Ilmenau University of Technology
Faculty of Mathematics and Natural Science
Institute of Physics
Theoretical Physics II / Computational Physics

Accelerating the Molecular Dynamics Program LAMMPS using Graphics Cards' Processors and the Nvidia Cuda Technology

Bachelor Thesis to achieve the academic degree Bachelor of Science

Lars Winterfeld

Supervising Professor:
Univ.-Prof. Dr. rer. nat. habil. Erich Runge,
Department for Theoretical Physics I

Advisor:
Dipl.-Ing. Christian Robert Trott,
Department for Theoretical Physics II

Ilmenau, August 11, 2010

urn:nbn:de:gbv:ilm1-2010200267

Acknowledgement

I would like to thank everybody who supported me during the creation of this paper, especially Stefan Brechtken for proofreading. Special thanks goes to the adviser of this work, Christian Robert Müller, who was unstinting of his time and answered questions outside of usual working times.

Erklärung: Hiermit versichere ich, dass ich diese Bachelorarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe. Alle von mir aus anderen Veröffentlichungen übernommenen Passagen sind als solche gekennzeichnet.

Ilmenau, August 24, 2010 .....
Lars Winterfeld

Abstract in English:

This bachelor thesis discusses the possibilities and limits of extending the widely-used Molecular Dynamics program LAMMPS (the Large Scale Atomic/Molecular Massive Parallel Simulator) in order to accelerate simulations by making use of modern graphic processing units in the framework of NVIDIA's Cuda technology. An own implementation serves as basis for the discussion. The first chapters summerizes selected aspects of Molecular Dynamics, LAMMPS' design and the Cuda technology for graphics card programming. On that basis, details of the above mentioned implementation are discussed, comparing different force calculation strategies and showing methods to avoid frequent data transfers from and to the graphics card. The following part of the work deals with benchmarks and the influence of factors like numerical precision and system size in typical Molecular Dynamics simulations, always comparing the results to the original version of LAMMPS with no graphics card support. Finally, conclusions of this work are presented and the implementation is compared with other similar projects.
The document is written in English.

Zusammenfassung in Deutsch:

Diese Bachelorarbeit diskutiert die Möglichkeiten und Grenzen einer Erweiterung des weit verbreiteten Molekulardynamik-Programms LAMMPS (dem Large Scale Atomic/Molecular Massive Parallel Simulator), in welcher Simulationen durch das Einbeziehen moderner Grafikkarten - im Rahmen von NVIDIAs Cuda Technologie - beschleunigt werden. Eine eigene Implementierung dient als Grundlage der Diskussion. Die ersten Kapitel wiederholen ausgewählte Aspekte der Molekulardynamik, Lammps und der Cuda Technologie zur Grafikkartenprogrammierung. Darauf aufbauend werden Details der vorgenannten Implementierung diskutiert, wobei verschiedene Strategien zur Kraftberechnung verglichen werden und Methoden zur Vermeidung eines häufigen Datentransfers von und zur Grafikkarte aufgezeigt werden. Der darauf folgende Teil der Arbeit beschäftigt sich mit Geschwindigkeitstests und dem Einfluss von Faktoren wie numerischer Genauigkeit und Systemgröße in typischen Simulationen der Molekulardynamik, wobei die Ergebnisse immer mit jenen der ursprünglichen LAMMPS-Version verglichen werden, die über keine Grafikkartenunterstützung verfügt. Abschließend werden Schlussfolgerungen aus dieser Arbeit vorgestellt und die Implementierung wird mit anderen, ähnlichen Projekten verglichen.
Die Arbeit ist in englischer Sprache verfasst.

Lists and Definitions

List of symbols

Symbol	Meaning, if not explicitly differently declared
$\mathbb{R}$	the set of real numbers
$\mathbb{Z}$	the set of integers
$\vec{E}$	electric field
$\vec{F}$	force
$\vec{r}$	position or distance
$\vec{v}$	velocity
d	dimensionality
I	number of instructions
k_B	Boltzmann constant
m	mass
N	the number of processes
n	total number of particles
Φ	virial
p	pressure
q	charge
S	speed-up
T	temperature
t	time
U	potential
V	volume

List of abbreviations

Abbreviation	Meaning
2D	two-dimensional
3D	three-dimensional
API	Application Programming Interface (set of documented functions)
approx.	approximately
CPU	Central Processing Unit
C	the programming language C
C++	the programming language C++
CUDA	NVIDIA's Compute Unified Device Architecture
FSB	Front Side Bus
GiB	GigaByte, 1 GiB = 1024 MiB
GiB / s	GigaByte per second
GPU	Graphics Processing Unit
KiB	KiloByte, 1 KiB = 1024 Byte
LJ	Lennard-Jones
LAMMPS	Large Scale Atomic/Molecular Massive Parallel Simulator
MiB	MegaByte, 1 MiB = 1024 KiB
MD	Molecular Dynamics
MP	MultiProcessor
MPI	Message Passing Interface (a communication API)
NL	Neighbour List
PC	Personal Computer
PCI	Peripheral Component Interconnect (computer interface for peripherals)
PCIe	PCI Express
PPPM	Particle-Particle, Particle-Mesh
prec.	precision
RAM	Random Access Memory
RDF	Radial Distribution Functions
SIMD	Single Instruction, Multiple Data
SIMT	Single Instruction, Multiple Thread
SISD	Single Instruction, Single Data
STL	Standard Template Library (part of the C++ standard)

List of mathematical definitions

To avoid potential misunderstandings, some (mathematical) conventions used in this work should be repeated.

(a) $\mathbb{N}$ are the natural numbers, as defined by the Peano axioms, starting with 0.

(b) $\mathbb{N}^+ := \mathbb{N} \setminus \{0\}$ are the natural number except 0.

(c) Consistent with the C programming language, the elements of a set M, $|M| = N$, and components of a vector $\vec{x}$, $\vec{x} \in \mathbb{R}^N$, are indexed from 0 to $N-1$.

(d) The notation $\langle \cdot, \cdot \rangle$ defines the usual scalar product.

$$\langle \vec{x}\,,\, \vec{y} \rangle := \sum_{i=0}^{d-1} x_i\, y_i$$

(e) Unless explicitly defined otherwise, for any vector $\vec{x} \in \mathbb{R}^N$, the same symbol without arrow, x, should denote its Euclidean norm.

$$x := |\vec{x}| := \sqrt{\langle \vec{x}\,,\, \vec{x} \rangle}$$

(f) In this work, the *big-$\mathcal{O}$* notation is defined as follows:
if f and g are functions $f, g : \mathbb{N} \to \mathbb{R}$, then:

$$f \in \mathcal{O}(g) \;:\Leftrightarrow\; \exists\, c \in \mathbb{R} :\; 0 \leq \limsup_{n \to \infty} \left| \frac{f(n)}{g(n)} \right| \leq c\,.$$

The usual conventions for this notation are used, e.g., a program is said to "consume $\mathcal{O}(n^2)$ memory", if the function $f : \mathbb{N} \to \mathbb{N}$ describing the memory amount needed (depending on an n clearly defined by the context) is $f \in \mathcal{O}(g)$, $g : \mathbb{N} \to \mathbb{N}$, $g(n) = n^2$.

(g) The symbol $\propto$ means "directly proportional".
If f and g are functions $f, g : \mathbb{D}_f \to \mathbb{R}$, $\mathbb{D}_f \subseteq \mathbb{R}$, then

$$f \propto g \;:\Leftrightarrow\; \exists\, c \in \mathbb{R} :\; \forall t \in \mathbb{D}_f :\; f(t) = c\, g(t)\,.$$

(h) The operation `mod` means modulo. Let $a, b, m \in \mathbb{N}$, then

$$a \bmod m = b \;\;:\Leftrightarrow\; \exists\, z \in \mathbb{Z} : a = b + z \cdot m.$$

(i) The notation $\lceil r \rceil$ denotes the ceiling function. If $r \in \mathbb{R}$, then $\lceil r \rceil := \min\{z \in \mathbb{Z} | z \geq r\}$.

(j) When an item is indexed by multiple indices, this work sometimes denotes the tuple $\vec{i} \in \mathbb{N}^d$ with an vector arrow, knowing $\mathbb{N}^d$ is not a vector space in the mathematical sense of the term.

Contents

Chapter 1

Introduction

> "Given for one instant an intelligence which could comprehend all forces by which nature is animated and the respective situation of the beings who compose it - an intelligence sufficiently vast to submit these data to analysis - it would embrace in the same formula the movements of the greatest bodies of the universe and those of the lightest atoms; for it, nothing would be uncertain and the future, as the past, would be present to its eyes."
>
> Pierre-Simon Laplace in 1814 [1]

Molecular Dynamics (MD) is a method of computational physics which allows to study properties and behaviour of complex systems like gases, liquids and solids by calculating the motion of every particle in the system over a given time. While the atomistic idea to solve the many-body problem numerically following Newton's equations is not exactly new, the wide range of applications and systems approachable by MD has gained new attention with the availability of modern powerful computers. On the one hand, MD aids fundamental studies of statistical physics concerning, e.g., kinetic theory, fluid dynamics, transport theory or phase transitions by providing an "experiment on the computer". On the other hand, more applicative scenarios in material sciences, polymer physics, biology or chemistry are also feasible by means of MD. As there are established theories for solids and delude gases, MD is usually used for dense systems, such as liquids or glasses. There is no known experimental method which could present that level of microscopic detail for all particles in a studied system at once.

Since MD is a rather straight-forward method that - for a defined interaction - does not rely on simplifications or an abstracted model, there are also limitations of this method that come along with the advantages. Because of the limited computer memory, calculation speed and time one is willing to wait for the result, systems accessible by MD are limited in size (typically $10^5 \dots 10^6$ particles) and time (typically pico or nano seconds). Therefore, it is always interesting to find ways to accelerate the computational-intensive MD simulation. A usual approach to do so is to involve more processors into the calculation and parallelise the simulation. While it may be expensive to set up a cluster of multiple computers, the last years have revealed another alternative of parallelization, namely to make use of the processors of a graphics card (GPUs). Graphics cards have not only turned into powerful computational units being able to generate high quality three-dimensional images in real time, their potential is also wasted most of the time while the computer is not running a graphic-intense application. That is why this work aims at discussing the possibilities and limits of accelerating an MD program by graphics cards' processors. While there are quite a few programs which already use the GPU for special MD purposes, the more specialised those programs are, the more they lack the possibility to extend their scope of approachable scenarios into the wide range of general MD problems. Therefore, the discussion in this document will be based on own (already published [2]) implementation extending an existing full-featured program. The programming part was developed in a team consisting

of Christan Müller and myself (Lars Winterfeld). A list of modules is given along with their main contributor in the appendix. Nevertheless, this work will describe all parts of the programming project.

The MD program chosen is LAMMPS, the Large Scale Atomic/Molecular Massive Parallel Simulator - key aspects of it are described in the second chapter after a repetition of main ideas and algorithms of MD being relevant for this work. A special programming interface is needed to access a GPU; this document limits itself to one such interface: the Compute Unified Device Architecture (CUDA) introduced by the company Nvidia (a chipset and graphics card producer). While programming details on the level of interface specifications are less interesting for this work, the third chapter will give an overview of concepts of GPU programming serving as a basis for the following discussion. A reader already familiar with the topic could skip those preceding chapters. The fourth chapter will then combine the ideas of the preceding chapters to explain the implementation decisions, also discussing the effects on performance. A more detailed benchmark in different scenarios is given in the fifth chapter also investigating the influence of different calculating precisions, right before the sixth chapter compares the implementation to other GPU MD projects and the seventh chapter sums up the conclusions of this work. Finally, the appendix lists some algorithm remarks and other facts, which would have interrupted the text flow in the main part.

Chapter 2

Molecular Dynamics Simulation

2.1 The method of MD

As mentioned in the introduction, Molecular Dynamics (MD) is a method of computational physics which solves the classical n-body problem numerically in discrete time steps. Ideally, a complete MD simulation requires three steps [3]:

(1^{st}) model an initial start scenario,

(2^{nd}) compute the movements of the individual particles and

(3^{rd}) analyse the simulation data for the desired physical properties.

Since the aim of this work is the acceleration of MD, the discussion is generally focused on the second step (the simulation itself). In particular, this chapter intends to repeat some aspects of MD simulations which are relevant for the later discussion, starting with typical pair forces. Afterwards, integration algorithms and the idea of neighbour lists are explained, followed by some words on parallelization and communication in MD programs. Finally, elements of the structure of the MD program LAMMPS are outlined, because the already mentioned implementation is based on it.

2.2 Typical forces and force calculation methods

2.2.1 Typical pair forces

In every time-step the forces of the particles on each other need to be calculated in order to get the desired interacting system. Forces can be categorised into *bounded* and *non-bounded* interactions. Bounded forces are introduced to bring the quantum mechanical bond of a molecule into the classical simulation scheme, while non-bounded forces, in contrast, can be understood more classical - they are not depending on a specific bounding direction. For non-bounding forces, *short-range* interactions are set apart from *long-range* interactions, because of a different treatment in MD simulations. In a system of dimensionality d (usually $d = 3$), a potential U, depending on the distance $r := |\vec{r}|$, should be called short-range, if and only if

$$\int_0^\infty U(r)\, r^{d-1}\, \mathrm{dr} < \infty \,, \tag{2.1}$$

and long-range otherwise.

A typical example for a long-range interaction is the *Coulomb* potential U_{col} of a particle with the charge q_1. Another particle with charge q_2 in distance $\vec{r}$ in the field of of the first one will see U_{col} (with ε_0 and ε_r being constants):

$$U_{col}(r) = -4\pi\,\varepsilon_0\varepsilon_r\,q_1q_2\,\frac{1}{r}\ . \tag{2.2}$$

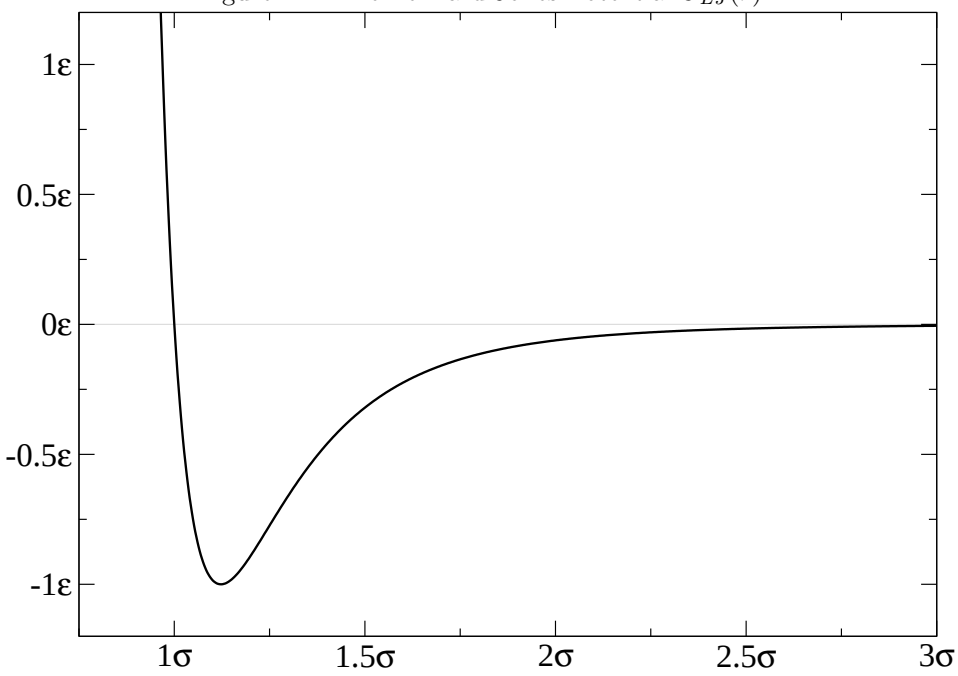
The gradient of the potential implies the Coulomb force $\vec{F}_{col}$:

$$\vec{F}_{col}(\vec{r}) = -\vec{\nabla}U_{col}(r) = 4\pi\,\varepsilon_0\varepsilon_r\,q_1q_2\frac{\vec{r}}{r^3}\ .$$

While the Coulomb potential U_{col} decreases slowly ($\propto r^{-1}$) with the distance r, there are other forces having their main contribution only in short ranges. A typical example for such a force on non-charged particles was proposed in 1924 by John Lennard-Jones and is named after him. It combines the ideas of an attractive van-der-Waals force being $\propto r^{-6}$ and the repulsive force between the electron shells of the particles. The latter is approx. $\propto e^{-\frac{r}{r_0}}$ for some radius r_0, but was approximated to be $\propto r^{-12}$, because it can be easier calculated in computer simulations (by squaring r^{-6}). The *Lennard-Jones potential* (see fig. 2.1) has two parameters, which can be fitted by experimental data or quantum mechanical calculations: ε (an energy marking the depth of the potential) and σ (a length proportional to the equilibrium radius):

$$U_{LJ}(r) = 4\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right]\ . \tag{2.3}$$

Figure 2.1: The Lennard-Jones Potential $U_{LJ}(r)$



Therefore, the force $\vec{F}_{LJ}$ between two particles in distance $\vec{r}$ is obviously:

$$\vec{F}_{LJ}(\vec{r}) = -\vec{\nabla} U_{LJ}(r) = 4\varepsilon \left[12 \left(\frac{\sigma}{r}\right)^{12} - 6 \left(\frac{\sigma}{r}\right)^{6} \right] \frac{\vec{r}}{r^2} \, .$$

To save calculation time, the influence of short-range potentials is usually neglected for distances r greater than a *cut-off radius* r_c:

$$U_{LJ,cut}(r) = \begin{cases} U_{LJ}(r) & \text{for } r \leq r_c \\ 0 & \text{otherwise} \end{cases} \tag{2.4}$$

$$\vec{F}_{LJ,cut}(\vec{r}) = \begin{cases} \vec{F}_{LJ}(\vec{r}) & \text{for } r \leq r_c \\ 0 & \text{otherwise} \end{cases} \, .$$

2.2.2 Neighbour lists

Since every of the n particles needs to interact with every other particle in a simulation step, there are

$$\binom{n}{2} = \frac{n(n-1)}{2} \in \mathcal{O}(n^2)$$

forces to be calculated. Especially for short range potentials with cut-off radii, $\mathcal{O}(n^2)$ interactions have to be checked, but only few of them actually make a contribution. Therefore, MD simulators usually implement a neighbour list strategy for these forces, i.e. a list of particles within the distance r_{neigh} (of, e.g., $r_{neigh} = 2\,r_c$) is kept for every particle for some time-steps. The slower the interaction sphere of the particles move out of that radius r_{neigh}, the longer can the neighbour list be kept. Assuming a fixed r_{neigh} and that the macroscopic density of the particles has an upper bound, there can only be a limited number of particles within the neighbour list radius for each of the n particles, thus, reducing the complexity of pure force computation to $\mathcal{O}(n)$. While this consumes $\mathcal{O}(n)$ extra memory, it is of course a better method than the naive $\mathcal{O}(n^2)$ one for a sufficient large number of particles.

Figure 2.2:

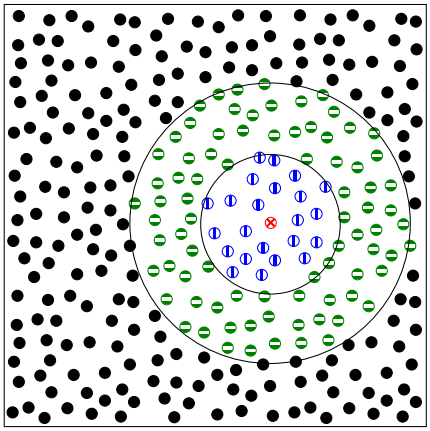


Figure 2.2 illustrates the neighbour list for one of all particles (crossed red) drafted in 2D: the interaction sphere is coloured in vertically striped / blue, the additional neighbour particles are horizontally striped / green.

2.2.3 PPPM and Ewald sum

For long-range potentials, the influence of particles in any distance should be taken into account, while it is still desired to avoid the direct calculation of every particle pair (needing $\mathcal{O}(n^2)$). One method to solve this dilemma is the Ewald summation, whose main idea is to rearrange the sum over all interactions into concentric spherical shells, for which charge neutrality is assumed. The interactions within a chosen radius are calculated directly, while the sum over all interactions outside that radius is Fourier-transformed (this implicitly assumes periodicity of the system). In reciprocal space, this sum converges more quickly and can be calculated in a chosen accuracy. This ansatz reduces the computational complexity to $\mathcal{O}(n^{1.5})$. [4]

Another method to handle long-range potentials is the Particle-Particle Particle-Mesh (PPPM) algorithm. Like the Ewald sum, it splits the interaction into a short-distance (PP) and a long-distance part (PM). However, the long-distance interactions are transformed from the current particle positions onto a regular grid - e.g. for Coulomb interactions this means to create a Cartesian mesh of effective charges. Each charge is distributed along a specific number of closest mesh point in each direction (conserving the total charge). While this approximation is associated with some calculation error [5], the regular periodic grid allows to do a fast Fourier transformation (FFT), which is used to calculate the electric field $\vec{E}$, for which the force is straight-forward to calculate. When choosing the internal parameters of this algorithm right, the run-time complexity reduces to $\mathcal{O}(\, n \cdot log(n)\,)$ [6] and is therefore faster than the Ewald summation (relying on the analytical Fourier transformation). [7]

2.3 Typical integration algorithms

The fundamental equations of mechanics $\frac{d\vec{v}}{dt} = \frac{\vec{F}}{m}$ and $\frac{d\vec{r}}{dt} = \vec{v}$ are used to numerically integrate the forces $\vec{F}$ to velocities $\vec{v}$ and positions $\vec{r}$ in every timestep for each particle of mass m. From the various possible numerically methods, MD simulators usually pick the *(velocity-)Verlet algorithm* [8], whose numerical truncation error varies as the fourth power of the timestep Δt and is therefore said to be a third order method:

$$\vec{r}(t+\Delta t) := 2\,\vec{r}(t) - \vec{r}(t-\Delta t) + \frac{\vec{F}}{m}\Delta t^2 \tag{2.5}$$

$$\vec{v}\left(t+\frac{\Delta t}{2}\right) := \frac{\vec{r}(t-\Delta t) - \vec{r}(t)}{\Delta t} \tag{2.6}$$

Although of rather low order, the verlet algorithm has some advantages over multi-step Runge-Kutta methods or predictor-corrector algorithms concerning runtime and memory usage: it does neither require to evaluate the forces more than once per timestep nor must positions and velocities be kept over a number of timesteps. In compliance with Newton's equations, the Verlet algorithm is also non-variant under time reversion. To justify the numerical inaccuracy of the Verlet algorithm in comparison to other methods, the errors could be alleged to be a wanted representation of uncontrolled fluctuations occurring in real systems, such as small mechanical vibrations in a lab.

The above algorithm keeps the total energy of the system constant, which models an micro-canonical ensemble (NVE) in case the total volume of the system V and the number of particles N are also kept constant. When modelling different ensembles, other quantities than the energy (like temperature T for the canonical NVT ensemble, pressure p or enthalpy H) are desired to be conserved. In MD, this can be achieved by rescaling the velocities after every timestep, which can be viewed as part of the integration algorithm. However, the rescaling is usually not implemented to be abrupt, but follows an own smooth dynamic instead.

Other integration techniques e.g. use nested loops of different step sizes to respect the different time scales of e.g. intra- and inter-particle movements.

2.4 Other computational conventions

2.4.1 Periodic boundary conditions

In order to avoid edge and wall effects on a molecular system, it is possible to simulate a system with periodic boundary conditions, i.e. the particles of a system are put into a box, which is treated as if it is surrounded by identical translated images of itself. Particles moving out of one site of that box are translocated back to the opposite site, and short ranged pair forces are only considered for the closest duplicate image of every pair of particles. [9]

2.4.2 Pressure and temperature calculation

In case of atoms, all particles are assumed to have no other degrees of freedom than their translational movements in every dimension, so that the equi-partition theorem can be used to calculate the temperature T in a d-dimensional system (usually $d = 3$) of n particles.

$$\frac{1}{n}\sum_{i=0}^{n-1}\frac{m_i}{2}v_i^2 = \frac{d}{2}k_B T$$

$$\Rightarrow \quad T = \frac{1}{d\,k_B\,n}\sum_{i=0}^{n-1} m_i v_i^2 \tag{2.7}$$

Knowing the positions $\vec{r}$ and forces $\vec{F}$ of all n (identical[1]) particles, the virial Φ is defined as:

$$\Phi := \sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1} \langle \vec{r}_i - \vec{r}_j\ ,\ \vec{F}(\vec{r}_i - \vec{r}_j)\rangle\ . \tag{2.8}$$

Using this virial Φ, the current pressure p in a system of dimensionality d with volume V can be calculated by [10]:

$$p = \frac{1}{V}\left(nk_B T + \frac{\Phi}{d}\right) \tag{2.9}$$

[1] "Identical" means all particles are the same type. In the general case, the force $\vec{F}$ may additionally depend on the particle types.

2.5 Parallelization and Communication

To overcome the limited memory and calculation speed available to one processor, many MD programs are designed to be parallelizable. There are different methods to do parallelization: one could run different (non-interacting) scenarios on independent processors, e.g. with different start configurations. While that[2] is always possible, this thesis will only consider the parallelization of one execution scenario. In this case, the parallel processors either have access to a shared memory or they execute on independent memory and need to communicate data and results among themselves. In the former case, special memory access patterns have to be considered to avoid interference of different processors on the same memory. Since shared memory is harder to realise on a large scale in hardware, the communication-based parallelization is more common. The simplest way for such communication is *data replication*, i.e. to copy all data to every processor. Obviously, this requires much memory and communication time. Therefore, one tries to find a way for *data partitioning* which aims at providing every processor only with the minimum data it needs for the calculation. In the case of short-range potentials, such a method is *domain decomposition* (or *spatial decomposition*), which geometrically divides the whole simulation area into sub-domains (*cells*) [11]. That way, every processor only needs to be aware of the particles of its own cell and the particles close to the border of the neighbouring cells. Communication is limited to the particles moving in and out of a cell and possibly forces acting on border particles.

To calculate an estimate for the expected speed-up S by using N processors, one can assume the same calculation speed for every processor, so that the time needed can by measured by just counting the number of instructions I (or similarly: the number of processor clock cycles).

$$S := \frac{t_{normal}}{t_{parallelized}} = \frac{I_{normal}}{I_{parallelized}}$$

Let φ denote the fraction of parallelizable instructions, so that there are $\varphi \cdot I_{normal}$ N-times parallelizable instructions and $(1-\varphi) \cdot I_{normal}$ non-parallelizable instructions. Accounting an overhead time $I_{overhead}$ for initialisation, synchronisation and communication of the parallelization (seen as the number of instructions that could have been executed during those operations), results in:

$$\begin{aligned} S &= \frac{I_{normal}}{(1-\varphi) \cdot I_{normal} + \frac{1}{N} \cdot \varphi \cdot I_{normal} + I_{overhead}} \\ &= \frac{1}{(1-\varphi) + \frac{\varphi}{N} + \frac{I_{overhead}}{I_{normal}}} \\ &= \frac{1}{(1-\varphi) + \frac{\varphi}{N} + \tau} \ , \end{aligned} \tag{2.10}$$

where $\tau := \frac{I_{overhead}}{I_{normal}}$ denotes an overhead ratio. For $\tau = 0$ the last equation is also known as *Amdahl's law* [12] in computer sciences. It implies e.g. that even if the overhead is neglected, a $\varphi = 90\%$-parallelizable program can never run more than $S = \frac{1}{1-\varphi} = 10$ times faster by parallelization, no matter how many processors are involved.

[2] pejoratively called "embarrassing parallelization"

2.6 LAMMPS

As MD is an established method, there are existing, highly developed MD simulation programs. The program considered here is LAMMPS [10], the Large-scale Atomic/Molecular Massive Parallel Simulator, which is written in the programming language C++ and freely available under the GNU Public Licence. In 2009 alone, more than 250 scientific papers, published that year, have referred to Lammps in their quotations, making it de facto one of the MD standard simulators.

Lammps does not require to compile own code for different MD scenarios and has instead invented an own scripting language to describe settings (e.g. concerning neighbour lists), initial configurations (like atom positions, geometrical sizes and periodicity), algorithms to be computed (including forces and integration algorithm) and which data to output (like positions, velocities, pressure or temperature) and when to do so. The wide range of possible setups given by Lammps' input scripts goes far beyond the scope of the discussion of this work, but the above mentioned pair forces, neighbour lists strategies, integration algorithms and the possibility for periodic boundary conditions are of course part of Lammps' functionality. Lammps is able to use a single processor as well as multi-processors and even multiple PCs, which are linked through an ethernet network. It follows the spatial decomposition strategy and uses a Message Passing Interface (MPI) library for communication. Each Lammps process calls the particles in its own cell *local*, those in the neighbour cells within the interaction radius *ghost*. The number of particles is referred to as n_{local} and n_{ghost}, respectively. Furthermore, $n_{all} := n_{local} + n_{ghost}$.

In order to achieve its flexibility, Lammps faces the problem to make many decisions at run time (based on the user's input script). Since these decisions take time, they should be avoided in often executed code parts. To do this in an elegant manner, Lammps makes use of object oriented C++ features like inheritance: a parent class (like `Pair` for pair forces) defines an interface of member variables and methods (like `compute()`) common to all pair forces. Lammps than initially constructs other objects with pointers to instances of derived classes (like `PairLJCut` for the Lennard-Jones force with cut-off radius). This idea is also known as the *(abstract) factory paradigm* in software engineering. That way, the perpetual evaluation of many conditions in code can be avoided by initially storing the addresses of exactly the (e.g. force computation) functions which had been chosen by the user. During the later program execution, the code can unconditionally jump to that address.

Chapter 3

GPU Programming and the Nvidia Cuda Technology

Graphics cards are designed to render high-quality 3D images in real time. The very competetive market in this field has turned modern graphics cards into powerful computational units at comparatively low cost. It is therefore interesting to find a way to make use of this computational power also for other purposes than graphics rendering. Traditionally, this *general-purpose computing on graphics processing units (GPGPU)* was rather difficult, since programming tasks had to be implemented into the rendering work-flow of the graphics card. Becoming popular in 2003, freely programmable parts (vertex and fragment shaders) were added to it in order to allow more realistic effects and could also be used e.g. to accelerate movie editing. In November 2006, the NVIDIA® company, a well-known producer of chip-sets and GPUs, introduced it's *Compute Unified Device Architecture (CUDA™)* aimed at general-purpose parallel computing. Because of the historical background in graphics rendering and the differences from serial to parallel program execution, GPU programming has some major differences to usual CPU programming. Those should be explained in this chapter based on Nvidia's CUDA, starting with an introduction to GPU architectures in comparison to CPUs. Based on this overview, the programming model and the memory access patterns introduced by CUDA are described, finishing with a list of technological limits which need special considerations.

The information in this chapter can also be found in the literature, like the Nvidia Cuda Programming Guide [13]. Therefore, this chapter aims at summarising the main aspects of the CUDA technology, not focusing on programming details. Vocabulary and concepts within the field of CUDA programming should be described, in order to have a basis for the discussion in the later chapters.

3.1 GPU Architecture

NVIDIA's graphics cards contain a number of multiprocessors (*MP*s), were each such *MP* consists out of eight processors. The following table shows a selection of NVIDIA's devices.

Release Date[1]	**Product Name**	**Number of MPs**
December 2008	GeForce 9200M GS	1
October 2007	Quadro FX 3700	2
April 2007	GeForce 8600 GTS	4
September 2009	Quadro FX 2700M	6
February 2008	GeForce 9600 GT	8
April 2007	Quadro FX 4600	12
July 2008	GeForce 9800 GT	14
May 2007	GeForce 8800 Ultra	16
June 2008	Quadro FX 4800	24
June 2008	GeForce GTX 280	30
March 2009	Quadro FX 4700 X2	2×14
November 2006	Tesla D870	2×16
January 2009	GeForce GTX 295	2×30
November 2008	Quadro Plex 2100 D4	4×14
October 2008	Tesla S870	4×16
June 2008	Tesla S1070	4×30

Each single MP is clocked with about 1.5 GHz, which - combined with the large number of MPs - underlines the above claimed computational potential, compared to dual or quad core CPUs nowadays being clocked between 2 GHz and 4 GHz. It is, however, misleading to directly compare GPUs to CPUs just by the product of the processor clock frequency and the number of processors, because those two architectures work fundamentally different and a so-calculated number does not correspond to the actual speed at which an arbitrary program would perform on a CPU and GPU, respectively.

A CPU is able to run completely different tasks on each of its cores in parallel, which requires advanced instruction flow control and makes it hard for a CPU producer to design a new CPU with more cores. CPUs also spent a lot of effort in sophisticated caching strategies allowing them to temporally buffer the values of often used memory addresses to which each of the executed processes has random read and write access to. This is done to reduce the latency of a memory access.

GPUs on the other hand are designed to run the same calculation on many data elements in parallel. Historically, an example for such a calculation would be to apply a graphical effect to many objects in a virtual scene or pixels on the screen. Since the same instructions are carried out by all cores, no advanced flow control is required and if the number of arithmetic instructions is much higher than the number of memory accesses, then the lack of sophisticated caching strategies has only a weak influence on the total execution time.

Besides these differences, both the CPU and the GPU cores have sub-units to read and write memory addresses, registers to internally store values and an instruction units supporting the same basic instruction set (like *add value B to register A*, *copy the contents of register A to register B* or *compare the value of register B to 0*). This made it possible for Nvidia to develop a programming interface compatible with an existing programming language, which later gets compiled to low-level GPU instructions, adapting already existing

[1] These dates vary in different sources.

compiler work-flows and programming language features. Choosing the already established programming language C, they made it easier for programmers to start writing their own GPU programs, also facilitating the commercial success of CUDA-enabled graphics cards.

3.2 Cuda Programming Model

3.2.1 Kernels

In order to use the parallel computing power mentioned in the last section, CUDA extends the C language by so-called *kernels*. A kernel is a special function, whose instructions can be executed N times in parallel on the graphics card. One such execution is called a *CUDA thread* and current graphics cards can run up to 1024 threads per MP concurrently. To differentiate between code executed on the CPU and the GPU, the main CPU program and the computer's main memory (RAM) is called the *host*, while the GPU part is generally called *device*.

In order not to execute the same calculation on the same data more than once, each of the N threads gets one individual index at the beginning, which can be used by the kernel e.g. to tell which item of an array it should run its instructions on. The following code gives an example of a complete and valid kernel definition.

```
0: __global__ void myKernel(float* x, float* v, float dt) {
1:     int i = threadId.x;
2:     x[i] = x[i] + v[i] * dt;
3: }
```

The above example follows the *Single Instruction, Multiple Data (SIMD)* pattern, which is typical of a GPU and is opposed to the traditional CPU pattern of *Single Instruction, Single Data (SISD)*. Furthermore, the C code of the kernel also allows conditional ("if-then-else") parts. Nvidia calls the resulting concept *Single Instruction, Multiple Thread (SIMT)* in the style of the just mentioned classifications. To give an example, it would be possible to run different code whenever the index i is odd or even. However, due to the above stated hardware architecture, the instruction unit of the GPU would then firstly execute all threads whose index is odd and afterwards the threads whose index is even. This code serialisation (also called *branching*) temporally freezes half of the processors and causes a serious performance loss and should therefore be avoided where possible. If, however, all threads evaluate an "if"-condition to the same result, the execution will not branch, therefore not causing performance losses. To be more precise, the instruction unit of one MP always fetches instructions for four GPU clock cycles. Since one MP consists out of eight processors, the decision if branching occurs, depends on whether all $(8 \cdot 4 =)$ 32 threads come to the same result. This is one of the main reasons why Nvidia introduced a new term for such a group of 32 threads: it is called a *warp*.

3.2.2 Thread Hierarchy

Since computer graphics usually deals with two-dimensional (2D) or three-dimensional (3D) data, CUDA conveniently enables the threads to be virtually grouped into a 2D or 3D grid, resulting in a 2D or 3D index, which can e.g. be used to address 2D or 3D arrays. More precisely, CUDA allows two different layers of such grouping: firstly, up to 512 threads may be grouped into a *thread block* and secondly, the overall *grid* of one kernel may consist of up to roughly four billion of such thread blocks. While one MP can execute many blocks in parallel, one block cannot be executed on multiple MPs. A CPU program can call a kernel similar to a usual function call in C, just additionally specifying the just mentioned grid dimensions (together called *execution configuration*) with a special syntax, outlined in the following code:

```
0: int main() {
1:     // ...
2:     dim3 blocks_on_grid(1000, 1000);
3:     dim3 threads_per_block(8, 8, 8);
4:     myKernel<<<blocks_on_grid, threads_per_block>>>(x, v, 0.1);
5: }
```

Inside the kernel function, CUDA automatically sets built-in variables to access the grid and block dimensions the host program had specified (`gridDim`, `blockDim`) and the index - in the range of those dimensions - of the currently executed thread (`blockId`, `threadId`).

Starting α blocks with β threads each ($\alpha, \beta \in \mathbb{N}^+$) or β blocks with α threads each results in the same number of total threads ($\alpha \cdot \beta$). Between these two scenarios, there are, however, two differences. Firstly, threads inside of one block can synchronise their execution, i.e. a kernel can call the special function `__syncthreads()` marking a point inside the code all threads in the block have to reach before any of them is allowed to proceed further. Each thread must call the same number of `__syncthreads()` calls during its execution. Secondly, the threads of each block can share a small amount of memory (16 KiB). The memory handling used on a CUDA GPU is described in the next section.

3.3 GPU Memory

3.3.1 Memory Spaces

Despite the shared memory of a thread block already mentioned above, there are additional memory spaces on the graphic card, which are deeply influenced by concepts of traditional GPU programming. Device code only operates on device memory.[2] The understanding of the different GPU memory spaces is crucial, since their usage has strong influence on the overall performance. The following table shows an overview of the different memory spaces, which will be discussed below.

memory	**global**	**constant**	**texture**	**shared**	**local**
access	read & write	read only	read only	read & write	read & write
size	Ω	64 KiB	Ω	16 KiB	Ω
cached	no	yes	yes	no	no
latency	Δ	ε	$\approx \Delta/10$	ε	Δ
lifetime	persistent	persistent	persistent	block	thread

The symbols Ω, ε and Δ are explained below.

Global Memory

The *global memory* is the main memory of the device to which a kernel can read and write data. New memory can not be reserved during a kernel execution, but must be allocated before a kernel gets invoked. Its size is the total graphics card memory (an often mentioned product feature) and covers usually several 100 MiB, above denoted as Ω. Access to it is not cached, and has an latency of 400 to 600 clock cycles (Δ) - to compare, an addition of two numbers takes four clock cycles. In order to have reasonable performance, it is therefore necessary to perform many arithmetic operations per memory access. The global memory is persistent across kernel calls, i.e. subsequently called kernels can perform different operations on the same data.

Constant Memory

The *constant memory* is a small (64 KiB) read-only memory, which is also persistent across kernel calls. The host program can load any data into the constant memory, the attribute *read only* applies only inside the kernel (executed on the *device*). Access to it is cached, so a read is as fast as an access of a processor register (ε) after the constant memory address has been fetched for the first time from the device memory.

Texture Memory

The *texture memory* is a fast read-only memory. Access to a texture element costs one read from device memory when reading for the first time and a much smaller time to read from the texture cache afterwards. An exact latency for an texture access is not documented and depends on the memory access pattern of the kernel. CUDA allows the host to declare parts of the global memory as texture, so that kernels can make use of the texture distribution mechanisms. It is, e.g., also possible to load a texture with a discrete pixel width and access an linearly interpolated value for any floating point index in between.

[2] Newer versions of Cuda may also read and write directly on a non-pageable CPU memory (called *zero copy*). However, this memory is limited and using it may slow down other processes on the operating systems. The access latency over PCIe is much greater than internally on the graphics card.

Shared Memory

The *shared memory* is a small (16 KiB) memory, which does not reside in the global device memory and is not persistent across kernel calls. The host cannot load any data into it. However, when the host calls a kernel, it may specify a size up to 16 KB, which will be available to each thread block as read and write memory. When a thread block starts, this memory is allocated; while the (up to 512) threads of the block execute, they can use this memory to share data among themselves and after the last thread has finished, the memory is freed. Access to a shared memory element is as fast as an access of a processor register and thus does not need to be cached.

Local Memory

The *local memory* has similar features as the global memory - only the lifetime and scope of local variables are bounded to one single thread each, making them quite unattractive. The reason why they have been introduced can be summed up as follows: Since an MP can run up to 1024 threads concurrently and has *only* 16384 registers[3], each thread can use 16 registers on full load. If more different variables are needed at the same time than there are registers available, the remaining variables are written out to the local memory. Unfortunately, the choice of how many local memory should be used to save registers must be done at compile time, while the number of started threads may be chosen at runtime. This can cause a kernel to refuse to start, when too many threads are requested. Since the local memory allows only slow access, it should be avoided where possible. This can e.g. be done by writing simpler code, that does not require many different variables at the same time.

3.3.2 Memory Copying

CUDA provides host functions (included in the *CUDA API*) to copy data to and from the device' *global memory*, as well as functions to load data to the *constant memory*. Only continuous memory blocks can be transferred - therefore, multi-dimensional arrays have to be flattened into a one-dimensional arrays before copying. The bandwidth between host and device memory is limited by the PCI-express interface and - moreover - the motherboard chipset. Typically[4], it lays between $2\frac{GiB}{s}$ and $6\frac{GiB}{s}$. The device kernels can load textures and constants and work either on global and shared memory explicitly, or on local memory implicitly. However, since shared and local memory are not kept after a kernel has finished, eventually every result of a device computation has to be copied back to the *global memory*.

A kernel can e.g. copy data from the global to shared memory with a simple C-assignment in the device code.

```
my_shared_array[i] = my_global_array[j];
```

The bandwidth for such an operation inside the graphics card is quite high (about $100\frac{GiB}{s}$), but the above mentioned access-latencies have to be considered.

[3] The exact numbers have changed over time, the above mentioned are those of the latest specification (compute capability 1.3). See the Nvidia Programming Guide for older specification details.

[4] Appendix 7.4 lists a bandwidth benchmark for different workstations.

3.4 Limitations

Beside already mentioned limitations like bandwidth and latency, graphics card programming has a variety of other restrictions which originate from either the parallel nature of GPU programming or principles of GPU design. While there are a number of rather technical restrictions (e.g. concerning the number of instructions per kernel or the ranges for a valid *execution configuration*), this chapter will point to Nvidia's CUDA Programming Guide for these specific questions and will instead focus on the description of more principle limitations.

3.4.1 Write Conflicts

When multiple threads write different values to one single memory address while they are executed in parallel, only one of the write-requests is guaranteed to succeed. If, e.g., N threads want to increment an integer z in global memory by one, the result is likely not $z + N$, but instead a smaller result like $z + 1$, because each thread first reads the same initial value of z, adds one and writes the result back. The result may as well be $z + 2$ or greater, depending on whether a thread was scheduled to start after another had finished. Because the exact behaviour depends e.g. on the number of MPs on the GPU, it can not be predicted while writing the program.

To overcome this limitation, Nvidia introduced a set of *atomic functions* allowing such read-modify-write operations to be performed without interference from other threads on integer (but not floating point) values. These enhanced abilities of a GPU-series are marked with a version number Nvidia called *Compute Capability* - being currently 1.0, 1.1, 1.2 or 1.3. Starting from Compute Capability 1.1 atomic functions for 4-byte words are supported, Compute Capability 1.2 and later grants for atomic operation support for 8-byte words.

3.4.2 Coalesced Access

The threads of a kernel read from or write to *global memory* in groups of either 32, 64 or 128 consecutive bytes. The hardware is able to transmit 64 bytes at once. When accessing 4-byte-words (like `float` or `int`), this means that a maximum of ($\frac{64B}{4B}$ =) 16 threads can simultaneously access memory. Such a group of 16 threads therefore got its own name *half-warp*.

If all threads of half-warp simultaneously access data from a single memory segment, the data can be transferred in one single transaction - the access is then said to be *coalesced*. To see how important coalesced access is, it will be compared to a scenario where all threads read different (4-byte) memory elements lying in the N different memory segments: each thread then fetches 32 bytes (the minimum transaction size), but only uses 4 bytes of the result, having wasted $\frac{7}{8}$ of the bandwidth. The accesses are serialised and each of them has its own latency. In practise, this causes non-coalesced access to be an order of magnitude slower than coalesced access (for 4-byte elements). For non-coalesced access, the "efficiency factor" of bandwidth usage increases from $\frac{4}{32} = \frac{1}{8}$ to $\frac{8}{32} = \frac{1}{4}$ when accessing 8-byte elements or $\frac{16}{32} = \frac{1}{2}$ when accessing 16-byte elements.

For completeness, it should be noted, that for devices with Compute Capability 1.0 and 1.1, the n_{th} thread of a half-warp had to access the n_{th} memory element in a memory segment. Since Compute Capability 1.2 it can be any permutation as long as the accessed elements belong to the same memory segment.

3.4.3 Bank Conflicts

When different threads of a kernel read from or write to *shared memory*, the memory access is only as fast as a register access as long as there are no *bank conflicts*. The memory bandwidth to shared memory is divided into 16 equally-sized memory modules, called *banks*. Memory elements are always accessed through the bank number: memory address `mod` 16. Bank conflicts occur as soon as two threads of a *block* simultaneously try to use the same *bank*. There is only one exception to this rule: if *all* threads simultaneously access the same element in shared memory, a special broadcast mechanism is triggered, which avoids bank conflicts.

3.4.4 Other

A selection of other limitations is listed below:

- *Double precision* is not supported for devices before Compute Capability 1.3. Calculations in double precision are about four times slower than their counterparts in single precision, because each MP has only one unit able to perform double precision operations. If one wants to transfer an existing array in double precision to the device, every single array element needs to be casted to single precision before the calculation starts[5]. There are fast built-in implementations for elementary mathematical functions (like sinf, expf, sqrtf), only in single precision.
- Kernels do not support *external function calling*. Whenever there is a function call inside the kernel code, the function is inlined into the kernel at compile time, which is why recursion is impossible.
- Operations like *division and modulo* are ten times slower than an addition.
- The device code supports C and only a small subset of *C++ features*: polymorphism, default parameters, operator overloading, namespaces and function templates. Features like exception handling, STL functions, classes and inheritance are not supported.

[5]The cast has do be done explictly, since `float` values use 1 Bit for the sign, 8 for the exponent and 23 for the significant, while `double` uses 1, 11 and 52, respectively.

Chapter 4

Implementation Strategies

The aim of this chapter is to reason the decisions token in the mentioned implementation. Where possible, it tries to do so by confronting the decisions with possible alternatives and their implications. Firstly, the discussion will deal with the question of why porting an existing program and how to implement Cuda there. Since it turns out that force calculations play a major role in MD simulations, their implementation will be discussed in detail. Still, the overall speed-up is better if other parts of the simulation are also done on the graphics card directly. Therefore, the last section of this chapter will discus how to implement these parts on the GPU.

4.1 Integration Strategy

4.1.1 Why port an existing program?

When trying to write an MD simulator for the GPU, there are two possibilities: either write a completely new program or try to port an existing program to the GPU. The advantages of inventing a new one are that there is no need to cope with old hangovers from CPU design decisions and one could respect GPU specifics (e.g. concerning parallelization) right from the beginning, which could result in an faster program. On the other hand, one would need to re-invent existing concepts (like input file parsing) and it is likely to run into design flaws (and even problems one is not thinking about) an existing program has already overcome. Writing a new program potentially also means much more work, but if one is doing a project based on an existing program, it is also possible to use its community (e.g. for testing or extensions). Therefore, the decision was to port an existing program with the first goal to *keep* its *flexibility*, i.e. being able to adapt all Lammps features, automatically accepting that such a general-purpose program is likely to run slower than a special-purpose code, which does not need to respect many decisions of the user at runtime.

4.1.2 Integration of Cuda code into Lammps

As mentioned before, Lammps makes use of rather advanced C++ features, while the device code supports mostly only C with only a small subset of C++ features. The special syntax to invoke a Cuda kernel (`<<<...>>>`, see section 3.2.2) is not valid C-code and is only understood by Nvidia's C compiler. Therefore, an interface between these two parts (Lammps and Cuda) on the common ground C is needed. The idea was to compile the Cuda kernel and a function which calls it (using Cuda's syntax) with Nvidia's compiler into an object file, which gets linked into Lammps' binary during compilation, so that Lammps can call the kernel-invoking functions in its C++ code - to do so, the prototype of each such external functions has to be declared in a header file both the Lammps and Cuda part include.

Data transfer

Some functions, which are provided by the Cuda API, can be encapsulated into C++ classes in order to keep the advantages of object oriented programming - e.g. for memory management a data class had been implemented. When a data object is constructed, it allocates the necessary memory on the GPU and frees it again, whenever the class is destructed (even without an explicit call). It provides methods to download and upload data to an from the device - transparently converting a multi-dimensional array into a memory-block and casting from double to single precision, if required. The compiler can produce extra byte-code for every scenario occurring in the source code out of one single written C++ template class.

Precision

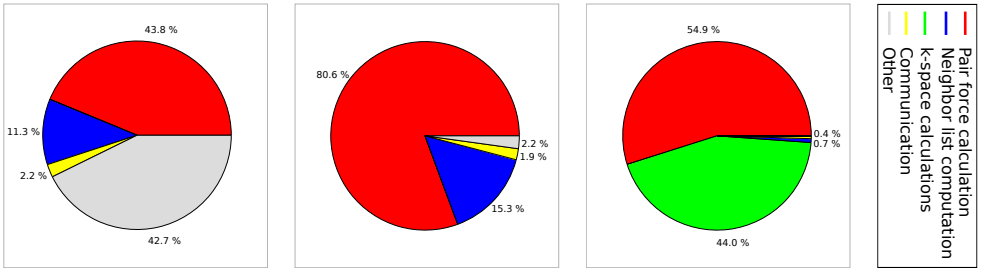
MD simulators generally prefer `double` precision, because it causes a smaller numerical error (for both order of convergence and consistency). However, GPUs are generally designed to operate on single precision numbers (`float`) and are known to work much slower on `double`s (see section 3.4.4). In order to leave a choice to user whether he or she prefers accuracy or speed, and to be able to do benchmarks on this topic, a compile-time switch was introduced. It triggers pre-processor substations and macros, which allow to use

- an own Cuda precision type (which is simply replaced by either `float` or `double` by the pre-processor),
- different precisions for different tasks (*mixed precision*),
- an automated switch between the `float`-optimised mathematical functions (like `sqrtf` or `expf`) and their `double` counter-parts (`sqrt` and `exp`),
- constants in different precision, because the compiler distinguishes between `double` (`1.0`) and `float` (`1.0f`) constants and bases the calculation accuracy on it.

4.1.3 Which parts to port?

Obviously, it does not make sense to port every part of Lammps to the GPU - otherwise it would have been possible to write a completely new program. While Lammps' MPI-based parallelization techniques can be used to communicate between different CPU processes (where each of them may incorporate its own GPU), it is - unfortunately - not possible to use MPI for parallelization on the GPU, because MPI relies (besides a different API) on independent CPU processes (with SISD architecture), while the GPU needs code based on a SIMD (or SIMT) architecture. Therefore, appropriate parts of Lammps have to be chosen thoughtfully. From all parts which are parallelizable on the GPU (that excludes, e.g., input file parsing) the most computational-intensive parts are chosen, in order to achieve the second goal of *maximising the speed-up*. As Lammps provides the user with time statistics after every run[1], it is easy to find out the most time-consuming parts. The pie charts in figure 4.1 illustrate the proportions in three randomly chosen examples[2].

Figure 4.1: Time proportion for sample runs on 2 CPUs



(a) LJ Poisson Flow (b) LJ Melt (c) Silicate Buckingham (d) Legend

As obvious from fig. 4.1, the force calculations take most of the time in many scenarios, as it could have been also concluded from theoretical considerations. Therefore, their porting is important for a GPU-based MD-simulation and will be the topic of the next section.

4.2 Force calculation on the GPU

Since force calculations are crucial for the acceleration of MD programs, they are discussed in this section in detail. However, there are different types of forces needing a different treatment. Therefore, this section will firstly discuss non-pair forces, afterwards short-range and finally long-range forces.

4.2.1 Non-pair forces

Just like many MD simulators, Lammps offers the possibility to enforce global forces into the simulation, like adding a constant force to a group of particles (e.g. to simulate a gravitational field), or prevent particles from moving at all by resetting any force on them to zero every timestep. This basic idea is rather trivial to implement, because one just has to write a kernel setting the desired force and than start n_{local} threads of this kernel. One single instruction will modify multiple forces. The only problem left is choosing a good *execution*

[1] Otherwise, profiling with g++ -g and gdb -p would have been possible.
[2] The "Benchmark" chapter will give more details about them.

configuration, i.e. the number of blocks on the grid and threads per block, those product is the total number of threads. Since n_{local} could be prime, the number of started threads has to be rounded up to a number n_{round}, e.g. to the next multiple of 64. Each thread has to calculate its own particle-id out of the multi-dimensional index (`blockId` and `threadId` - see section 3.2.2) - this is trivial and listed in the appendix 7.3.1. It is recommended to start a multiple of 32 threads per block in order to always have full warps. Consequently, starting more thread than there are memory elements implies that each thread needs to check whether its index is still in the valid memory range. This single condition check generally does not course performance losses by branching, since only the last $n_{round} - n_{local}$ threads will evaluate the condition to a different result than the other threads in their warp and the latter number is sufficiently small for a large number of particles.

4.2.2 Short-range pair forces

Concerning pair-forces the situation is less trivial than for non-pair-forces. When using cut-off-radii, the goal of any implementation strategy must be to avoid a full $\mathcal{O}(n^2)$ check. The method implemented in Lammps to solve this problem is the use of neighbour lists (NLs), which are an accepted method, often recommended in the literature and used in highly optimised programs like Lammps. A direct port of the already implemented code appears to be easier, since the existing infrastructure can be kept. However, methods proved to be efficient on the CPU may work out different on the GPU, where this rather new technology also lacks standard literature recommending any strategy. NLs generally consume a lot of memory and graphics cards usually have less of it than CPU architectures. Also, NLs rely on random accesses in memory, being ten times slower than coalesced memory accesses on the GPU. Therefore, this section will develop the idea for an alternative method. Firstly, general implementation ideas are given, afterwards implementation details for both methods are presented and finally a discussion will compare both strategies.

General implementation ideas

Generally, it is possible to load numbers like n_{local} or force parameters (like ε and σ for the LJ force) into the device' *constant memory* (as long as they fit into 64 KiB), in order to have fast access to this small set of often needed constants.

Lammps stores the particle positions $\vec{r}_i = (x_i, y_i, z_i)$ ordered by particle-id, which results in a memory structure like the one drafted below:

x_0	y_0	z_0	x_1	y_1	z_1	x_2	y_2	z_2	x_3	y_3	z_3	x_4	y_4	z_4	...

Since the forces $\vec{F}(\vec{r}_i - \vec{r}_j)$ should be calculated for many particles in parallel and a force calculation will need to load the atom positions from *global memory*, the consecutive reading of the position's coordinates will result in *non-coalesced access*. However, if the positions are re-arranged into a memory scheme ordered by dimension, each such access can be coalesced, since in almost all cases a contiguous memory region can be read. The following scheme illustrates the memory structure with the concurrently needed memory elements underlined and coloured in blue.

x_0	x_1	x_2	x_3	x_4	...	y_0	y_1	y_2	y_3	y_4	...	z_0	z_1	z_2	z_3	z_4	...

The re-ordering can be done on the CPU or on the GPU and can be seen as the more efficient, the more often the positions are accessed afterwards.

Implementation of neighbour lists

Since the primary aim of this section is to determine the faster force computation strategy, the question of whether and how to build NLs on the GPU will be ignored for the moment, because the speed comparison of pure force calculation can be done with the NLs built on the CPU, for the moment simply uploaded the GPU.

The algorithm is straight-forward and outlined in the following 14 lines of "free C code", where data transfer from (↞) and to (↠) the *global memory* are marked by arrows.

```
0:  __global__ void LJ_Kernel() {
1:     int i = my_particle_id_from(blockId, threadId); // appendix 7.3.1
2:     float3 my_r ↞ r⃗[i]; // coalesced access, if reordered as explained above
3:     int my_type ↞ type[i];
4:     float3 my_F = 0⃗;
5:     int my_neigh_count ↞ neigh_count[i];
6:     foreach(neighbour j of i) {
7:       int neigh_type = type[j];
8:       float3 neigh_r ↞ r⃗[j];          // non-coalesced access
9:       float3 Δr² = (my_r - neigh_r)²;
10:      if(Δr² ≤ r_c²) my_F += F⃗_LJ(Δr); // both r_c² and F⃗_LJ depend on the
11:    }                                  // types my_type and neigh_type
12:    my_F ↠ F⃗[i]
13: }
```

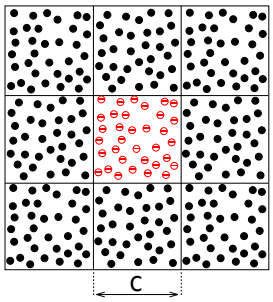
There are three things to note about this algorithm: firstly, the *global memory* is often accessed non-coalesced, as the neighbour positions and types are "randomly" distributed in memory[3]. Secondly, *branching* will occur very often, not only because of the `if`-condition on line 10, but also due to the variable number of neighbours per particle (thus, different number of iterations for the loop starting on line 6). While leaving some processors idle can not be avoided principally in this force calculation, the threads of a *warp* can be hindered to diverge further, by demanding a synchronisation (`__syncthreads()`) before the end of the neighbour loop on line 11. Since every thread must have the same number of `__syncthreads()` calls, either the loop has to be executed as many times as the maximum number of neighbours over all particles, or additional ($max\_neigh\_count - my\_neigh\_count$) `__syncthreads()` have to be called after the end of the loop. Thirdly, any force between a pair of particles is calculated twice (not using Newton's Third Law $\vec{F}_{ab} = -\vec{F}_{ba}$), because once $\vec{F}_{ab}$ has been computed and added to the force of particle a, it can not be subtracted from the force of particle b, as this could cause *write conflicts*, since another thread might be simultaneously working on the same particle.

Apparently, the apprehension of this section's introduction could be confirmed: the workflow implicitly implied by Lammps' NL data structures seems to be sub-optimal for GPU parallelization.

[3] in the sense, that they can not be read as a continuous memory region

Implementation of grid cell lists

Figure 4.2: cell lists, drafted in 2D



The last paragraph showed a working strategy for parallel force calculations on the GPU, albeit pointing to the discussion whether it is worth to implement an alternative method, which does not need neighbour lists and is based on memory structures aiding coalesced accesses. The idea is to continue the spacial decomposition into a regular grid of small sub-cells, with a maximum number of atoms per cell. This is also based on the experience, that the density in many studied systems is homogeneous, i.e. there is an upper bound for the number of atoms in any cell (drafted red / striped in figure 4.2). A disadvantage of this method is the need to re-order the existing data structures in Lammps for the GPU calculation, which also requires to convert the data back into the original Lammps format for every usual Lammps computation not done on the GPU. This additional effort could be justified if the cell list method was substantially faster than the calculation based on neighbour lists.

For this *cell list approach*, the data needs to be reordered into cells, which makes it necessary to chose a geometrical cell size c (see fig. 4.2) and the maximum number of atoms per cell n_{cell_nmax}. Afterwards, a multiple of n_{cell_nmax} bytes in GPU memory has to be allocated (e.g. for positions, velocities and forces) and after the atom positions have been copied into that new memory, the forces can be calculated. The cell size depends on how the elements of the above idea are chosen to correspond with GPU programming structures. The implementation idea was to associate every *cell* with a Cuda thread *block* and have every *thread* of it calculate the forces for one *particle* in the cell, i.e. - in the first place - each of the n_{cell_nmax} threads will read different positions $\vec{r}_i$, but will write the resulting forces only to its 'own' memory.

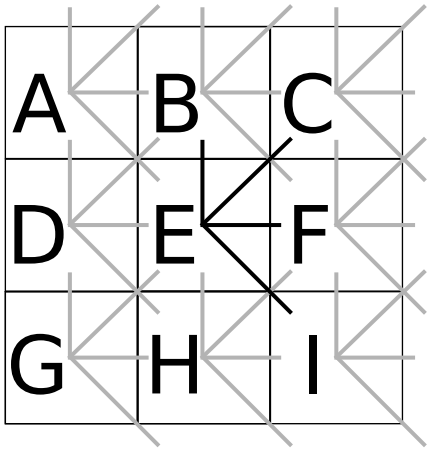
As a Cuda block can only contain 512 threads, $n_{cell\_nmax} \leq 512$ is required and the geometrical size c of the cell must be chosen small enough. Additionally a certain margin is required to ensure that - even with slightly varying particle density during the simulation - a cell will never contain more than 512 particles. Moreover, the force calculation will look at all particle pairs in the cell and if the cell size c was much greater than r_c, the distance between many pairs would extend r_c, which would not only imply needless distance checks, but could also support branching - i.e. to leave processors waiting for other force calculation threads to finish. While these arguments support the idea not to grow a cell arbitrarily large, there are also reasons not to shrink it arbitrarily small. Guided by the recommendation to start a multiple of 32 threads, it is obvious that a cell size being too small would result in a waste of memory, because only a few particles would be in the smallest possible cell large enough to store 32 particles. Further on, it appears to be the easiest way to calculate the forces first in the own cell and afterwards sequentially with each neighbour cell (see fig. 4.2), whose number is $3^3 - 1 = 26$ (in 3D) only if c is chosen greater than r_c, because in this case only the closest neighbour cells have to be considered. Additionally, the time evolution of the system has to be considered: the greater c is chosen, the less frequent the reordering into cells need to be done - which supports the idea to chose $c \approx r_{neigh}$ (see section 2.2.2). Thus, this line of argument recommends to choose $c \approx 2\, r_c$ and n_{cell_nmax} as a multiple of 32 between 32 and 512 depending on the particle density.

To avoid *branching*, each cell can be filled up (to next multiple of 32) with "zero particles", whose force constants (e.g. σ, ϵ) are set to zero. That way, they do not influence the result, but take care that the same instructions are executed on all data elements.

The reordering into cells itself can be done by two different methods in parallel, which

could be described by a *push* and and a *pull* strategy. Either, as many threads are started as there are cells and each threads loops over all n particles testing for each particle whether it belongs to the own cell and if so, add it to the cell's memory (thus, "pulling" it) - or, n threads are started and the corresponding cell is determined by each particle's position and then "pushing" it into that cell. The latter approach (which was chosen in the presented implementation) needs to make sure, that only one thread wants to add a particle to a cell at once. This can be done by using *atomic functions* (see section 3.4.1): whenever a particle should be added, 1 is added to the particle count of that cell $n_{cell\_count\_i}$. The function returns the old value of $n_{cell\_count\_i}$, which is the memory position to write the particle.

Figure 4.3: cell update pattern



For the force calculation itself, at least two more optimisations can be done in this cell list approach, which are not possible for NLs. The first is to use Newton's Third Law $\vec{F}_{ab} = -\vec{F}_{ba}$ to save half of the force calculation time. In 2D, this means to only consider 4 out of 8 neighbour cells - figure 4.3 drafts an example of such an update pattern, with the chosen neighbours of cell E marked by a solid black line. If every cell follows this pattern, the interactions between all neighbouring cells will eventually be considered exactly once, as one may verify for cell E. In 3D, only 13 out of 26 cells are considered. Note, however, that not every selection of 13 neighbour cells fulfils the required periodicity. An effective algorithm for automated generation in the 3D case is given in the appendix 7.3.2 [4].

While this sounds trivial, it must be recalled that thread blocks are allowed to be executed in any order, in sequence or in parallel. Therefore, *write conflicts* may occur: in figure 4.3, e.g. cell A and D might try to update the forces in cell B at the same time. In order to avoid the resulting error in the calculation, the idea was to simply execute different groups of cells, which cannot interfere, after another. In the above 2D example, there are four such groups, with cell A, C, G and I being part of one[5]. This does not significantly affect performance since N groups are executed each in approximately $\frac{1}{N}$ of the original time.

The second optimisation is the use of *shared memory*. Initially, the positions are stored in *global memory* and need to be loaded from there. However, since multiple particles (threads) inside a cell (block) will need the same positions, it is an obvious idea to load all positions from the own or a neighbour cell only once from global into shared memory and re-use the data afterwards from this local and much faster memory. In order to hinder one thread to load positions from the next cell, while other threads are still needing the current data, a `__syncthreads()` can be triggered (see section 3.2.2).

As a position $\vec{r}$ contains three `float` elements - each 4 Bytes in size (or 8 Bytes for `double`) - only 6 KiB (for `float`) or 12 KiB (for `double`) of shared memory are needed, even if the maximum of 512 threads were started - this is less than the hardware limit of 16 KiB, thus *not* implying additional restrictions for the cell size.

With this state of development, the cell lists seem to be ready for speed comparison with the *neighbour list approach*.

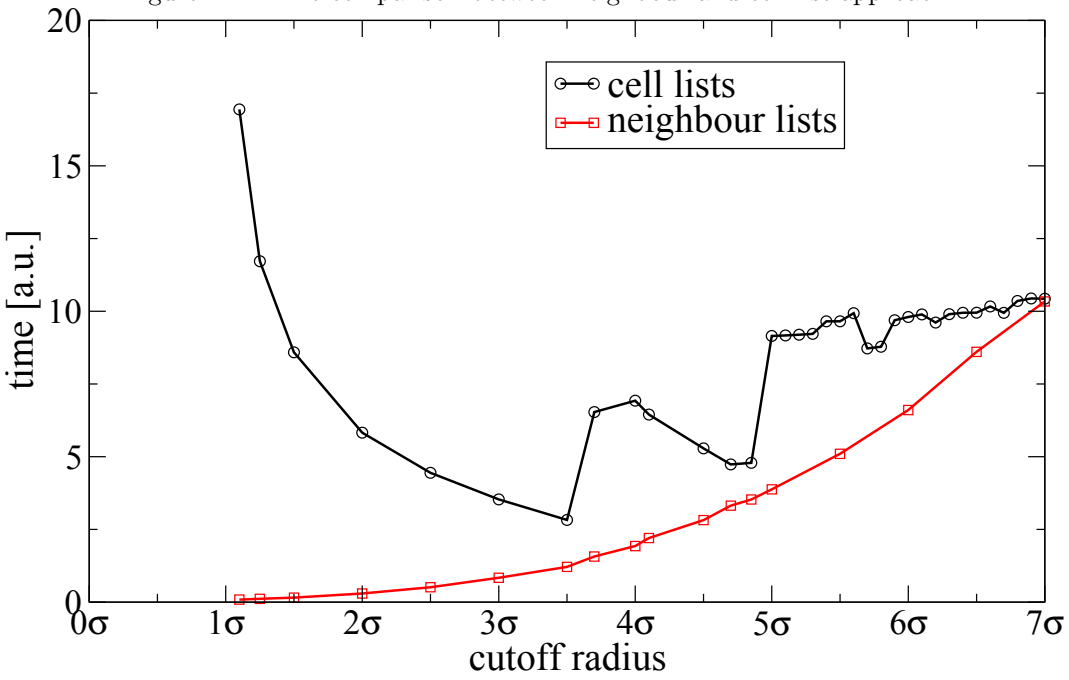
[4] One of the reasons for writing an algorithm for that simple geometrical problem is that later on, multiple neighbour shells are considered, where a selection of a valid updated pattern might extend simple intuition.

[5] Mathematically spoken, for c neighbour shells, the set of cells with indices $\vec{k}$, $\vec{l} \in \mathbb{N}^d$ is partitioned according to the equivalence relation $\sim$. $\vec{k} \sim \vec{l} :\Leftrightarrow \forall i \in [0, d) : (k_i - l_i) \bmod c = 0$

Speed comparison

From the current state of development, the time for the pure LJ force calculation kernel is measured for different cut-off radii on one GPU for the cell and neighbour lists approach. In the example, $n \approx 2 \cdot 10^4$ particles are uniformly distributed in a box of approximately $(25 \times 25 \times 25)\sigma$ and the time corresponds to one initial time step, where the same results are computed by both different methods.

Figure 4.4: Time comparison between neighbour and cell list approach



Since the runtime of the above neighbour list algorithm is in $\mathcal{O}(n_{neigh})$, it can be understood immediately, that the time for the neighbour list based force calculation is $\mathcal{O}(r_c^3)$, simply because the number of neighbours n_{neigh} and relevant atoms can be expected to be proportional to the volume of a sphere ($n_{cell\_nmax} \propto r_c^3$).

In contrast, it appears harder to explain the measured time for the cell list approach. Being quite a surprise, the cell list approach consumes more time than the neighbour list version and thus preforms worse! Therefore, this section will proceed to explain firstly, why the cell lists generally take more time and afterwards the local minima and maxima in the curve.

Even though the cell lists were specifically designed for the GPU, one has to consider both memory access (i.e. reading the other atom's positions) and force calculation time. The memory access is assumed to be 10 times faster for the cell list approach (due to the *coalesced accesses*). However, the calculation is done in parallel by firstly checking if the distance r to another particle is $r \leq r_c$. Whenever one thread finds an atom close enough and evaluates the force formula, the other threads have to wait (even if their distance is too high to contribute to the force) until every thread in the *warp* finished the calculation. Therefore, if $n_{candidates}$ denotes the number of read atom positions $\vec{r}_i$, then the total execution time t_{tot} is in $\mathcal{O}(n_{candidates})$.

Figure 4.5: required memory elements, drafted in 2D

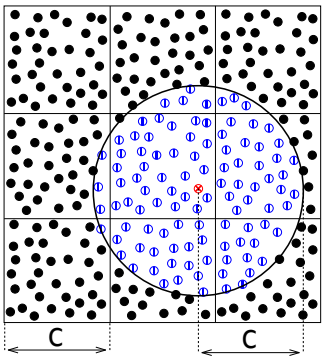


Figure 4.5 illustrate the 3D situation in a 2D sketch: the cell list approach reads (fast) all atom positions from the 27 surrounding cubes, each of edge length $c = 2r_c$, while the neighbour lists only contain atoms within a sphere of radius c. The cell list approach needs to load the data of an entire neighbour cell as soon as one atom in the own cell may interact with an atom in the neighbour cell. The positions of those atoms in the corners of the cube are fetched for nothing from memory. To estimate whether the faster memory accesses can compensate the huge amount of needed memory, a proportion $\gamma \in (0, 1)$ of the time processing a single interaction t_{single} is assumed to be spent on memory accesses.

$$\begin{aligned} t_{\text{total}} &= n_{\text{candidates}} \cdot t_{\text{single}} \\ t_{\text{single}} &= \underbrace{t_{\text{single}} \cdot \gamma}_{\text{memory read}} + \underbrace{t_{\text{single}} \cdot (1-\gamma)}_{\text{actual calculation}} \end{aligned}$$

While the actual force calculation for one interaction is the same for both approaches, the time for memory access and $n_{\text{candidates}}$ vary: the cell list will read all neighbour cells and thus $n^{\text{neighbour}}_{\text{candidates}} \propto 3^3 c^3$, while the NL will read $n^{\text{cell}}_{\text{candidates}} \propto \frac{4}{3}\pi c^3$ atoms (assuming a homogeneous density with the same proportionality factor for both $n_{\text{candidates}}$).

Not to introduce further variables, the time the cell lists need for memory access is simply divided by 10 (because the access is 10 times faster). Since $\vec{F}_{ab} = -\vec{F}_{ba}$ is used, only 13 neighbour cells must be loaded additionally to the own cell. This results in:

$$\begin{aligned} t^{\text{neighbour}}_{\text{total}} &\propto \frac{4}{3}\pi c^3 \cdot t_{\text{single}} \cdot (\ \gamma\ + 1 - \gamma) \\ t^{\text{cell}}_{\text{total}} &\propto 14c^3\ \cdot t_{\text{single}} \cdot \left(\frac{\gamma}{10} + 1 - \gamma\right) \end{aligned}$$

If the cell list approach should be faster (i.e. $t^{\text{cell}}_{\text{tot}} \leq t^{\text{neighbour}}_{\text{tot}}$), then the above equations imply that γ needs to fulfil:

$$\gamma \ \geq\ \frac{1 - \frac{4\pi}{14 \cdot 3}}{1 - \frac{1}{10}} \ \approx\ 78\% \tag{4.1}$$

In other words, the cell list approach has the advantage of very fast memory accesses (≈ 10 times faster), but with the drawback to load much needless memory elements ($\frac{14 \cdot 3}{4\pi} \approx 3.3$ times more than the neighbour lists). If the memory accesses dominate the force calculation time (consuming at least 78 % of it), the cell list can play off its advantage. However, the practical test showed that this high ratio is not even reached for the LJ force, which is computationally less demanding.

Now the extrema of the cell list curve in fig. 4.4 should be explained. For cutoff radii $r_c \leq 3.5\sigma$, the smallest valid cell size of $n_{cell\_nmax} = 32$ atoms per cell was chosen. Though, a usual $r_c = 2.5\sigma$ leaves only about 9 atoms per cell, which wastes more than $\frac{2}{3}$ of the processor power. The smaller the cell, the less atoms are in it - which explains the negative slope in the first part of the diagram (presumably $\propto \frac{32}{r_c^3}$). For the next plotted point at $r_c = 3.7\sigma$, n_{cell_nmax} jumps to the next multiple of 32, which is 64. This again results in much wasted processor power and causes a jump in the required time. For $r_c = 4.85\sigma$ an optimum is reached again, i.e. there are about as many real particles in the cell as there are threads started. By increasing r_c further, one may pass plateaus, because a slightly

varied cutoff radius may result in the same cell size and n_{cell_nmax}. With increasing r_c, the cell lists get close to the time of the neighbour lists. This can be explained by the use of shared memory, which supports the calculation only sparsely for small cells, when little instructions are performed on the data, but starts to play a role when the data of large cells can often be used again. For $r_c = 7\sigma$, both approaches take about the same time. However, cut-offs that large are only of theoretical interest - at least for the LJ force (also see fig. 2.1). Unfortunately, the crossover could not be plotted, because $n_{cell\_nmax} \propto r_c^3$ extends the limit of 512 particles for greater cutoffs. The NL approach is also limited, but instead by the size of the *global memory*. On the tested workstation, the graphics cards' memory of $1\,GiB$ was too small to store all neighbour lists for $r_c \geq 9\sigma$.

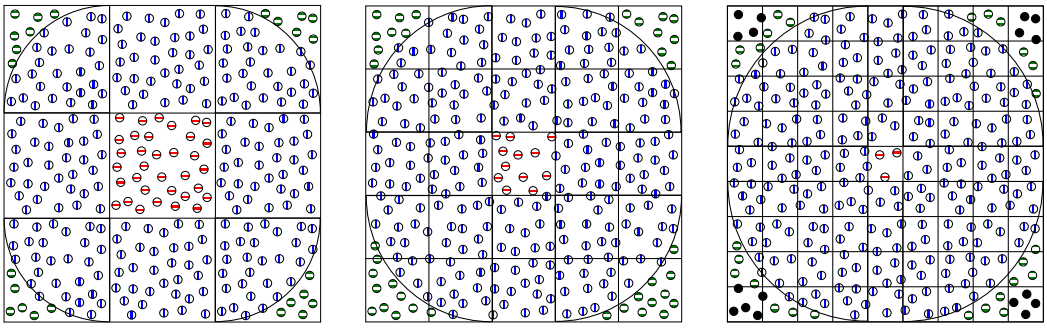
The cell list approach may perform better, if each cell could already be filled for $r_c = 2.5\sigma$. However, due to the repulsive part of the LJ force, the particles will not be much closer than the equilibrium radius $r_{eq} = \sqrt[6]{2}\,\sigma$. (The fact that both r_c and r_{eq} are proportional to σ reasons to plot the above curves over multiples of σ in the first place.) Independently from the algorithm, a higher particle density would be needed. To see an upper bound, a cube with edge length $c = 2 \cdot 2.5\sigma$ can be filled with hexagonal closest packed spheres with radius r_{eq}. In that case

$$\frac{(2 \cdot 2.5)^3}{\sqrt[6]{2}^3} \frac{\pi}{\sqrt{18}} \approx 65 \tag{4.2}$$

atoms could be filled into one cell. Still, this estimate ignores the margin a real algorithm would reserve and moreover, densities being that high are seldom in usual MD simulations of liquids or gases.

In order to avoid the principle disadvantage of the cell lists (i.e. to load to much positions), one could try to enhance the method by using multiple neighbour shells:

Figure 4.6: Multiple neighbour shells, drafted in 2D



(a) one neighbour shell (b) two neighbour shells (c) four neighbour shells

Legend: The particles are coloured as follows: red / horizontally striped marks the current cell. Loaded and sometimes needed particles are blue / vertically striped. The loaded but not needed particles are green / white striped. The solid black particles are neither loaded nor needed.

In that case, the cells in the corners of the above mentioned cube can be skipped during the calculation, which gives the hope to reduce the ratio of needlessly loaded memory. That approach can also handle larger cut-offs, since the same geometrical size is now split over several cell, thus, the limit of 512 particles is reached later. However, a great number of neighbours shells is needed to see an effect (see fig. 4.6) and unless unrealistic large cut-offs are chosen, the cell again gets too small to be filled with enough atoms (at least close to 32)

to be performant. In fact, for the above example and a usual cutoff $r_c = 2.5\sigma$, the use of two neighbour shells decreased the performance by a factor of about 10.

As a small conclusion, the cell list approach works well

- for high particle densities,
- for computationally inexpensive forces and
- for large cutoff radii when simultaneously using multiple neighbour shells.

However, for scenarios usually investigated by MD simulators, these conditions are generally not fulfilled. Even in that case, the cell lists do not seem to have a benefit great enough to justify

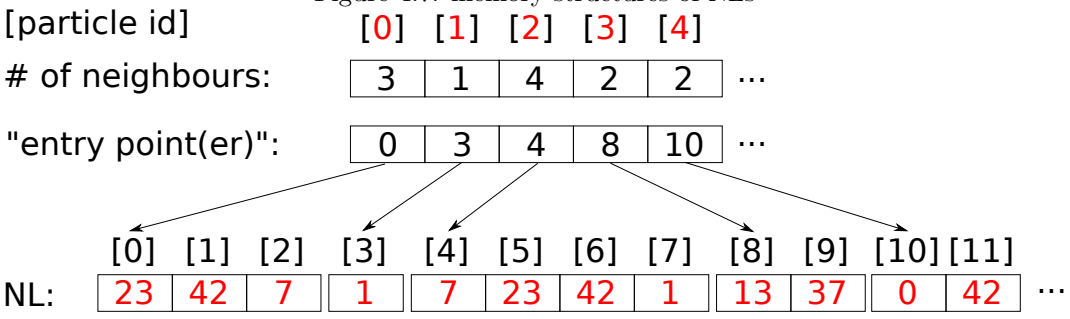
- the need to re-order all data into a new structure for the GPU and copy the data back into the old format whenever the original Lammps need it, and
- the programming effort to re-invent code parts, which could otherwise be adapted from the Lammps code, that already uses neighbour lists.

After knowing the above explained results, the current implementation generally focused on neighbour lists, where the later development additionally made use of a `float4` structure the *texture cache*, increasing the performance by a factor of 1.2.

Building neighbour lists on the GPU

Since the last section revealed NLs as the preferred alternative for force calculation, it is desired to build them directly on the graphics card - not just to have the force calculation completely on the GPU, but especially to avoid time-consuming data transfers to the device and of course, in the hope to speed up the NL build by parallelization. However, the different number of neighbours for each particle brings up some difficulties: the CPU lists all neighbours of the first particle, and always appends a list of neighbours for the next particle to the end of that list, remembering the starting point in this long list for every particle (as a cumulative sum or as pointers) - see fig. 4.7.

Figure 4.7: memory structures of NLs



If the neighbour lists should be build in parallel for every particle, each particle would try to find and store its neighbours. For that reason, the starting point of every list already has to be known before the number of neighbours had been calculated. There are two possible solutions: either reserve a constant maximum number of neighbours n_{neigh_max} (potentially

wasting memory) or firstly start a trial run of the neighbouring algorithm just to count the number of neighbours (wasting time). In the presented implementation the decision was to choose the former alternative, because the density in many studied systems, like incompressible liquids, is homogeneous - thus, each particle has about the same number of neighbours.

To solve the problem of how to choose n_{neigh_max}, an initial small value can be guessed. If and while the succeeding procedure of building the neighbours list fails (because there were to many particles within the neighbour radius), n_{neigh_max} can be increased. Either, it may be increased by a fixed value, which potentially takes many steps, but will find a close approximation - or it may be doubled each time, which will grant for a smaller number of re-allocation steps (logarithmic complexity), but will potentially consume too much extra memory. Since memory is rather limited on a GPU and a re-allocation occurs only seldom in usual scenarios, the increase by a fixed value was chosen. In the current implementation, n_{neigh_max} never decreases.

The number of checks, whether a particle is within the neighbour radius r_{neigh}, can be done with all other particles. Alternatively, it can be reduced by firstly sorting all particles into a cell list - re-using the code developed in section 4.2.2. Still, only the positions are reordered. Afterwards, the check only incorporates close neighbour cells. Just as Lammps does, both methods are implemented and left as a choice to the user.

4.2.3 Long-range pair forces

Regarding the long-range forces, the GPU implementation is based on Lammps' existing code and limits itself to the PPPM algorithm, disfavouring the direct Ewald sum (see section 2.2.3). Some parts of the code can be parallelized in a straight-forward manner, e.g. by delegating operations on every particle (or every grid node) to a Cuda thread, which had been done inside a loop in the original Lammps code.

Thanks to Nvidia, there is a program library called *CUFFT*, which can compute an important part of the PPPM algorithm - the FFT - on the GPU. Using MPI, Lammps also parallelizes the FFT over multiple CPU processes. That topic will be dealt with in the next section while discussing the communication. Having that said, only one detail of the PPPM algorithm should be highlighted to the end of this section, which is the creation of the charge grid.

A possible implementation would start a thread for every grid node and try to find all particles within the critical distance and add the effective charge (in case of the Coulomb potential) to its *own* grid data element. However, this idea is less performant, e.g. because it would result in useless read accesses from *global memory* in almost all cases, since the effective charge is only influenced by a few surrounding particles. Therefore, the alternative idea is to stay close to the CPU implementation and start a thread for every particle and distribute its charge over those grid nodes close to the own position. However, this emerges to be problematic, since it potentially causes write conflicts and *atomic functions* are only supported for integer values (see section 3.4.1), while effective charges are rational numbers (`float` or `double`). To overcome this dilemma, atomic functions are used anyway by implementing a fixed-point arithmetic, i.e. every rational numbers is multiplied by a large integer $L \in \mathbb{N}^+$, in order to move prior decimal places into the computational range of integers and after the calculation, each number is again divided by L. Furthermore, L is chosen adaptively by determining the highest significant bit in the result.

4.3 Avoiding data transfers between host and device

So far, the implementation of the force calculation has been parallelized. In term of Amdhal's Law (see 2.10), i.e. often executed code-parts were chosen in order to get a proportion φ of parallelized instructions (as close as possible to 1). Since the number of processors N can not be influenced by software development, the other critical parameter left is the overhead ratio τ, which is mainly caused by communication between CPU and GPU. This section will therefore discuss methods to decrease τ (and increase φ). Although Amdahl's Law oversimplifies the situation, because it was only derived for a set of equal processors, it will be used in this discussion for CPUs and GPUs as well, because the trends are still the same.

The first overhead coming to mind is the time spent waiting for data transfers between host and device every timestep, e.g. the GPU-calculated forces must be downloaded to numerically integrate them to new velocities and positions and upload this data again for the next timestep. In order to get an impression whether it is worth to do something about these data transfers, the times spent on calculation and data transfers are compared. From the already mentioned LJ melting example (4.2.2), the following rough estimates can be made, which are neither exact nor general, but give an idea about the orders of magnitude. (In that example, the force calculation takes $\varphi \approx 80\%$ of the total time.) For further approximations, the average time in seconds per atom and timestep ("$\frac{s}{astep}$") is token. It can easily be measured that approximately 1 $\frac{\mu s}{astep}$ is spent for the force calculation on the CPU and 50 $\frac{ns}{astep}$ on the GPU. When a PCIe transfer speed of $2\frac{GiB}{s}$ is assumed, it takes $12\,\frac{ns}{astep}$ to transfer 24 Bytes for positions and forces (2 vectors with each 3 components, each 4 Bytes for `float`). Likewise, it would take 24 $\frac{ns}{astep}$ for `double`. Additionally, each data transfer between GPU and CPU has a latency (caused by hardware and software delays)[6] and in case of using single precision on the GPU, the time spent for the conversion between `double` (Lammps) and `float` (Cuda) must be considered, as well. In other words, the data transfer already takes about 30% - 50% of time for the force calculation on the GPU. Hence, at some point it appears difficult to improve the force calculation further, but in this situation it can be seen from Amdahl's Law, that if the overhead τ was reduced e.g. to $\frac{\tau}{10}$, it would have the same speed-up effect as making the force calculation three to five times faster.

The data (e.g. positions, velocities, forces) needs to be downloaded from the GPU to the CPU for three purposes:

- to modify the data (e.g. do the integration),
- to extract desired (macroscopic) properties (e.g. the current pressure) out of the microscopic data and
- to synchronise data with other CPU processes of Lammps.

Thus, in order to avoid the data transfers, more than just the force calculation has to be done on the GPU. Even if that implies more programming work, the above estimates promise that it is worth the effort. This section will discuss the implementation of these three aspects.

[6] The appendix 7.4 lists a few latency- and bandwidth benchmarks.

4.3.1 Modifications

The modification are discussed on the example of the different integrators that Lammps encapsulated into C++ classes, each for the one of the ensembles like NVE or NVT (see section 2.3). One can now easily inherit from these classes and override the computing routines with methods that call a corresponding Cuda kernel. As each thread of the kernel can work on independent data and Lammps has already implemented each desired integration algorithm (e.g. Verlet) in C++, the porting is straight-forward. As a side note, the resulting new classes are always simply referred to e.g. as `verlet/cuda`, `nve/cuda` and `lj/cut/cuda` in Lammps' input scenario file.

Unfortunately, the resulting kernel is - in a manner of speaking - too simple: once the positions, velocities and forces have been fetched from *global memory*, only little operations are performed on the data, thus the long latencies can not be hidden. Nevertheless, even if the integrators are *only* about five times faster on the GPU, just the avoidance of data transfers already justifies the effort.

4.3.2 Computations

The computation of system properties is a little less trivial, since e.g. temperature and pressure are based on a sum over all particles (see section 2.4.2), which would produce write conflicts (see section 3.4.1) in a naive implementation. Still, a sum

$$s := \sum_{i=0}^{N_0-1} a_i$$

can be parallelized using the following iterative idea: $N_{r+1} := \lceil \frac{N_r}{2} \rceil$ threads are started in round r, where each thread i adds $a_{i+N_{r+1}}$ to a_i. Every consecutive round, only the first half of the remaining threads start the calculation, the others are left idle, until finally the trivial case has been reached. In case N_0 is initially rounded up to the next power of two, this trivial case is one element (which is equal to s). Otherwise, the final iteration may also perform a sequential summation of the last few elements.

With that algorithm, desired (macroscopic) properties can be calculated on the GPU in $\mathcal{O}(\ log(N_0)\ )$ and only a few bytes must be transferred, e.g. every 10 timesteps (based on the input file specifications), which does significantly reduce τ.

4.3.3 Communication

As mentioned before (section 2.6), Lammps uses spacial decomposition and an MPI based parallelization to communicate between the different CPU processes of a Lammps program run. Because one CPU process supervises one graphics card, this communication is also needed to use multiple graphics cards. For it, a buffer is transferred to the other Lammps processes, that contains

- positions (and forces) for those particles, which are close the border of the sub-box of the own process (so-called *ghost atoms*) and
- all data from atoms moving into or out of the sub-box.

Since the current implementation generally stores all data inside the graphics card's memory - for the reasons explained above - it is wise to build the buffer directly on the GPU - and afterwards only transfer this smaller set of data through to the CPU host in order to share it with other Lammps processes.

As usual, Lammps has a *C++ class* for communication (`Comm`). However, unlike forces and integrators, the communication class can not be chosen by a user script and is instantiated only once in a program run. Therefore, it is not possible to inherit from the original `Comm` class or differently base a new class on it. Instead, the `Comm` class had to be modified directly to call appropriate kernels.

The communication buffer itself is built in two steps. At first, the indices of the atoms needing to be transferred are determined by calling a kernel for each particle and building the list (using *atomic functions* - similar to the idea in section 4.2.2). Of course, the kernel needs to be aware of the simulation box geometry - again, the *constant memory* can be used. Afterwards, as many threads are started as there are items in the above list and the needed data is copied into the buffer in parallel. Still, some particles may carry addition information, like the charge. To let the device code know which information to include into the buffer, new atom classes were introduced.

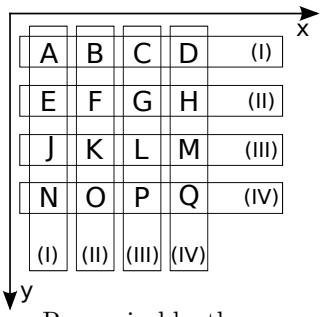
As mentioned in section 4.2.3, another component of Lammps communicating data is PPPM. Even without giving further details about the PPPM method, it is clear, that the Fourier transform of a 3D charge density ρ will be based upon the formula

$$\mathcal{F}(\rho_k(\cdot)) \propto \int_{\mathbb{R}} e^{-2\pi i k_z z} \left(\int_{\mathbb{R}} e^{-2\pi i k_y y} \left(\int_{\mathbb{R}} e^{-2\pi i k_x x} \rho(x,y,z)\,\mathrm{dx} \right) \mathrm{dy} \right) \mathrm{dz}\,. \tag{4.3}$$

Obviously, the three integrals need to be calculated one after another - while of course, the FFT will replace the integrals with discrete sums over the grid of effective charges.

Lammps now uses the following parallelization strategy: it simply splits the calculation in each dimension over the number of involved processes and sends the result (generally being a function, in this case given by all discrete values on the grid) to the other processes so they can perform the calculation in the next dimension.

Figure 4.8:



To give a clarifying example, four processors are assumed and the idea is sketched in 2D (fig. 4.8). In the first step, process (I) calculates the effective charges for all grid nodes in the range of A, B, C and D, afterwards performing the FFT in the x-direction. Likewise, process (II) does the same for E, F, G, H; process (III) for J, K, L, M and process (IV) for N, O, P, Q. Afterwards the results of this calculation is needed for the FFT in y-direction. Hence, process (I) has to receive the grid data from the cells E, J and N and sends the data from B, C and D to the corresponding other processes (by the help of an MPI library).

Recognisably, the communication overhead can be immense compared to the time needed by the GPU-accelerated (CU)FFT - especially, since the data also needs to be transferred from the device to the host and vice versa. Later benchmarks revealed, that - speaking in terms of Amdahl's Law - the overhead τ can even be large enough to reduce the speed-up S to a value smaller than 1, i.e. PPPM would be slower when using multiple processes. Therefore, the presented implementation does not parallelize the FFT over multiple GPUs, but instead initially sends the complete data to every graphics card and makes each of them calculate the complete FFT alone. In other words, *data replication* (see section 2.5) was used.

Chapter 5

Benchmark and precision analysis

This chapter will list benchmarks of the presented implementation and start with the presumably most interesting property: the speed-up. Afterwards, the performance is investigated for multiple processes, where communication starts to matter. The speedup and the scaling is also shown for the different parts of the program. Since the GPU-accelerated Lammps is not only wanted to be faster, but also to produce the same results, the last part of this chapter will compare the output of the GPU-aided version to the original Lammps and analyse influence of different (`float` or `double`) precisions.

All benchmarks are done with code based on the neighbour list approach.

5.1 Benchmark

Three different scenarios are benchmarked, which are considered to represent different and typical MD simulations:

1. **Lennard-Jones Poisson Flow.**
   A two-dimensional LJ liquid is put between a fixed upper and lower wall. In the other (x) dimension, the system is periodic. The lower wall is forced not to move, while the upper wall is accelerated in negative x direction. The LJ liquid ($r_c \approx 1.2\sigma$) is accelerated in x direction and its temperature is hold constant. Lammps requires an additional function to force the system to stay in 2D. The enforcement of the these constraints takes about 42% of the total simulation time on the CPU.

2. **Lennard-Jones Melt.**
   A three-dimensional periodic box is started from an fcc lattice and the atoms get an initial kinetic energy high enough to leave their lattice places. Using a LJ force with cut-off 2.5σ and a Verlet NVE integrator, the melting of the system is studied. On the CPU, approx. 80% of the time is used for the force calculation.

3. **Silicate Buckingham.**
   In a three-dimensional periodic box, the transport of Lithium ions in a Si-O system is studied. The three particle types interact with a short-range Buckingham potential ($U(r) := A \exp\left[-\frac{r}{\rho}\right] - \frac{C}{r^6}$ for $r < 10$ Å, 0 otherwise) and a Coulomb interaction. The parameters A, C and ρ are chosen to represent a real system. To calculate the long-range interaction of the charged particles, the PPPM method is used, whose execution takes roughly half of the simulation time.

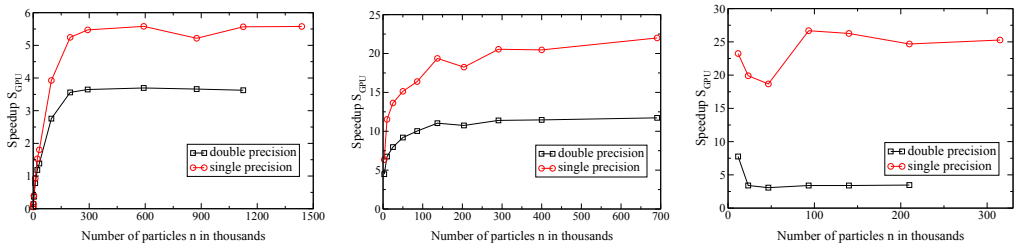
The amount of time spent for each part of the simulation has already been mentioned in section 4.1.3.

5.1.1 Speedup

Total runtime speedup depending on system size

In the first test, the speed-up $S_{\mathrm{GPU}} := \frac{t_{\mathrm{CPU}}}{t_{\mathrm{GPU}}}$ for the different scenarios is plotted depending on the system size, i.e. the factor by which the total program execution time is shrunk when using the GPU-aided Lammps version.

Figure 5.1: Speedup for different scenarios, 1 CPU vs. 1 GPU



(a) LJ Poisson Flow (b) LJ Melt (c) Silicate Buckingham

All CPU times are measured on workstation 7.2.1 (see appendix). (a) and (b) are done with a GTX 280 having 1 GB global memory, while (c) is done with an Tesla C1060, because it has the needed 4 GB RAM to test that number of particles. Both cards use the same graphics chip with the same clock rate, while the Tesla card has slower memory access. The shown peaks are reproducible on every system.

In the above examples - using Cuda graphics cards - the total Lammps runtime can be reduced by a factor between 3 and 10 in double precision, and between 5 and 27 in single precision. The factor strongly depends on the scenario and appears to be reasonable for a general-purpose program like Lammps. Though, it should be noted that the speed-up for pure force calculation is generally higher (up to $S_{\mathrm{GPU}} \approx 40$ as detailed in the section below).

Because the GPU can not fully use its processor capability for small systems and due to an $\mathcal{O}(1)$ overhead, the speed-up is generally better if the system is larger. The abscissa in the above plots could not be extend much further, since the number of particles is limited by the GPU memory (mainly needed to store the neighbour lists). Multiple GPUs can overcome this limitation by providing more total memory, which is also one of the reasons why their scaling is investigated in the next chapter.

Since each MP has only one unit for double precision arithmetics and because calculations in double precision need to load about twice as much data from global memory, single precision calculation can generally be expected to have an advantage of at least factor two.

In the Poisson Flow example, the speedup is relatively small and GPU version is even slower than the original Lammps version for less than approx. 10^4 atoms. Many simple additional constraints are required by the input script, that cause the call of many different Cuda kernels. They perform only very few operations on the data, so that the memory access latencies can not be hidden. Each of the short kernels has to face this problem again, which encumbers performance. Double and single precision versions are less than a factor of two apart, which supports the argument that the executions is neither limited by calculation speed nor memory bandwidth, but memory access latency.

When doing the 5.1(b) benchmark on the same card as 5.1(c), the speedup drops noticeably (down to a factor of $\frac{2}{3}$). Since both cards perform the pure calculations with the same speed, a reasonable explanation is that the memory access again dominates the execution time. Therefore, more computational intensive forces than LJ may see better speed-ups,

simply because the CPU calculation time would increase and the GPU could execute some more instructions while waiting for the next memory element.

Since the speedups for the Silicate Buckingham and the Melt example are about the same in single precision, it can be implied that the PPPM method is accelerated roughly as good as the LJ force calculation.

The speedup in the 5.1(c) example shows a non-monotone behaviour in the left half of the plot, which is not explained by the pair force calculation, but by the PPPM computation using FFT. In its common implementation, the FFT algorithm is based upon the factorisation of the grid dimensions into prime numbers. The performance may crucially depend on the size and number of these factors, which are - in a sense - not monotone in $\mathbb{N}$.

Speedup of different parts

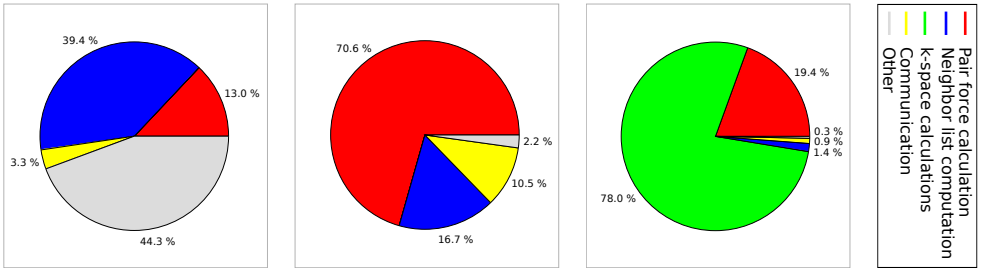
While the last section only calculated the total runtime speedup, this section will now investigate how much time is needed by the different parts of the GPU-aided program and compare them against the CPU version (fig. 4.1), also to give a better explanation of the above results.

In further diagrams, a fixed number of particles is chosen for each of the three scenarios, where the speedup is already close to its maximum.

	LJ Poisson Flow	LJ Melt	Silicate Buckingham
Number of atoms n	$1.1 \cdot 10^6$	$6.9 \cdot 10^5$	$9.3 \cdot 10^4$

Figure 4.1 already showed the proportion for *two* CPUs, which is why figure 5.2 gives a first illustration of the proportions in single precision also for *two* graphics cards. This makes it possible to fairly compare communication times and give a general impression which program parts are important.

Figure 5.2: Time proportion for sample runs on 2 GPUs in single precision

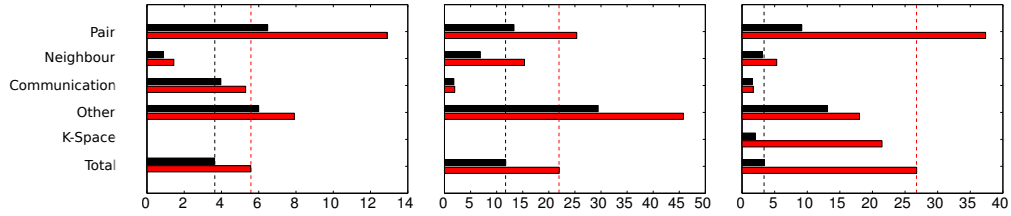


(a) LJ Poisson Flow (b) LJ Melt (c) Silicate Buckingham (d) Legend

The fact that the proportion of the force calculation generally shrunk in comparison to two CPUs (fig. 4.1) implies that the this part could be accelerated the most. While the proportions in the Melt example look similar to those in fig. 4.1, the Silicate Buckingham example is now clearly dominated by the time for PPPM method (with FFT). In the LJ Poisson Flow example, the neighbour list creation now takes more than $\frac{1}{3}$ of the time, which means the acceleration of this part is comparably small.

The proportions look quite similar for double precision, which can be seen from fig. 5.3 showing the speedup detailed for every program part, for the execution of 1 GPU in comparisons with the 1 CPU:

Figure 5.3: Speedup S_{GPU} of different program parts



(a) Legend (b) LJ Poisson Flow (c) LJ Melt (d) Silicate Buckingham

The black bars show the speedup in double and the red bars in single precision. The dashed vertical lines mark the total runtime speedup.

As expected, the speedup is generally greater in single precision for all parts. The force calculation and the integration ("Other") experience the best speedup, because the paradigm to perform a single instruction on multiple data fits best in these cases. In contrast, the neighbour list creation performs worse, because each particle corresponds to a thread, which loads another position for the distance check (long-latency access from global memory), but performs only little operation on the data. Therefore, its speedup is always below the total runtime speedup and its time proportion in fig. 5.3 grow in comparison to fig. 4.1 - most noticeable in the LJ Poisson Flow and Melt example. Since the Melt example is dominated by the force calculation just as the Silicate Buckingham example is by PPPM, it is logical that the total speedup of these examples is not much different from the speedup of those parts. As guessed above, the FFT causes the Silicate Buckingham example to perform badly in double precision, because the dominant FFT part obviously performs much better in single precision. The speedups for communication, neighbour lists and other are less important in the Silicate Buckingham example, since they take only about 1% of the runtime.

Time proportion of data transfer

Preliminary considerations in section 4.3 already reasoned the decision to do more than the force calculation on the GPU. To the end of this section, the program execution time is benchmarked when doing the numerical integration on the CPU, which also needs data transfers between host and device. Technically, this is achieved by simply changing "fix nxt/cuda" back to "fix nvt" in Lammps' input file. The program detects a required operation which cannot be done on the GPU and therefore downloads all data ($\vec{x}$, $\vec{v}$, $\vec{F}$, atom types) from the device, performs the operation on the CPU and uploads all data again. In the Melt example, this procedure increases the time for the numerical integration from about 2% to 95% of the total execution time, decreasing the total speedup S_{GPU} from 22 to 2. If the program had known that the integration only needs to download forces and upload positions, the additionally required time would have been about eight times smaller - thus, having a the total speedup $S_{\mathrm{GPU}} \approx 10$. In conclusion, it is however definitely worth having the integration done on the GPU.

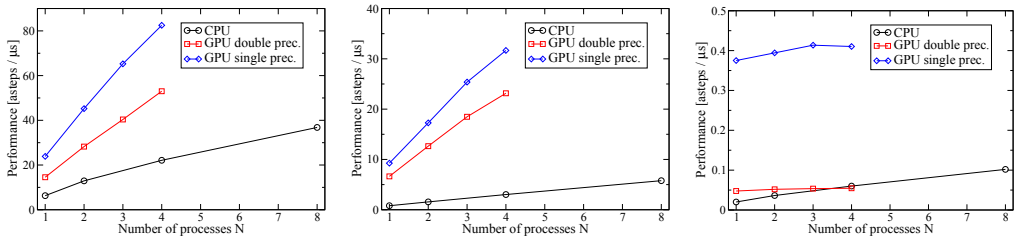
5.1.2 Scaling

While the above benchmarks only compared the times for one process, this section will look at the performance scaling of multiple graphics cards, also compared to multiple CPU cores. As stated before, each graphics card is governed by one CPU process and an MPI library is used to communicate between the processes in either case.

Performance depending on the number of processes

The performance of the three scenarios is measured by the number of atoms and simulation steps which can be done per execution time ("$\frac{astep}{\mu s}$").

Figure 5.4: Performance for multiple GPUs and CPUs



(a) LJ Poisson Flow (b) LJ Melt (c) Silicate Buckingham

The speedups of the last section are found again in the relation of the values at $N = 1$. The trend of GPUs showing more performance than CPUs and being better in single than in double precision generally continues for multiple processes, as well. However, example 5.4(c) scales very badly, while 5.4(a) and 5.4(b) look promising. The main difference from 5.4(c) to the other two examples is the use of PPPM, which is therefore assumed to cause the bad scaling. The next section will analyse the different program parts in more detail.

It is difficult to extrapolate additional performance information for greater N out of the plot (e.g. by just looking at the slope), because firstly, the performance may scale differently when off-board communication (i.e. network transfer between multiple PCs) starts to matter and secondly, the functions do not continue linearly. To get a better impression on the scaling, the speedup $S_N := \frac{t_{\text{one process}}}{t_{\text{N processes}}}$ is calculated[1]. $S_N \leq N$ can be expected.

S_N	LJ Poisson Flow	LJ Melt	Silicate Buckingham
2 CPUs	2.051	1.953	1.813
4 CPUs	3.512	3.782	2.995
8 CPUs	5.846	7.231	5.080
2 GPUs (single prec.)	1.895	1.865	1.051
3 GPUs (single prec.)	2.734	2.742	1.103
4 GPUs (single prec.)	3.456	3.421	1.093
2 GPUs (double prec.)	1.942	1.905	1.091
3 GPUs (double prec.)	2.776	2.786	1.127
4 GPUs (double prec.)	3.645	3.491	1.147

As expected, the benefit from using one additional process decreases with the number of

[1] While S_{GPU} gave the factor by which one GPU-aided simulation is faster than one CPU simulation, the speedup S_N in this section refers to the time factor between multiple CPU cores and a single core, or, between multiple graphics cards and only one card.

processes. The Poisson Flow and the Melt example scale about equally on the GPU, but still the (relative) scaling is worse than on the CPU. This may be due to the same reason why double precision calculations seem to show a little better speedups: the overhead (e.g. delay for cross-process data transfers) is about the same in absolute values and therefore its influence is the larger the smaller the calculation time.

Although it was used before only to guide the discussion, this scenario now really has equal processes, so that Amdahl's Law (2.10) provides a fitting function. To compare the effectiveness of CPU and GPU parallelization, the parameters φ and τ are extracted:

		LJ Poisson Flow	LJ Melt	Silicate Buckingham
CPUs	φ	94.8 %	98.4 %	98.4 %
	τ	$5.3 \cdot 10^{-6}$	$5.0 \cdot 10^{-9}$	$8.4 \cdot 10^{-7}$
GPUs (single prec.)	φ	95.0 %	94.8 %	16.1 %
	τ	$3.5 \cdot 10^{-3}$	$6.2 \cdot 10^{-6}$	$2.7 \cdot 10^{-2}$
GPUs (double prec.)	φ	95.4 %	96.6 %	17.6 %
	τ	$9.6 \cdot 10^{-6}$	$2.7 \cdot 10^{-6}$	$4.7 \cdot 10^{-3}$

The two parameters, fitted by three sample points, suggest that the overhead takes only about one millionth of the execution time and the GPU-accelerated program is able to parallelize about 95 % of its execution, while the Silicate Buckingham example has a higher overhead and a much smaller $\varphi \approx \frac{1}{6}$. In comparison, the CPU version parallelizes up to 98%. Based on that numbers, a high performance computer with many GPUs could be up to $S_\infty = 20$ times faster than one GPU (which is again a factor of up to 25 faster than one CPU), when the overhead τ is ignored. However, τ is less than the communication time measured by Lammps' internal timers (which report about 2 %), suggesting that the simplest performance model given by Amdahl's Law does not precisely describe the situation. On the one hand, both φ and τ will depend on N and on the other hand, additional effects may play a role for the scaling, e.g. if less atoms are simulated per GPU, the (texture) cache can be used more effectively, which may even result in some $S_N > N$. The behaviour is hard to predict theoretically and can probably be revealed only by large scale benchmarks for more than four graphics cards, which could also incorporate network communication, the behaviour of $\tau(N)$, cache usage and other effects. This task is left for future studies.

Performance under load

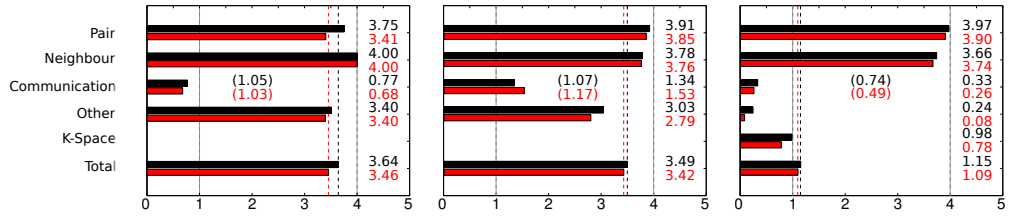
It should also be noted that the performance of the CPU executions can be negatively influenced by other processes running on the system, which is a typical scenario on multi-user PC clusters. To simulate this situation, the above LJ Melt example was run on a quad-core CPU two times: the first time while the PC was on idle and the second time with three other identical Lammps processes running. While still every process had its own CPU core, the program needed about twice as long (factor ≈ 2.07). Presumably, the processes had to share their memory bandwidth over the main-board. On the over hand, the performance of four concurrently running GPU-aided processes is not measurably influenced, since each simulation is done on an independent graphics card with its "own" memory and memory bus.

This effect may be smaller on modern high performance main-boards.

Scaling of different parts

To the end of this section, the speedup S_N of the different program parts are compared between one and four GPUs, while the numbers of particles n are still the same as in section 5.1.1.

Figure 5.5: Speedup S_4 of different parts on one GPU



(a) Legend (b) LJ Poisson Flow (c) LJ Melt (d) Silicate Buckingham

The black bars show the speedup in double and the red bars in single precision. The total speedups S_4 are again marked with vertical dashed lines. The maximum is predicted with $S_4 = 4$, while parts with speedups $S_4 < 1$ consume even more time when using four GPUs.

Figure 5.5 shows that $3.4 \leq S_N \leq 4$ in the LJ Poisson Flow and Melt example, when using four GPUs for all parts - except the communication, whose time grew in comparison to the execution with one GPU. The numbers in brackets show the scaling in comparison to *two* GPUs. While the latter two examples are considered to have a good scaling, the Silicate Buckingham has not shown a significant performance boost by the use of multiple GPUs in figure 5.4 already, for which the PPPM method can now be made responsible. The parallelization of PPPM gets into the dilemma that on the one hand, if the FFT is done on every GPU alone again - like in the current implementation - the factor $S_N \approx 1$ is clearly no surprise, but on the the other hand the FFT gets a significant speedup when using the GPU (at least in single precision), which is why the communication time would always be greater than the FFT calculation time, if the FFT was parallelized over multiple GPUs. Future research in MD may look for methods scaling better than PPPM for the calculation of long range forces.

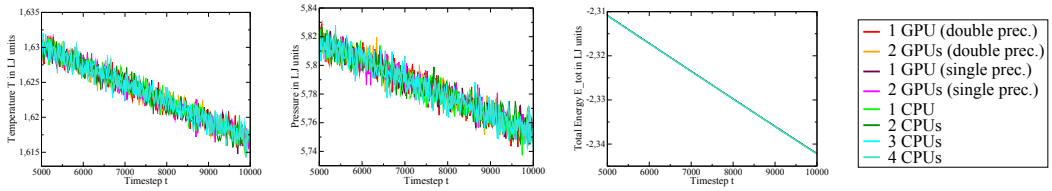
5.2 Precision analysis

While the last section investigated program timings, it is also important to make sure that the accelerated program still computes the correct results, which will be analysed in this section. Only the *LJ Melt* example is detailed here, because the other examples show similar behaviour and would not provide the reader with more information.

5.2.1 Why compare thermodynamic properties?

Unfortunately, the correctness of the implementation cannot be shown by comparing the output byte by byte, because the microscopic data of any simulation already varies when executing Lammps on different CPU processor configurations. Although the same mathematical operations are computed, they are generally not associative, i.e. numerically e.g. $a \cdot (b \cdot c) \neq (a \cdot b) \cdot c$, for floating point numbers. Therefore there is an numerical error, which can cause the calculated trajectories of the chaotic n-body system to drift from the results of another execution. Operations are executed in another order, e.g. when two CPUs partition the simulation area or when the GPU adds forces in another order than the CPU. For that reason, the position output (with eight decimal places) already differs between CPU and GPU after a few hundred time steps. Still, MD simulators believe that despite the microscopic differences, all simulations principally produce the same macroscopic behaviour. Therefore, the following discussion will only look at the evolution of macroscopic (thermodynamic) properties over time, e.g. temperature T, pressure p or total energy E_{tot}.

Figure 5.6: Macroscopic properties over time



(a) Temperature T (b) Pressure p (c) Total Energy E_{tot} (d) Legend

The GPU lines are plotted in a shades of red, the CPU lines in shades of green.

Figure 5.6 shows a zoomed-in view of the above mention properties and should only provide the following quantitative impressions:

- Both CPU and GPU programs output the same trend of the macroscopic variables.
- Different processor configurations do not produce the exactly same results.
- The deviation between GPU to CPU executions seems not to be larger than those between different CPU runs.

While only a small interval was plotted above, these three statements hold true for every tested number of timesteps. For the first approx. 1000 timesteps, the calculated pressure is the same for one CPU and one GPU in double precision, when comparing all eight outputted decimal places.

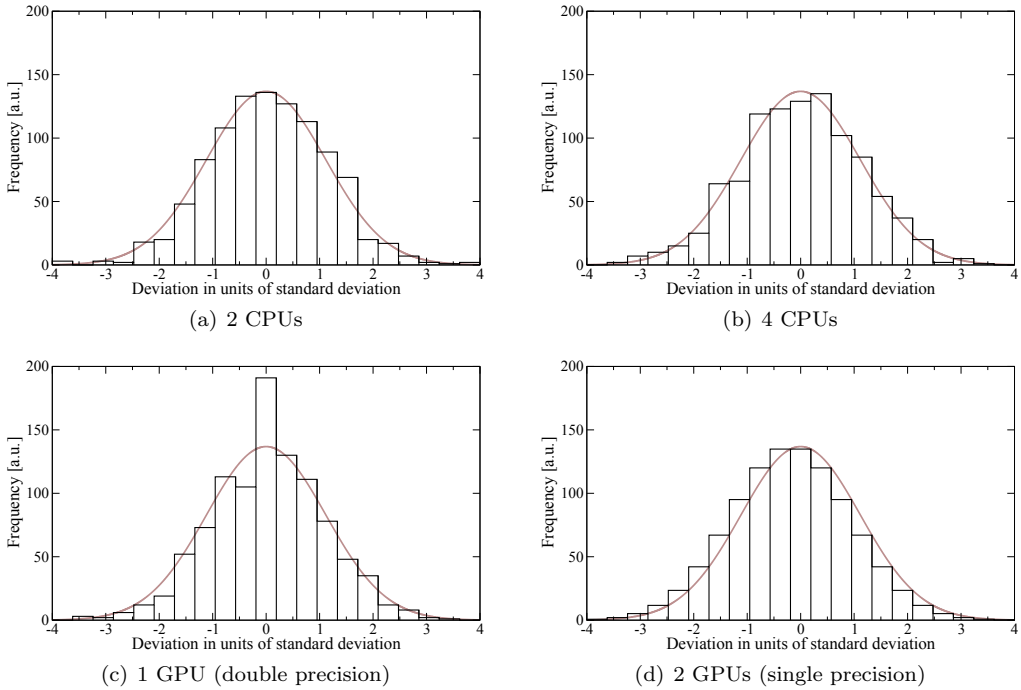
5.2.2 Deviations

To study the differences further, it appears to be better to investigate the relative error instead of the absolute values, taking the execution of *1 CPU* as reference. As the pressure p shows the largest deviation (also due to the way of its calculation - see 2.4.2), it is assumed to be the most interesting property and is therefore picked to be studied in detail. In the example, the system approaches an thermodynamic equilibrium after about 1000 timesteps. For the following t_{steps} timesteps, the standard deviation of the reference data for one CPU σ_{ref} can be calculated and the deviation $\Delta p := p_i(t) - p_{\text{ref}}(t)$ can be calculated[2] and scaled to σ_{ref}, which is done in figure 5.7 for four different configurations. When plotted over time, the absolute value of Δp is always smaller than 4 σ_{ref} during all tested timesteps, and does not noticeably grow over time. However, a direct plot of $\Delta p(t)$ looks just similar to random static, which is why histograms are plotted instead. To give a visual reference, each of the diagrams has the same Gauß-function as background.

$$\overline{p}_{\text{ref}} := \frac{1}{t_{\text{steps}}} \sum_{t=0}^{t_{\text{steps}}-1} p_{\text{ref}}(t) \tag{5.1}$$

$$\sigma_{\text{ref}} := \sqrt{\frac{1}{t_{\text{steps}}-1} \sum_{i=0}^{t_{\text{steps}}-1} \left(p_{\text{ref}}(t) - \overline{p}_{\text{ref}}\right)^2} \tag{5.2}$$

Figure 5.7: Histograms of the pressure deviation $\frac{\Delta p}{\sigma_{\text{ref}}}$



(a) 2 CPUs

(b) 4 CPUs

(c) 1 GPU (double precision)

(d) 2 GPUs (single precision)

[2]The index i corresponds to the eight different configurations (1 CPU, 2 CPUs, 3 CPUs, 4 CPUs, 1 GPU single prec., 2 GPUs single prec., 1 GPU double prec., 2 GPUs double prec.).

The deviations in figure 5.7 are centred to 0 and have a width of approximately σ_{ref}, which means that the GPU calculations show no drift away from the CPU reference and most of the differences Δp are within $[-1\,\sigma_{\text{ref}}\ ,\ 1\,\sigma_{\text{ref}}]$. In double precision, a deviation close to zero is more frequent than in the CPU runs.

To make the above qualitative results more quantitative, the difference of the GPU data from the reference (1 CPU) should now be compared to the deviation between the $N_p = 4$ CPU outputs, which are considered to be correct. Therefore, each timestep t is associated with with a one-point average $\overline{p}_{\text{one}}(t)$ and a one-point deviation $\sigma_{\text{one}}(t)$:

$$\overline{p}_{\text{one}}(t) \quad := \quad \frac{1}{N_p} \sum_{i=1}^{N_p} p_i(t) \tag{5.3}$$

$$\sigma_{\text{one}}(t) \quad := \quad \sqrt{\frac{1}{N_p - 1} \sum_{i=1}^{N_p} \left(p_i(t) - \overline{p}_{\text{one}}(t)\right)^2} \tag{5.4}$$

A trajectory deviation σ_i (taking the 1 CPU run as reference) can be compared to the average deviation $\overline{\sigma}$:

$$\overline{\sigma} \quad := \quad \frac{1}{t_{\text{steps}}} \sum_{t=0}^{t_{\text{steps}}-1} \sigma_{\text{one}}(t) \tag{5.5}$$

$$\sigma_i \quad := \quad \sqrt{\frac{1}{t_{\text{steps}} - 1} \sum_{t=0}^{t_{\text{steps}}-1} \left(p_i(t) - p_{\text{ref}}(t)\right)^2} \tag{5.6}$$

For the pressure p in the LJ Melt example, this results in the following data:

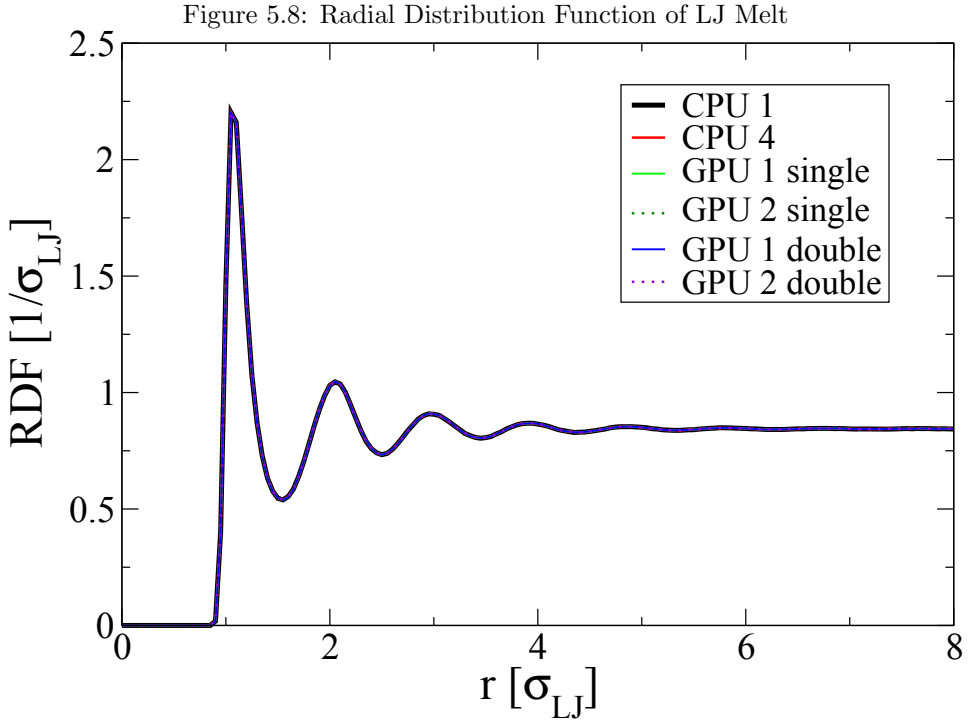
Configuration	σ_i / $\overline{\sigma}$
2 CPUs	1.487
3 CPUs	1.508
4 CPUs	1.542
1 GPU single prec.	1.441
2 GPUs single prec.	1.551
1 GPU double prec.	1.441
2 GPUs double prec.	1.510

Obviously, the deviation σ_i from reference (1 CPU) is larger when using more processors, but there seems to be no difference in the accuracy of the results comparing CPU and GPU data - even no significant difference between single and double precision.

5.2.3 Radial distribution function

It is also possible to compare other physical properties, like the radial distribution function[3], which is calculated out of the final positions of the particles in the LJ Melt simulation.

Figure 5.8: Radial Distribution Function of LJ Melt



The computed radial distribution functions are very close to another, with almost no visible differences.

5.2.4 Conclusion

All things considered, it can be concluded that both GPU accelerated and normal CPU version of Lammps produce the same (macroscopic) results within an acceptable range of microscopic fluctuations, which are in the same order of magnitude as the differences between different CPU executions. Single precision calculations perform faster, but come with the theoretical risk of greater numerical error. However, the precision analysis of the example above provides no reason to prefer double precision. For this reason, an analysis of mixed precision calculations is not done, as it is expected to produce very similar results, with execution times between double and single precision.

While the discussion in this chapter was only focused on the LJ Melt example, the other two mentioned examples show very similar results.

[3]See appendix 7.5 for the used definition.

Chapter 6

Conclusions

This final chapter sums up the results of this GPU parallelization approach by mentioning virtues and drawbacks, reviewing general impressions and giving an outlook to further development.

6.1 Virtues

The most computational intensive part of MD turned out to be the force calculation, being well-suited for parallelization, because the same instructions are carried out on an large array of identically structured data. Using Nvidia's Cuda technology, the general-purpose MD program Lammps could be extended to make use of graphics card's processors, while keeping its C++ design patterns and its flexibility. The principle work-flow of Lammps could be adapted - thus, there was no need to re-invent existing procedures. At the same time, the force calculation could be accelerated by a factor between 15 and 40 and thus, a total runtime speed-up from factor 5 to 30 could be achieved - depending both on simulation and PC system parameters. In double precision, the total program runtime was between two and four times longer.

For short-range pair forces, computations based on neighbour lists were compared against a cell list approach. Although both methods worked in principle, the neighbour list approach was finally chosen because a benchmark revealed it to be faster in usual scenarios and the resulting Cuda code is closer to the original Lammps design, which did not only simplify development, but also aids future code maintenance.

Long-range pair forces needed a different treatment: the PPPM algorithm was ported from Lammps using the Cuda library for fast Fourier Transforms (CUFFT).

Since the data transfers between CPU and GPU toke a significant amount of time compared with the time spent for the GPU force calculation, much better speed-ups could be achieved by avoiding these data transfers. This could be accomplished by doing other parts of the MD simulation on the graphics card as well. In particular, the modifications (e.g. the numerical integration), the computation of (macroscopic) properties (e.g. pressure or temperature) and the construction of the synchronisation buffer for other CPU Lammps processes, was ported to the GPU.

Benchmarkes showed that one graphics card can already replace a modern oct-core processor in common scenarios. The availability of high performance PCs (which can take up to 16 GPUs), the comparatively low cost of GPUs and the good scaling of the implementation gives a strong economical argument to prefer GPUs over CPUs when setting up a computer cluster for applications like molecular dynamics.

Core routines of the GPU-aided implementation (i.e. force calculation, neighbour list creation and numerical integration) approved to scale very well over multiple GPUs (tested with up to four). MD simulations are often done in computer centres, where one process has to share resources with the programs of other users - concerning e.g. the memory bandwidth between CPU and main memory. However, Cuda programs can explicitly lock one graphics

card for their own execution and are therefore not influenced by other processes.

6.2 Drawbacks

The implementation also showed up some drawbacks of the presented approach.

In practise, the different memory spaces on the GPU made programming more complicated, since additionally to the different access modes (e.g. read only) and concerns of parallel programming (e.g. write conflicts), the access pattern (e.g. non-/coalesced), the size, caching and especially the latencies have to be considered while writing the program. It is therefore tempting to end in sub-optimal solutions.

Since the Cuda device code does not support C++, data stored inside an object had to be copied into some C structure or variable in order to make it accessible by a Cuda kernel.

For short range forces, the GPU memory amount needed for the neighbour lists can limit the cutoff radius and the number of particles, which can be processed on one GPU.

For long range forces, the PPPM algorithm turned out to need so much communication time, that the overall program is faster, if every involved GPU does the FFT alone again. The time spent on actual computation could be shrinked to a level, where the PPPM execution time would have been dominated by communication. However, this implied a very bad scaling of the PPPM method over multiple GPUs.

The goal to keep the modularity and flexibility of Lammps was very important - though, this objective causes some alleviation in performance. Most clear on the example of the numerical NVE integration, a Cuda kernel had to wait longer for one single memory fetch from global memory than actually doing the entire computation. Whenever many different operations were performed on the same data (e.g. computing the mean square displacement, computing the virial and doing an NVT integration), the modular design implied the execution of different independent Cuda kernels, where every of them had to deal with the memory latencies again. It is possible to perform different often needed operation in one kernel, but this violates the modular structure of Lammps. A program specifically compiled for one problem may achieve better speed-ups.

6.3 Retrospection

Since known software design patterns (like *divide and conquer*) appear hard to be implemented on parallel architectures, in that case a general recommendation seems to be a *data-driven design* for three reasons:

- In order to avoid write conflicts, it turned out to be a good idea first to look at the required result data and parallelize the algorithm in way that every thread computes one independent memory element of the result. (This is even required, in case *atomic functions* can not be used.)
- While the performance of algorithms is usually judged by complexity classes (e.g. $\mathcal{O}(\ n\ \log(n)\ )$) referring to the number of CPU clock cycles, the number of memory accesses emerges to be similarly important (at least on the GPU).
- When multiple parallely executed threads want to exchange data, the memory bandwidth can easily become a bottleneck.

6.4 Comparison with other GPU MD projects

LAMMPS is not the only MD program which has lately got Cuda support. By just giving keywords, this section will compare four different projects in terms of features and scalability. In a lack of credible and matchable benchmarks, the speedups are not compared.

- **ACEMD** is focused on stability, robustness and usability. It uses OpenCL and runs under Windows and Linux. ACEMD is not published under the GPL, but a basic version is avaliable for free.
  - *Purpose:* bio-molecular dynamics
  - *Full simulation done on the GPU:* yes
  - *Designed from scretch for GPU:* yes
  - *Scaling:* uses MPI based parallelization
  - *Features:* Amber forcefields, Particle mesh Ewald (PME), Rigid and harmonic hydrogen bonds, user-defined (non-bounded) pair-forces can be loaded from a file with a look-up table
- **GROMACS** has GPU support in beta status using a library called OpenMM.
  - *Purpose:* primarily for simulation of biochemical molecules like proteins, lipids and nucleic acids
  - *Full simulation done on the GPU:* no
  - *Designed from scretch for GPU:* no
  - *Scaling:* Multiple GPU cards are not supported.
  - *Features:* basic force fields, Particle-Mesh-Ewald (PME), integration, constraints and implicit solvent Generalized Born methods
- **HOOMD** shares the idea to have a full simulation done on the GPU for better speedups, but has currently less fetures than Lammps.
  - *Purpose:* general purpose
  - *Full simulation done on the GPU:* yes
  - *Designed from scretch for GPU:* yes
  - *Scaling:* can use multiple GPUs, but only in one PC
  - *Features:* basic bounded and non-bounded pair forces (8 pair, 2 bond, 2 angle, 1 dihedral, 1 improper, 1 wall); NVT, NVE, NPT integrators
- **NAMD** is not focused on GPU support, but can transparently enable some basic GPU acceleration by a compile-time switch.
  - *Purpose:* high-performance simulation of large biomolecular systems
  - *Full simulation done on the GPU:* no
  - *Designed from scretch for GPU:* no
  - *Scaling:* can use multiple GPUs, recommended to have them in one PC only
  - *Features:* only uses the GPU for nonbonded force evaluation and energy evaluation

- **LAMMPS_CUDA** was introduced in this thesis and the features listed below represent the current (July 2010) state of development.
  - *Purpose:* general purpose
  - *Full simulation done on the GPU:* yes (overlaps bonded interactions on CPU with nonbonded on GPU)
  - *Designed from scretch for GPU:* no
  - *Scaling:* scales well up to several hundred GPUs (with efficiency 80% - depending on problem)
  - *Features:* non bonded force fields (20), bonded interactions (8 bonds, 8 angle, 6 dihedral), PPPM

The GPU acceleration of Lammps clearly has competitors in other MD codes, but the project seems to be at least on a par with them.

6.5 Outlook

Some parts of MD, like Lennard Jones force or the Verlet algorithm had been chosen in its form so that they can be calculated easily. However, on the GPU many calculations are memory-bounded and therefore future MD simulators with GPU-support could explicitly use higher-order numerical integrators or do not use approximations (like, e.g., the repulsive $\propto r^{-12}$ term) at almost no extra cost. For the same reason, more computational intensive forces show better speed-ups.

While writing this thesis, the work on the project went on and more features have been implemented, e.g. much more pair forces than just Lennard Jones and Coulomb were added. Currently, the project is designed as Lammps user package and we are looking forward to a planned publication of it in one of the next Lammps releases 2010.

For the future, other approaches are planned to be tested, e.g., to do the force calculation for *less* particles at the same time in order to make better use of the small GPU's MP cache, hoping to achieve better speed-ups. Furthermore, it may be worth to do asynchronous data transfers, i.e. to download data of already finished calculations while other threads are still active or uploading data and start threads on the first elements of it, while the remaining data is still copied. Until now, this was not considered high priority, since our approach generally minimises data transfer - it may, however, be useful when building the inter-process communication buffer.

Finally, we are also looking forward to Nvidia's Fermi architecture, especially since it has full C++ support for device code (probably making the integration into Lammps easier). Just to mention some more advantages, the Fermi graphics cards are designed for double precision and each multi-processor possesses a full Level-Two-Cache, which could eliminate some of the above mentioned problems caused by modularity and independent kernels.

Chapter 7

Appendix

7.1 Contribution list

Module	**Main Contributor**	
	Christian Müller	Lars Winterfeld
infrastructure		
compiler issues / Makefiles		✓
data handling		✓
communication	✓	
precision	✓	
neighborlists		✓
cell list neighbor list / “binning”		✓
pair forces		
lj/cut/cuda		✓
lj/cut/coul/long/cuda	✓	
buck/coul/long/cuda	✓	
pppm/cuda	✓	
fixes		
nve/cuda		✓
nvt/cuda		✓
enforce2d/cuda	✓	
temp/berendsen/cuda	✓	
temp/rescale/cuda	✓	
addforce/cuda	✓	
setforce/cuda	✓	
aveforce/cuda	✓	
computes		
temp/cuda	✓	
pressure/cuda	✓	
pe/cuda	✓	
atom styles		
atomic/cuda	✓	
charge/cuda	✓	

7.2 Workstations

The benckmark of this work were done on the workstations listed below with its technical specifications.

7.2.1 Workstations A

- **Graphics cards:** Two Nvidia® GeForce™ GTX 280.
  - **Chipset:** GT200 (core clock rate: 602 MHz. shader clock rate: 1296 MHz, 240 stream processors).
  - **Memory:** (1 GiB GDDR3, clock rate: 1107 MHz, bandwidth: 141.7 GiB/s, interface: 512 bit).
  - **Compute capability:** 1.3
  - **Other:** 65 nm fab. 1.4 billion transistors.
- **CPU:** Intel® Core™2 Quad Q9550 at 2.83GHz.
- **RAM:** 8 GiB DDR2 at 800 MHz.
- **Mainboard:** EVGA 780i Mainboard 3xPCIe2.0 16x.

7.2.2 Workstations B

- **Graphics cards:** Four Nvidia® Tesla™ C1060.
  - **Chipset:** GT200 (core clock rate: 602 MHz. shader clock rate: 1300 MHz, 240 stream processors).
  - **Memory:** (4 GiB GDDR3, clock rate: 1600 MHz, bandwidth: 102.4 GiB/s, interface: 512 bit).
  - **Compute capability:** 1.3
- **CPU:** 2 x Intel X5550 at 2.66 GHz.
- **RAM:** 48GiB DDR3 at 1066 MHz.
- **Mainboard:** Supermicro X8DTG-QF R 1.0a 4xPCIe2.0 16x.

7.2.3 Workstations C

- **Graphics cards:** One Nvidia® GeForce™ GT240M.
  - **Chipset:** GT216 (core clock rate: 550 MHz. shader clock rate: 1210 MHz, 240 stream processors).
  - **Memory:** (1 GiB GDDR3, clock rate: 1600 MHz, bandwidth: 25.6 GiB/s, interface: 128 bit).
  - **Compute capability:** 1.2
  - **Other:** 40 nm fab.
- **CPU:** Intel® Core™2 Duo CPU P7550 at 2.26 GHz.
- **RAM:** 4 GiB.
- **FSB:** 1066 MHz.

7.3 Algorithm side notes

7.3.1 Flatting a multi-dimensional array

In many situations, a multi-dimensional array needs to be converted into a flat one-dimensional memory array - in the context of this work, e.g., the forces (which are stored in a two-dimensional array indexed by particle-id and space-dimension) were needed as one single memory block for the transfer from and to the graphics card. A simliar problem is the mapping between multi-dimensional thread block indices and the one-dimensional memory strutrue of, e.g., particle types. In that case, a d-dimensional array can be strored as a sequence of all $(d-1)$-dimensional arrays, while 1-dimensional arrays can be stored directly. To put it as a formula, the flat index $f = f_{d-1}(\vec{i})$ of a element with indices $\vec{i} \in \mathbb{N}^d$ in a d-dimensional array with sizes $\vec{n} \in \mathbb{N}^d$ can be calculated by:

$$f_k(\vec{i}) = \begin{cases} i_0, & \text{for } k = 0 \\ i_k + n_k \, f_k(\vec{i}), & \text{for } k > 0 \, . \end{cases} \tag{7.1}$$

For instance, the element `a[3][5][7]` of an 3D array declared by `int a[10][20][30];` would result in element `b[7 + 30 * (5 + 20 * 3)]` of the flat array `int b[10*20*30];`.

7.3.2 Cell list indexing without modulo operations

The following code lists a short algorithm, which will produce a set of difference vectors $\vec{D} \in \mathbb{Z}^3$, which are needed by the cell list approach (section 4.2.2) for its update pattern. The number of neighbour shells to be considered $\vec{s} \in \mathbb{N}^3$ in each dimension is token as parameter - e.g. if only the closest neighbor cells should be considered, then $\vec{s} = (1, 1, 1)$. The main idea is to keep updating only in a "forward direction" and a simple implementation can waive integer division and modulo operations (which are ten times slower then additions on GPUs).

```
0:  void which_cells(int s[3]) {
1:    int c = ( (2*s[0]+1) * (2*s[1]+1) * (2*s[2]+1) - 1) / 2;
2:    int D[3] = {0, 0, 0};
3:    for(int i = 0; i < c; i++) {
4:      if(D[2] != s[2]) ++D[2];
5:      else {
6:        D[2] = - s[2];
7:        if(D[1] != s[1]) ++D[1];
8:        else {
9:          D[1] = - s[1];
10:         ++D[0];
11:       }
12:     }
13:     // process neighbour cell with relative position D
14:   }
15: }
```

7.4 Bandwidth and latency benchmark

The approximate values of bandwidth and latency for data transfers from and to a Cuda graphics card were important in this work. To meassure it, two different methods are applied:

1. Upload and download different data blocks from $m = 10$ KiB to 100 MiB in size, meassuring the transfer time t and by linear regression fitting it to $t(m) = \text{latency}_a + \frac{m}{\text{bandwidth}}$.
2. Upload and download 100 MiB as a whole and afterwards every byte individually. From the difference of these times, the latency_b can be calculated.

The following tables lists some results of these two methods on different workstations.

	upload			download		
Workstation	latency_a	latency_b	bandwidth	latency_a	latency_b	bandwidth
Workstation A (7.2.1)	181 μs	92 μs	2.07 GiB/s	204 μs	103 μs	2.07 GiB/s
Workstation C (7.2.3)	384 μs	25 μs	2.21 GiB/s	414 μs	34 μs	1.54 GiB/s

Transfers with data less than 1 MiB have lower latencies, but also lower bandwidth.

7.5 Radial distribution function

As there are different conventions in literature, the defintion for the radial distribution function (RDF) used in section 5.2 is given and motivated below. For this purpose, the formulas can be simplified to the use of one atom type only.

At first, the particle density $\rho(\vec{r})$ can be modeled from all known positions $\vec{r}_i$ of the atoms, in the simplest case as a sum of delta-like peak functions:

$$\rho(\vec{r}) := \sum_{i=0}^{n-1} \delta(\vec{r} - \vec{r}_i) \; . \tag{7.2}$$

This density is centered on every particle and averaged.

$$\overline{\rho}(\vec{r}) := \frac{1}{n} \sum_{j=0}^{n-1} \rho(\vec{r} - \vec{r}_j) \tag{7.3}$$

The RDF $g(r)$ should correspond to the variations of this density. Intuitivly, the number of particles of a hollow sphere from radius r to $r + \Delta r$ is devided by its volume and $g(r)$ is the limit of this fraction for $\Delta r \to 0$.

$$\begin{aligned} g(r) &= \lim_{\Delta r \to 0} \frac{\int_r^{r+\Delta r} \overline{\rho}(\tilde{\vec{r}})\, d^3\tilde{r}}{\frac{4\pi}{3}\left[(r+\Delta r)^3 - r^3)\right]} \\ &= \frac{\lim_{\Delta r \to 0} \frac{1}{\Delta r} \int_r^{r+\Delta r} \int_0^{\pi} \int_0^{2\pi} \overline{\rho}(\tilde{\vec{r}})\, \tilde{r}^2 \sin\vartheta \, d\varphi \, d\vartheta \, d\tilde{r}}{\lim_{\Delta r \to 0} \frac{1}{\Delta r} \int_r^{r+\Delta r} 4\pi\tilde{r}^2 \, d\tilde{r}} \\ g(r) &:= \frac{\int_0^{\pi} \int_0^{2\pi} \overline{\rho}(\vec{r})\, r^2 \sin\vartheta \, d\varphi \, d\vartheta}{4\pi r^2} \end{aligned} \tag{7.4}$$

7.6 Bibliography

[1] D. C. Rapaport. *The Art of Molecular Dynamics Simulation*, volume 2, chapter Introduction. Cambridge University Press, 2004.

[2] Christian Müller and Lars Winterfeld. https:// sourceforge.net / projects / lammpscuda.

[3] J. M. Haile. *Molecular Dynamics Simulation - Elementary Methods*, chapter Phenomenology. John Wiley & Sons, Inc., 1992.

[4] D. C. Rapaport. *The Art of Molecular Dynamics Simulation*, volume 2, chapter 13.2 Ewald method. Cambridge University Press, 2004.

[5] Markus Deserno and Christian Holm. How to mesh up ewald sums. ii. an accurate error estimate for the particle-particle-particle-mesh algorithm. *Journal of Chemical Physics*, 109(18), November 1998.

[6] Tom Darden, Darrin York, and Lee Pedersen. Particle mesh ewald: An n log(n) method for ewald sums in large systems. *The Journal of Chemical Physics 98*, (10089), 1993.

[7] Thomas Peter Gössi. *Computing Platforms for Parallel Molecular Dynamics*. PhD thesis, ETH Zurich, September 2000. 1.2.3 Long Range Interactions.

[8] J. M. Haile. *Molecular Dynamics Simulation - Elementary Methods*, chapter 4.4.1 Verlet's Algorithm. John Wiley & Sons, Inc., 1992.

[9] Thomas Peter Gössi. *Computing Platforms for Parallel Molecular Dynamics*. PhD thesis, ETH Zurich, September 2000. 1.2.4.2 Periodic Boundary Conditions.

[10] Steve J. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of Computational Physics*, 117(1-19), March 1995. http:// lammps.sandia.gov /.

[11] Michael Griebel, Stephan Knapek, and Gerhard Zumbusch. *Numerical Simulation in Molecular Dynamics*, chapter 4.2 Domain Decomposition as Parallelization Strategy for the Linked Cell Method. Springer-Verlag Berlin Heidelberg, April 2007.

[12] Harry F. Jordan and Gita Alaghband. *Fundamentals of Parallel Processing*, chapter 2.7.2 A Simple Performance Model - Amdahls's Law. Pearson Education, Inc., 2003.

[13] NVIDIA. *CUDA Programming Guide*, 2.3.1 edition, August 2009. http:// developer.download.nvidia.com / compute / cuda / 2_3 / toolkit / docs / NVIDIA_CUDA_Programming_Guide_2.3.pdf.