figure 4.6 shows weight function w_1 of handle H_1 . The scale runs from blue (0) to red (1). We see that as the number of handles increases from 2 to 5, the influence of handle 1 decreases. In the end, only a small area of the graphene is colored red. In the post-translation figures, we also see how the graphene is fixed at the handles as more handles are added.

As addressed in [2] the bind time needed to solve the quadratic program scales superlinearly in the number of handles. The number of unknowns is $N_h n$, where N_h is the number of handles and n is the number of vertices. The number of equality constraints is $N_h + n$. In Table 4.1 we show the average time needed to solve the quadratic program for a uniform 30 by 30 2-D lattice. Also, we show the time needed to solve the separated quadratic program, where we solve N_h separate programs with n unknowns and N_h equality constraints. We see here that bind time is significantly reduced when we separate the program.

In Figure 4.7 we see that this reduced bind time comes at a cost in terms of quality of the weights. Figure 4.7a shows the error in a 31 by 31 uniform lattice when solving N_h separate problems. Though the weight functions are still convex, small errors might be found with respect to the weights predicted in the original quadratic program. As already noted in [2], these errors are larger at points further away from handles. For the 4-handle example in Figure 4.7 we see how the error is zero at the handles and reaches its maximum value between two handles. Figure 4.7b shows a scatter plot, with the shortest

N_h	Original BT	Separated BT	Speedup
2	1.30	0.49	2.65
3	2.38	0.81	2.93
4	4.13	1.05	3.93
5	7.23	1.47	4.91

Table 4.1: Time needed to solve the original quadratic program (BT = bind time) or the separated quadratic program for a system with 900 vertices and 2 to 5 handles. For much larger systems, [2] reports speedups up to 50.

path distance to a handle on the x -axis and the weight error on the y -axis. We see how the error depends on the distance to the closest handle and on the number of influencing handles. The larger the number of influencing handles, the smaller the error. The larger the distance, the larger the error. Furthermore, errors are expected to be larger along the boundaries of the lattice, as the separate problems impose less constraints than the original problem. These results are in accordance with those in [2]. All in all, the errors are small compared to the weight values, which run from 0 to 1, and deformations using separate problems or one problem are visually indistinguishable.

Problem 4: Molecule manipulation

The BlendProt program, written in C++ and using Python for optimization, shows the same behavior as the Matlab toy models. In Figure 4.8, we show translation of one handle in a 4-handle setup, as in Figure 4.6. Also, we can manipulate more complex proteins like the bovine pancreatic trypsin inhibitor (BPTI) protein, popular in MD simulations of proteins [26]. In Figure 4.9 we see the weight function of one handle on the BPTI graph and see what happens when we move this handle.

In Figure 4.10 we 'unzip' a (10, 10) armchair single-walled carbon nanotube (SWCNT). The equilibrium length between bonded carbon atoms is $b_0 = 1.42\text{\AA}$. We scale the weights as in Eq. (4.4) and apply a cutoff at 2\AA : if a bond is stretched this far it 'snaps'. Beyond this point, atoms interact through a Lennard-Jones potential. Using this model we can set handles at the nanotube and apply transformations so that the tube unzips as in Figure 4.10. Every time the mouse is released, the bonds are evaluated and some bonds snap. The edge weights in the graph are adjusted and a new set of convex weight functions is found on the molecular graph by solving the separated quadratic program. Hence, we repeatedly apply step 1 of the algorithm after some interval. The shorter this interval, the more accurate deformation is, but the less interactive manipulation is. Instead of the simple force field used here, we might apply a reactive empirical bond order (REBO) potential to even include angles and dihedrals in the graph weighing [99, 100].

Problem 5: Simulation manipulation

One goal of the method developed here is to allow the researcher to intervene in a running simulation, e.g. to help a simulation escape a local energy minimum or apply some heuristic manipulations to a macromolecule. We have run a small NVT MD-simulation on the GPU, with a 5-monomer polymer solvated in 343 liquid argon particles. Parameters are similar to those in Benchmark 4 of Chapter 3. Bonds between monomers are modeled with springs, $d_0 = 2.4$, $k = 30000\epsilon/\sigma^2$, non-bonded interactions are modeled by LJ potentials. Temperature is set to $T = 350K$ and we simulate in a 3-D box with box lengths 25.56\AA . Figure 4.11 shows this system. Initially, we run 20000 steps of the MD algorithm.

Then we load the system state in our BlendProt software, apply appropriate weights to the edges of the molecular graph and apply some very small transformations to some of the monomers, see Figure 4.11. We resume the MD simulation and see how the temperature has changed due to our manipulations (Figure 4.12). After 20000 more steps in which the temperature returns to its normal level, we apply a somewhat larger manipulation. We see that temperature increases dramatically, but returns again to its original value after some 10000 steps. This shows that we can use BlendProt to apply minor or major changes in running simulations. This technique might also be applied to energy minimization algorithms, where the user can intervene in the simulation.

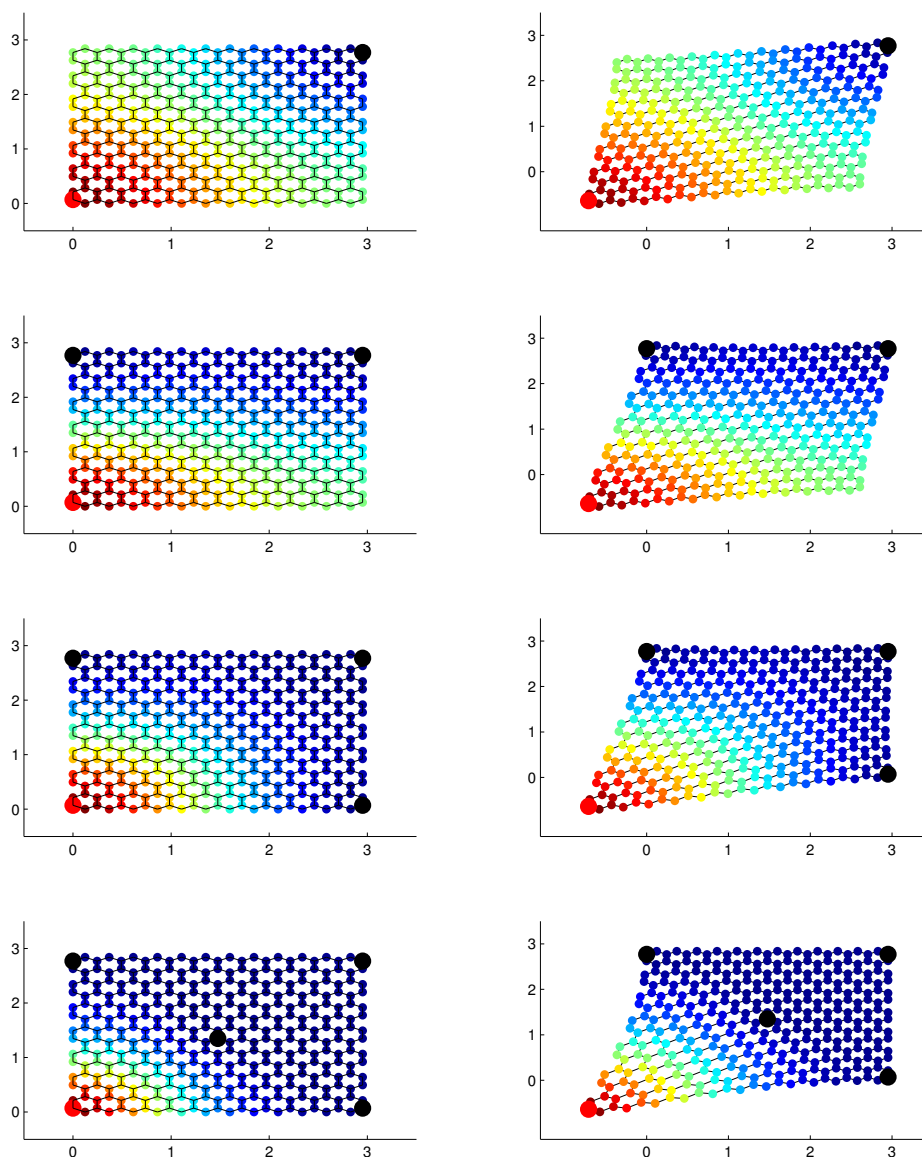


Figure 4.6: 350-particle graphene sheet with 2, 3, 4 or 5 handles (large dots). As the number of handles increases, the influence of handle 1 (red dot) decreases. Color scale runs from blue (0) to red (1). We see how there are no local maxima and the function is smooth. Handle 1 is translated, this translation is blended on the other vertices according to the weight function w_1 .

4.6 Conclusion and discussion

We have introduced a method which allows the user to interactively apply rigid body transformations to parts of a molecule, based on a force field of choice applied to a weighted molecular graph. A set of convex weight functions is found on the molecular graph, which has attractive and intuitive properties such as unity of partition and non-negativity. During manipulation, every particle is translated and rotated according to rigid body transformations applied to a finite set of handles. These are represented by dual quaternions and blended by dual quaternion blending. We have shown how we might preserve structures

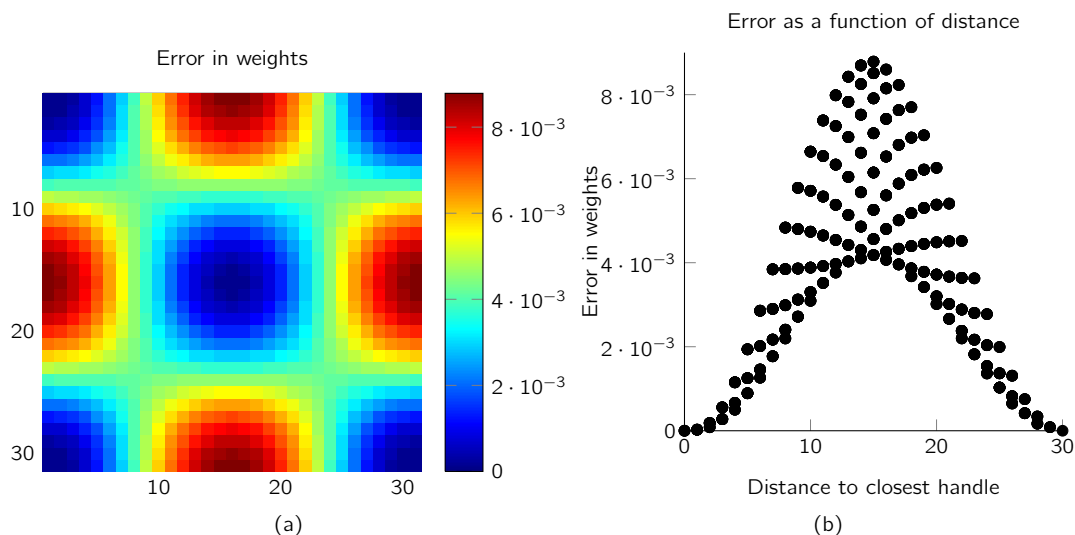


Figure 4.7: Error in weights. Figure (a) shows the norm of the weight error vector of every vertex on a 31×31 lattice. Handles were put on the four corners of the lattice. We see that on the boundaries of the lattice, between handles, the error in the weight approximation is largest. In the center of the lattice the error is very small. Figure (b) shows a scatter plot, with the shortest path distance to a handle on the x -axis and the weight error on the y -axis. This supports the claim that near the boundaries of the graph (distance approximately 15) the weight errors are larger than in the center (distance 30). This might be because the weights are less restrained near the boundaries in the separate problems.

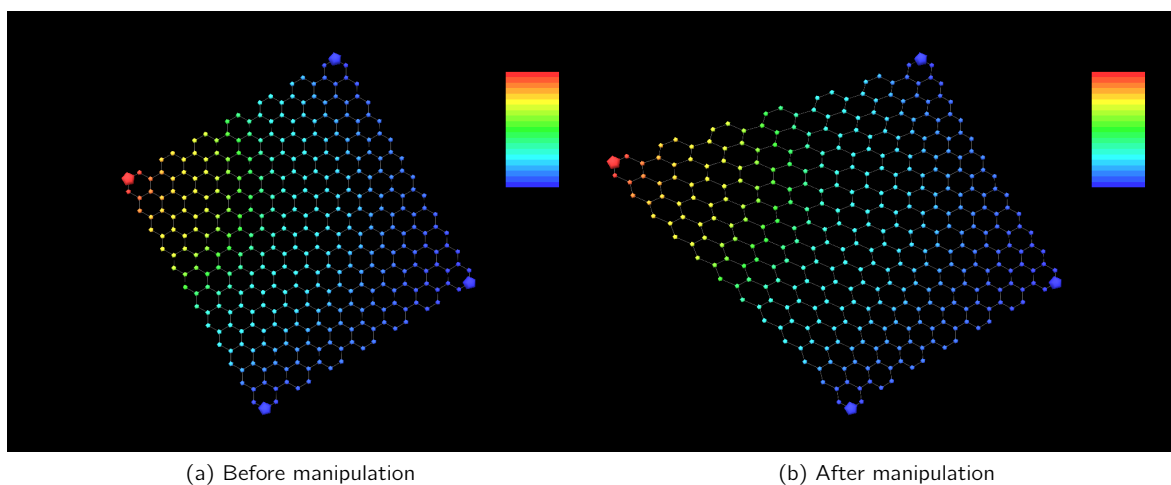


Figure 4.8: BlendProt program manipulating a 350-particle sheet of graphene. In Figure (a) we see the square hexagonal lattice before manipulation, with the weight function of the top left handle. In Figure (b) we see how the handle has been translated to the left and the sheet is stretched more closely to this handle. The weight distribution is very similar to that in Figure 4.6.

within the molecule and how we can change stiffness in the molecular bonds. Furthermore, we can change the structure of the graph on the fly and update weight functions accordingly to model e.g. unzipping of a carbon tube. Finally, we have shown how we can use our method to intervene in a running MD simulation without causing implausible changes in its statistics.

In a forthcoming paper, Weber et al. publish on biharmonic coordinates (BC), an approach similar to bounded biharmonic weights in the sense that the biharmonic Dirichlet problem is solved over a region Ω [101]. In BC cage coordinates are blended, instead of affine transformations. It is shown that this is advantageous in some cases, where the transformations are automatically approximated in the BBW method. In future research, the advantages of BC for protein manipulation might be investigated. Also,

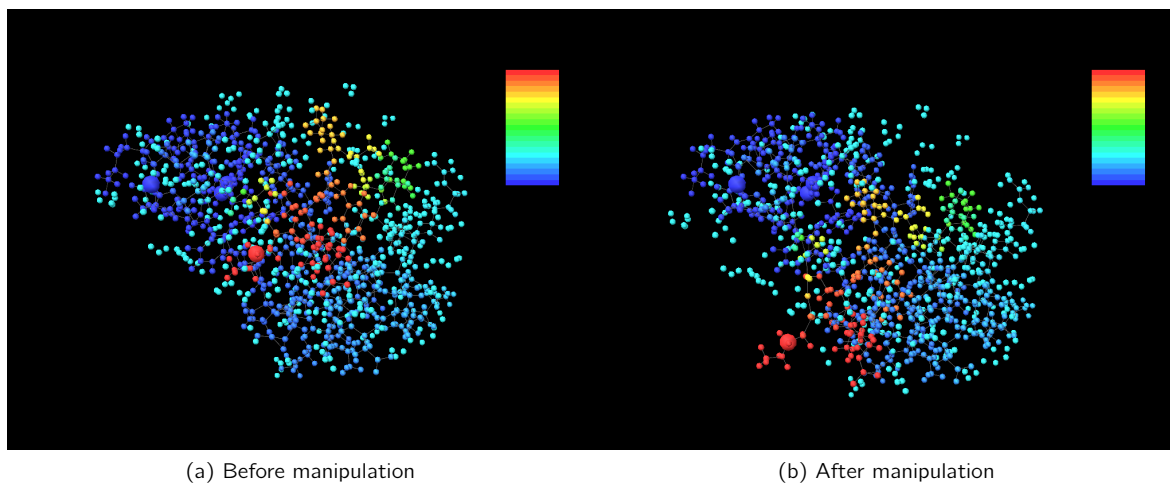


Figure 4.9: BlendProt program manipulating a bovine pancreatic trypsin inhibitor (BPTI) protein. In Figure (a) we see how the weights are spread over the molecular graph. The selected handle has a large influence on a subset of the particles and almost no influence closer to the other handles. In Figure (b) we have applied a relatively large translation to this handle. While most of the protein stayed in place, a small set of particles moved along with the handle according to their weights.

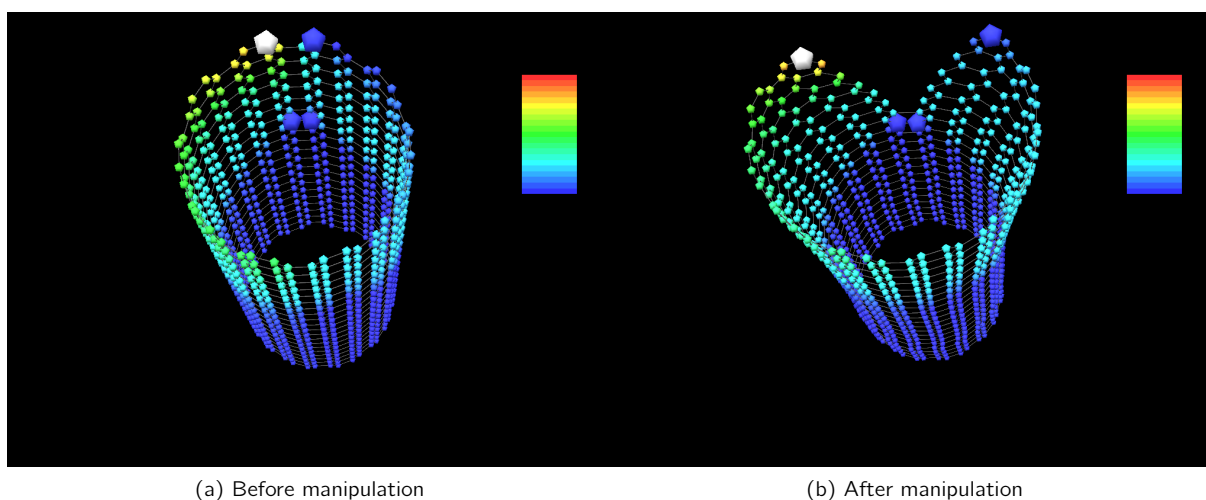


Figure 4.10: BlendProt unzipping a (10, 10) single-walled carbon nanotube (SWCNT). In Figure (a) we see how initially we place four handles. The selected (white) handle influences a small region. In Figure (b) we have moved the top two handles apart and caused the SWCNT to unzip. Bonded connections snap after some critical length, here 2\AA .

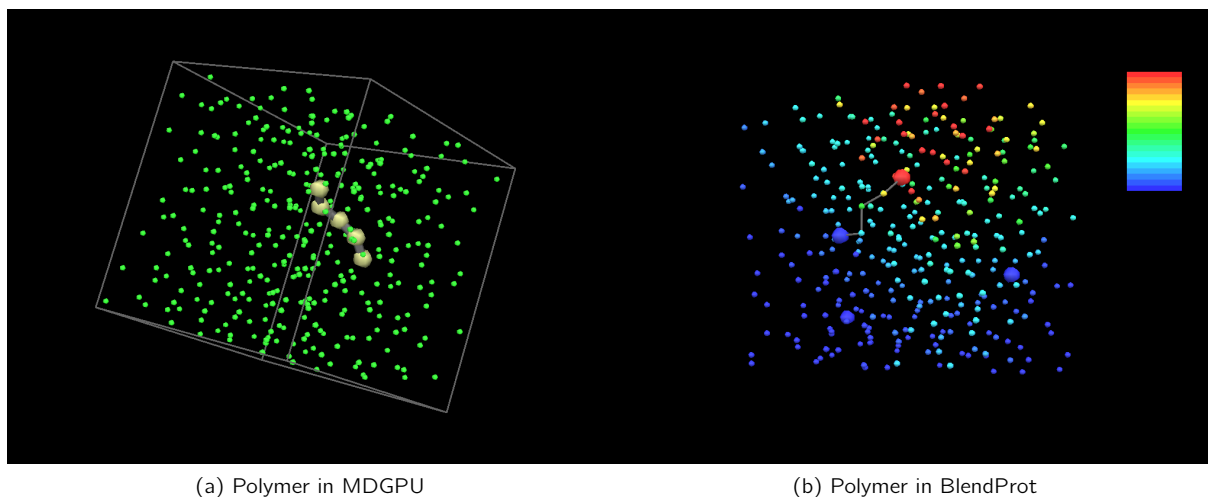


Figure 4.11: (a) Polymer in solvent in GPU MD (b) BlendProt program manipulating a 5-monomer polymer in an argon liquid solvent.

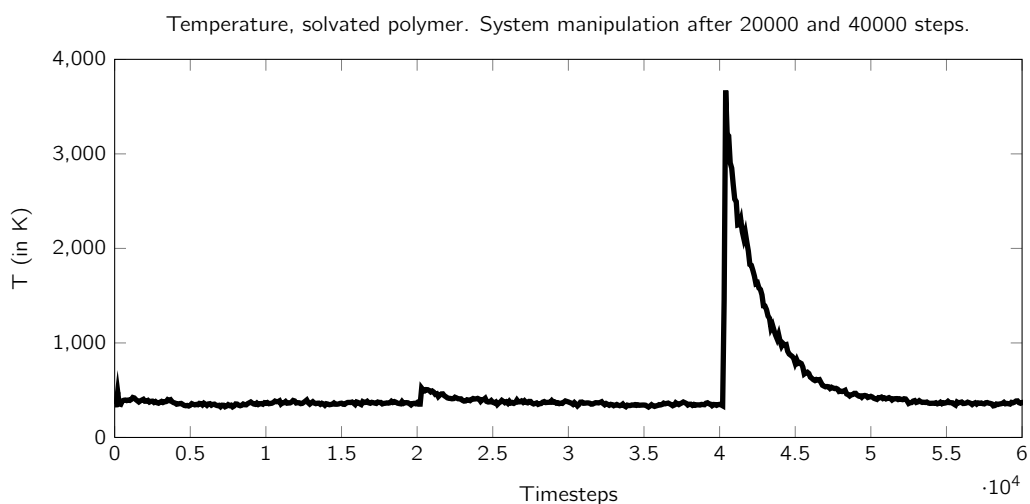


Figure 4.12: Temperature in NVT simulation of 5-monomer polymer solvated in 343-particle argon liquid. After BlendProt manipulation at 20000 and 40000 steps, the system is disturbed but returns to its original temperature.

future research might employ recent more sophisticated quatrharmonic shape-aware functions according to [102]. In this paper, Jacobson et. al show how an enhanced version of bounded biharmonic weights is able to preserve shapes and extrema better than bounded biharmonic weights at a computationally attractive cost.

The set of force fields which might be applied in our method is huge. In fact, every problem has his specific force field, and as such we can apply e.g. REBO to a nano tube or simple LJ potentials to argon liquid. It should be noted that the transformations which we propose are small: applying large transformations without updating the weights or forcing some MD or minimization step will still cause disruptions in the system statistics.

Conclusion and discussion

We have introduced two methods which can help extend the time scales of molecular simulations. These are two very different approaches, with the ARPS on GPU method introduced in Chapter 3 focused on molecular dynamics simulations of systems of particles and the protein manipulation approach in Chapter 4 pointed at modeling of such systems. The algorithm introduced in Chapter 3 tries to accurately mimic behavior of systems of particles even though degrees of freedoms are turned on and off dynamically. The algorithm in Chapter 4 is much less accurate but allows for very rapid manipulation of large systems of macromolecules.

In Chapter 3 we have seen how, using ARPS on a GPU, we might theoretically reach speedups of more than 20 times over a conventional MD on GPU implementation. Due to the lack of support for successful implementations of Newton's third law on the GPU, theoretical speedup is even better than on the CPU. In our experiments, we have seen that even when only a very small percentage of particles is active, we can still obtain correct statistics. For the corresponding levels of activity, ARPS was able to reach a speedup of 12 over a full dynamics GPU implementation in a million-particle NVE simulation. This full dynamics simulation in turn was 40 times faster than its CPU counterpart. The speedup of ARPS on a GPU does not depend only on the ratio of active particles, but also on the size of the system. The larger the system, the better the experimental results match theoretical speedup. Though considerable care has been given to general implementation details, such as the linked cell list and ordering of active and restrained particles, we do not claim that the implementations analyzed in Chapter 3 are optimal. Larger speedups might be reached with respect to the CPU implementations. However, because our full dynamics and ARPS program on the GPU share the same implementation details, we have successfully shown that ARPS might reduce computing time on the GPU significantly.

For a comparison of ARPS and full dynamics methods, in Table 5.1 we show just how much time we can simulate in one hour wall clock time, for a 21952 particle system on the CPU and a 343000 particle system on the GPU, with the same parameters and thresholds. Here we see two things: the GPU implementations are able to simulate more steps for larger systems in the same amount of time and ARPS is beneficial on both CPU and GPU. Here, the speedup caused by ARPS on GPU is around three, but we have seen that we might experimentally reach speedups of up to 12 for a larger system.

In the experiments run in Chapter 3 we have encountered some problems with respect to stability and reproducibility. ARPS on a GPU seems to be less stable and accurate than ARPS on a CPU or a full dynamics GPU implementation. Stability issues in GPU accelerated MD simulations are not new.

	CPU (N=21952)		GPU (N=343000)
	9.8ps	Full dynamics	28.1ps
	120.3ps	ARPS	87.8ps

Table 5.1: Simulated time in one hour wall clock time. Though the GPU particle system is 15 times larger than that on the CPU, in a full dynamics GPU simulation we can simulate three times more steps than on the CPU. ARPS on GPU further enhances this with a factor three.

In force summation, small errors accumulate. In single precision (SP) simulations, this is worse than in double precision (DP) simulations. Therefore, most MD simulations (including the ones in this thesis) are conducted using DP. However, this comes at a considerable performance penalty. For example, for NVIDIA's flagship GTX680, GPU SP and DP have a performance ratio of 20 to 1 [103]. Several methods have been introduced to employ the SP processing power of GPUs and still obtain DP quality results. In [103], a combination of single precision floating point numbers and 64-bit integers is shown to be able to perform all common MD tasks at high accuracy. In [69], double precision arithmetics is represented in *composite precision floating-point numbers*, for which operations are specified. The benefit of alternative floating point arithmetics for ARPS on GPU might be analyzed in future work.

We have shown the differences between neighbor lists and cell lists. In general, for the low levels of activity at which ARPS is beneficial, it is better to use a cell grid (Section 3.3). In [104] and [105], a more thorough comparison is made between neighbor lists (referred to as 'IVT: improved Verlet tables') and the cell-linked list (CLL) which we have used. It is found that in terms of memory, in parallel applications an IVT performs worse than a CLL or an improved CLL, introduced in [106]. The improved CLL approach sorts particles in neighboring cells along an axis through their centers to reduce the number of particle-particle distance computations. While sorting on the GPU is still relatively expensive, the improved CLL approach might be feasible in future work when progress is made in the field of GPU sorting. In [104] the author also points out that the memory requirements for IVT are much larger than those of CLL. These requirements might put a strong cap on the size of the simulated system.

Research on reduction of computing times for short-range non-bonded interactions is still a hot topic in both sequential and parallel applications, with new techniques introduced regularly (e.g. in [106]), aimed mostly at reducing the number of spurious distance computations. In [107] the **Rigid-CLL** method is introduced, which is in some ways similar to the CLL algorithm we use in steps 1.1 through 1.6 in Chapter 3. A system of particles is assumed to contain some rigid structures. Then to avoid unnecessary force or distance computations, interactions within those rigid structures are not computed. In one extreme case, all particles are individual rigid structures and interactions are computed as usual. In the other extreme case, all particles are part of one rigid structure and no interactions are computed. We might think of the set of restrained particles in ARPS as a rigid structure and all active particles as individual rigid structures. Then the algorithm proposed in [107] is very similar to that introduced in Chapter 4 of [40] for ARPS on a CPU. The only difference is the additional old force subtraction step in [40]. The algorithm proposed in Chapter 3 of the current work might be thought of as an SIMD parallel variant of the Rigid-CLL algorithm.

Through a series of experiments, we have shown that ARPS is feasible on the GPU. The force fields in these experiments were restricted to short-range non-bonded interactions and spring-like bonded interactions. In future work on ARPS and ARPS on GPU, applicability to other force fields might be researched, e.g. long-range non-bonded interactions through multi pole methods. Efficient full dynamics implementations for long-range electrostatics are already available and it should not be too complicated to extend these to ARPS [17].

In Chapter 4 we have introduced a method for interactive molecular manipulation along the lines of the methods proposed in [82] and [33], but very different in the sense that our method is based on graphics deformation. In terms of accuracy, the interactive molecular dynamics method introduced in [82] is definitely the best. In terms of speed and interactivity however, our method is advantageous. Using a very simple blending of transformations we can very quickly determine new positions for all particles. The computational caveat is in solving the quadratic program at bind time, which gives the weight functions on the graph. Solving this program with small intervals might cause the program to lose interactivity. However, we have seen that we can solve the program very rapidly by splitting it into several smaller quadratic programs and then combining the results. This allowed us to apply transformations to sheets of graphene, unzip carbon nano-tubes and move parts of a BPTI protein.

Future work on this method might include more accurate force fields and custom quadratic programs. In Chapter 4, we have built on the bounded biharmonic weights introduced in [102], but weights might also be found using other constraints. In some situations it might be beneficial to minimize a custom energy function. Also, accuracy might be improved by treating user-applied transformations not as transformations per se, but more as flexible 'intentions' on the handles. Then, iterating over bind and pose time, the system can move as close as possible to the desired position. Alternatively, we might include energy minimization steps to get a hybrid approach similar to interactive MD, where deformation and simulation steps alternate in order to accurately model displacements.

Other improvements might lie in coarse graining of structures to impose rigidity. Tools for secondary structure detection could be used to group particles in super particles and apply weight functions accordingly. In the examples in Chapter 4 we have chosen an all-atom approach, where structures were rigidified during quadratic problem solving, using some rigidity parameter β . To impose complete rigidity, the coarse graining approach might be preferable.

Future implementations of the BlendProt program could be linked to a haptic feedback device to give the user a more accurate sense of manipulation. Whenever the user applies a transformation to a particle, a force is returned through the device. Also, it would be very interesting to experiment with other input modalities, such as virtual reality or 3-D motion sensing techniques such as the recently announced LEAP device.

In Problem 5 of Chapter 4 we have seen how we can use the BlendProt approach to intervene in a running MD simulation. The MD simulation used was a polymer in solvent in the NVT ensemble, run in a full dynamics GPU implementation. Of course, we could run the MD simulation in ARPS on GPU. This way, we could have run the same simulation faster. In future work, running an MD simulation in ARPS on GPU and preparing or intervening in this simulation using the here presented interactive manipulation program might greatly enhance the time scale of the molecular simulation.

Both ARPS on GPU and the molecular manipulation algorithm offer interesting techniques which might be used together or independently to open new realms for molecular simulations, thereby allowing researchers in chemistry, biology and physics to reach better conclusions faster and spark innovations in e.g. drug design.

Bibliography

- [1] S. Artemova and S. Redon. "Adaptively Restrained Particle Simulations". In: *Physical Review Letters* 109.19 (2012).
- [2] A. Jacobson et al. "Bounded biharmonic weights for real-time deformation". In: *ACM Transactions on Graphics (TOG)* 30.4 (2011), p. 78.
- [3] L. Kavan et al. "Geometric skinning with approximate dual quaternion blending". In: *ACM Transactions on Graphics (TOG)* 27.4 (2008), p. 105.
- [4] D.C. Rapaport. *The art of molecular dynamics simulation*. Cambridge University Press, 2004.
- [5] B.J. Alder and T.E. Wainwright. "Phase transition for a hard sphere system". In: *The Journal of Chemical Physics* 27 (1957), p. 1208.
- [6] B.J. Alder and T.E. Wainwright. "Phase transition in elastic disks". In: *Physical Review* 127.2 (1962), pp. 359–361.
- [7] J.A. McCammon, B.R. Gelin, M. Karplus, et al. "Dynamics of folded proteins". In: *Nature* 267.5612 (1977), p. 585.
- [8] H. Ode et al. "Frontiers: Molecular dynamics simulation in virus research". In: *Frontiers in Virology* 3 (2012).
- [9] A. Perez, F.J. Luque, and M. Orozco. "Frontiers in molecular dynamics simulations of DNA". In: *Accounts of Chemical Research* 45.2 (2011), pp. 196–205.
- [10] J.D. Durrant and J.A. McCammon. "Molecular dynamics simulations and drug discovery". In: *BMC biology* 9.1 (2011), p. 71.
- [11] D.E. Shaw et al. "Anton, a special-purpose machine for molecular dynamics simulation". In: *ACM SIGARCH Computer Architecture News*. Vol. 35. 2. ACM. 2007, pp. 1–12.
- [12] M. Taiji. "MDGRAPE-3 chip: a 165 Gflops application specific LSI for molecular dynamics simulations". In: *Hot Chips*. Vol. 16. 2004, pp. 198–202.
- [13] Top500. *Top 500 Supercomputer Sites*. Retrieved August 22, 2012. 2012. url: <http://www.top500.org/>.
- [14] Folding@Home. *Folding@Home Statistics*. 2012. url: <http://fah-web.stanford.edu/cgi-bin/main.py?ctype=osstats>.
- [15] J.A. Anderson, C.D. Lorenz, and A. Travesset. "General purpose molecular dynamics simulations fully implemented on graphics processing units". In: *Journal of Computational Physics* 227.10 (2008), pp. 5342–5359.
- [16] J.A. van Meel et al. "Harvesting graphics power for MD simulations". In: *Molecular Simulation* 34.3 (2008), pp. 259–266.
- [17] J.A. Baker and J.D. Hirst. "Molecular dynamics simulations using graphics processing units". In: *Molecular Informatics* 30.6-7 (2011), pp. 498–504.
- [18] D.C. Rapaport. "Enhanced molecular dynamics performance with a programmable graphics processor". In: *Computer Physics Communications* 182.4 (2011), pp. 926–934.
- [19] W.M. Brown et al. "Implementing molecular dynamics on hybrid high performance computers—short range forces". In: *Computer Physics Communications* 182.4 (2011), pp. 898–911.
- [20] AMD Radeon HD7970 Specifications. 2012. url: <http://www.amd.com/us/products/desktop/graphics/7000/7970/>.

- [21] A.W. Götz et al. "Routine microsecond molecular dynamics simulations with AMBER on GPUs. 1. generalized Born". In: *Journal of Chemical Theory and Computation* 8.5 (2012), p. 1542.
- [22] Wikipedia. *Comparison of Nvidia graphics processing units — Wikipedia, The Free Encyclopedia*. [Online; accessed 31-August-2012]. 2012. url: http://en.wikipedia.org/wiki/Comparison_of_Nvidia_graphics_processing_units.
- [23] A. Vladimirov. *Arithmetics on Intels Sandy Bridge and Westmere CPUs: not all FLOPs are created equal*. 2012.
- [24] *The Research Collaboratory for Structural Bioinformatics PDB*. 2012. url: <http://www.rcsb.org/pdb/>.
- [25] D.E. Shaw et al. "Millisecond-scale molecular dynamics simulations on ANTON". In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. SC '09. New York, NY, USA: ACM, 2009, 39:1–39:11.
- [26] D.E. Shaw et al. "Atomic-Level Characterization of the Structural Dynamics of Proteins". In: *Science* 330.6002 (2010), pp. 341–346.
- [27] M. Vendruscolo and C.M. Dobson. "Protein Dynamics: Moore's Law in Molecular Biology". In: *Current Biology* 21.2 (2011), R68–R70.
- [28] G.E. Moore. "Cramming more components onto integrated circuits". In: *Electronics* 38.8 (1965).
- [29] S. Kmiecik, M. Jamroz, and A. Kolinski. "Multiscale Approach to Protein Folding Dynamics". In: *Multiscale Approaches to Protein Modeling* (2011), pp. 281–293.
- [30] J. Janin et al. "CAPRI: a critical assessment of predicted interactions". In: *Proteins: Structure, Function, and Bioinformatics* 52.1 (2003), pp. 2–9.
- [31] YASARA - *Yet Another Scientific Artificial Reality Application*. 2012. url: <http://www.yasara.org>.
- [32] LLC Schrödinger. "The PyMOL Molecular Graphics System, Version 1.3r1". 2010.
- [33] S. Crivelli et al. "ProteinShop: a tool for interactive protein manipulation and steering". In: *Journal of Computer-aided Molecular Design* 18.4 (2004), pp. 271–285.
- [34] D. Frenkel and B. Smit. *Understanding molecular simulation: from algorithms to applications*. Vol. 1. Academic Pr, 2002.
- [35] P.V.R Schleyer et al. "CHARMM: The Energy Function and Its Parameterization with an Overview of the Program". In: *The Encyclopedia of Computational Chemistry* 1 (1998), pp. 271–277.
- [36] D.A. Case et al. "AMBER 11". In: *University of California, San Francisco* (2010).
- [37] M.P. Allen and D.J. Tildesley. *Computer simulation of liquids*. Vol. 18. 195. Oxford university press, 1989.
- [38] E. Hairer, C. Lubich, and G. Wanner. *Geometric numerical integration: structure-preserving algorithms for ordinary differential equations*. Vol. 31. Springer Verlag, 2006.
- [39] R. de Vogelaere. "Methods of integration which preserve the contact transformation property of the Hamiltonian equations. Report Nr. 4, Dept". In: *Department of Mathematics, University of Notre Dame, Notre Dame, Indiana* (1956).
- [40] S. Artemova. "Adaptive algorithms for molecular simulation". PhD thesis. Université de Grenoble, 2012.
- [41] T. Lelièvre, G. Stoltz, and M. Rousset. *Free energy computations: A mathematical perspective*. World Scientific, 2010.
- [42] W. Wang and D.S. ROBERT. "Analysis of a few numerical integration methods for the Langevin equation". In: *Molecular Physics* 101.14 (2003), pp. 2149–2156.
- [43] M. E. J. Newman and G. T. Barkema. *Monte Carlo Methods in Statistical Physics*. Oxford University Press, USA, 1999.
- [44] *Abalone - Software for bio-molecular modeling*. 2012. url: <http://www.biomolecular-modeling.com/Abalone/>.

- [45] M.J. Harvey, G. Giupponi, and G.D. Fabritiis. "ACEMD: accelerating biomolecular dynamics in the microsecond time scale". In: *Journal of Chemical Theory and Computation* 5.6 (2009), pp. 1632–1639.
- [46] W. Smith, I.T. Todorov, and M. Leslie. "The DL_POLY molecular dynamics package". In: *Zeitschrift für Kristallographie* 220.5/6/2005 (2005), pp. 563–566.
- [47] B. Hess et al. "GROMACS 4: Algorithms for highly efficient, load-balanced, and scalable molecular simulation". In: *Journal of chemical theory and computation* 4.3 (2008), pp. 435–447.
- [48] *Highly Optimized Object-oriented Many-particle Dynamics – Blue Edition*. 2012. url: <http://codeblue.umich.edu/hoomd-blue/>.
- [49] S. Plimpton. "Fast parallel algorithms for short-range molecular dynamics". In: *Journal of Computational Physics* 117.1 (1995), pp. 1–19.
- [50] *LAMPSS Molecular dynamics simulator*. 2012. url: <http://lammmps.sandia.gov/>.
- [51] J.C. Phillips et al. "Scalable molecular dynamics with NAMD". In: *Journal of Computational Chemistry* 26.16 (2005), pp. 1781–1802.
- [52] M.S. Friedrichs et al. "Accelerating molecular dynamic simulation on graphics processing units". In: *Journal of Computational Chemistry* 30.6 (2009), pp. 864–872.
- [53] *OpenCL - The open standard for parallel programming of heterogeneous systems*. 2012. url: www.khronos.org/opencl/.
- [54] *GPU Computing Gems Emerald Edition (Applications of GPU Computing Series)*. 1st ed. Morgan Kaufmann, 2011.
- [55] *CUDA CUBLAS Library*. nVidia Corporation. 2010. url: http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUBLAS_Library.pdf.
- [56] J. Hoberock and N. Bell. *Thrust: A Parallel Template Library*. Version 1.3.0. 2010. url: <http://www.meganevtons.com/>.
- [57] G. Khanna and J. McKenon. "Numerical modeling of gravitational wave sources accelerated by OpenCL". In: *Computer Physics Communications* 181.9 (2010), pp. 1605–1611.
- [58] M. Feldman. *OpenCL gains ground on CUDA*. Retrieved September 12, 2012. 2012. url: http://www.hpcwire.com/hpcwire/2012-02-28/opencl_gains_ground_on_cuda.html.
- [59] J. Weinbub, K. Rupp, and S. Selberherr. "Towards distributed heterogenous high-performance computing with ViennaCL". In: *Large-Scale Scientific Computing* (2012), pp. 359–367.
- [60] B. Gaster et al. *Heterogeneous Computing with OpenCL*. Morgan Kaufmann, 2011.
- [61] S.J. Marrink et al. "The MARTINI force field: coarse grained model for biomolecular simulations". In: *The Journal of Physical Chemistry B* 111.27 (2007), pp. 7812–7824.
- [62] J.A. Izaguirre, S. Reich, and R.D. Skeel. "Longer time steps for molecular dynamics". In: *The Journal of Chemical Physics* 110 (1999), p. 9853.
- [63] L. Greengard and V. Rokhlin. "A fast algorithm for particle simulations". In: *Journal of Computational Physics* 73.2 (1987), pp. 325–348.
- [64] S. Grudinin and S. Redon. "Practical modeling of molecular systems with symmetries". In: *Journal of Computational Chemistry* 31.9 (2010), pp. 1799–1814.
- [65] S. Redon, N. Galoppo, and M.C. Lin. "Adaptive dynamics of articulated bodies". In: *ACM Transactions on Graphics (TOG)*. Vol. 24. 3. ACM. 2005, pp. 936–945.
- [66] S. Redon and M.C. Lin. "An efficient, error-bounded approximation algorithm for simulating quasi-statics of complex linkages". In: *Computer-Aided Design* 38.4 (2006), pp. 300–314.
- [67] Changjun H., Yali L., and Jianjiang L. "Efficient Parallel Implementation of Molecular Dynamics with Embedded Atom Method on Multi-core Platforms". In: *Parallel Processing Workshops, 2009. ICPPW '09. International Conference on*. 2009, pp. 121–129.
- [68] B.A. Bauer et al. "Molecular dynamics simulations of aqueous ions at the liquid–vapor interface accelerated using graphics processors". In: *Journal of Computational Chemistry* 32.3 (2011), pp. 375–385.

- [69] M. Taufer et al. "Improving numerical reproducibility and stability in large-scale numerical simulations on GPUs". In: *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE. 2010, pp. 1–9.
- [70] P. LeBrun et al. *Recent progress in accelerator physics simulations with the VORPAL code*. Tech. rep. 2010.
- [71] E. Westphal. "Multiparticle Collision Dynamics on GPUs". GPU Technology Conference 2012. 2012.
- [72] C.R. Trott, L. Winterfeld, and P.S. Crozier. "General-purpose molecular dynamics simulations on GPU-based clusters". In: *Arxiv preprint arXiv:1009.4330* (2010).
- [73] X Tian and K. Benkrid. "Mersenne Twister Random Number Generation on FPGA, CPU and GPU". In: *Adaptive Hardware and Systems, 2009. AHS 2009. NASA/ESA Conference on*. 2009, pp. 460–464.
- [74] C.L. Phillips, J.A. Anderson, and S.C. Glotzer. "Pseudo-random number generation for Brownian Dynamics and Dissipative Particle Dynamics simulations on GPU devices". In: *Journal of Computational Physics* 230.19 (2011), pp. 7191–7201.
- [75] B.G. Levine, J.E. Stone, and A. Kohlmeyer. "Fast analysis of molecular dynamics trajectories with graphics processing units: Radial distribution function histogramming". In: *Journal of Computational Physics* 230.9 (2011), pp. 3556–3569.
- [76] C. Gregg and K. Hazelwood. "Where is the data? Why you cannot debate CPU vs. GPU performance without the answer". In: *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*. IEEE. 2011, pp. 134–144.
- [77] A. Zhmurov et al. "SOP-GPU: Accelerating biomolecular simulations in the centisecond timescale using graphics processors". In: *Proteins: Structure, Function, and Bioinformatics* 78.14 (2010), pp. 2984–2999.
- [78] Y. He and C.H.Q. Ding. "Using accurate arithmetics to improve numerical reproducibility and stability in parallel applications". In: *Proceedings of the 14th international conference on Supercomputing*. ACM. 2000, pp. 225–234.
- [79] L. Nyland, M. Harris, and J. Prins. "Fast n-body simulation with cuda". In: *GPU gems* 3 (2007), pp. 677–695.
- [80] G. Schaftenaar and J.H. Noordik. "Molden: a pre-and post-processing program for molecular and electronic structures". In: *Journal of Computer-aided Molecular Design* 14.2 (2000), pp. 123–134.
- [81] W. Humphrey, A. Dalke, and K. Schulten. "VMD – Visual Molecular Dynamics". In: *Journal of Molecular Graphics* 14 (1996), pp. 33–38.
- [82] J.E. Stone, J. Gullingsrud, and K. Schulten. "A system for interactive molecular dynamics simulation". In: *Proceedings of the 2001 symposium on Interactive 3D graphics*. I3D '01. New York, NY, USA: ACM, 2001, pp. 191–194.
- [83] J. Stone et al. "Immersive molecular visualization and interactive modeling with commodity hardware". In: *Advances in Visual Computing* (2010), pp. 382–393.
- [84] M. Botsch et al. "Adaptive space deformations based on rigid cells". In: *Computer Graphics Forum*. Vol. 26. 3. Wiley Online Library. 2007, pp. 339–347.
- [85] M. Nesme et al. "Preserving topology and elasticity for embedded deformable models". In: *ACM Transactions on Graphics (TOG)*. Vol. 28. 3. ACM. 2009, p. 52.
- [86] M. Müller et al. "Point based animation of elastic, plastic and melting objects". In: *Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*. Eurographics Association. 2004, pp. 141–151.
- [87] M. Müller et al. "Meshless deformations based on shape matching". In: *ACM Transactions on Graphics (TOG)*. Vol. 24. 3. ACM. 2005, pp. 471–478.
- [88] Y. Lipman, D. Levin, and D. Cohen-Or. "Green coordinates". In: *ACM Transactions on Graphics (TOG)*. Vol. 27. 3. ACM. 2008, p. 78.

- [89] N. Faraj, J.M. Thiery, and T. Boubekeur. "3-scale freeform deformation of large voxel grids". In: *Computers & Graphics* (2012).
- [90] J.R. Nieto and A. Susin. "Cage based deformations: a survey". In: (2013).
- [91] J.P. Lewis, M. Cordner, and N. Fong. "Pose space deformation: a unified approach to shape interpolation and skeleton-driven deformation". In: *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. ACM Press/Addison-Wesley Publishing Co. 2000, pp. 165–172.
- [92] J.D. Foley et al. *Introduction to computer graphics*. Vol. 55. Addison-Wesley, 1994.
- [93] E. Pennestrì and P.P. Valentini. "Dual quaternions as a tool for rigid body motion analysis: a tutorial with an application to biomechanics". In: *Archive of Mechanical Engineering 2* (2010), pp. 187–205.
- [94] S. Artemova, S. Grudinin, and S. Redon. "Fast construction of assembly trees for molecular graphs". In: *Journal of Computational Chemistry* 32.8 (2011), pp. 1589–1598.
- [95] W. Kabsch and C. Sander. "Dictionary of protein secondary structure: pattern recognition of hydrogen-bonded and geometrical features". In: *Biopolymers* 22.12 (1983), pp. 2577–2637.
- [96] L.J. Grady and J.R. Polimeni. *Discrete Calculus: Applied Analysis on Graphs for Computational Science*. Springer, 2010.
- [97] MATLAB. *version 7.14.0 (R2012a)*. Natick, Massachusetts: The MathWorks Inc., 2010.
- [98] J.D.M. Andersen and L. Vandenberghe. *CVXOPT: Python Software for Convex Optimization*. Version 1.1.5. 2012. url: <http://abel.ee.ucla.edu/cvxopt/index.html>.
- [99] T. Ragab and C. Basaran. "The unravelling of open-ended single walled carbon nanotubes using molecular dynamics simulations". In: *Journal of Electronic Packaging* 133.2 (2011).
- [100] D.W. Brenner et al. "A second-generation reactive empirical bond order (REBO) potential energy expression for hydrocarbons". In: *Journal of Physics: Condensed Matter* 14.4 (2002), p. 783.
- [101] O. Weber, R. Poranne, and C. Gotsman. "Biharmonic Coordinates". In: *Computer Graphics Forum*. Wiley Online Library. 2012.
- [102] A. Jacobson, T. Weinkauff, and O. Sorkine. "Smooth Shape-Aware Functions with Controlled Extrema". In: *Computer Graphics Forum*. Vol. 31. 5. Wiley Online Library. 2012, pp. 1577–1586.
- [103] S.L. Grand, A.W. Götzx, and R.C. Walker. "SPFP: speed without compromise: a mixed precision model for GPU accelerated molecular dynamics simulations". In: *Computer Physics Communications* (2012).
- [104] E.S. Fomin. "Comparison of the Verlet table and cell-linked list algorithms on parallel architectures". In: *Numerical Methods and Programming* 11 (2010), p. 299.
- [105] E.S. Fomin. "Consideration of data load time on modern processors for the Verlet table and linked-cell algorithms". In: *Journal of Computational Chemistry* 32.7 (2011), pp. 1386–1399.
- [106] P. Gonnet. "A simple algorithm to accelerate the computation of non-bonded interactions in cell-based molecular dynamics simulations". In: *Journal of Computational Chemistry* 28.2 (2007), pp. 570–573.
- [107] V.R. de Angulo, J. Cortés, and JM Porta. "Rigid-CLL: Avoiding constant-distance computations in cell linked-lists algorithms". In: *Journal of Computational Chemistry* (2012).