LAMMPS$_{CUDA}$ - a new GPU accelerated Molecular Dynamics Simulations Package and its Application to Ion-Conducting Glasses

Dissertation zur Erlangung
des akademischen Grades
Doctor rerum naturalium (Dr. rer. nat.)

vorgelegt von

Christian Robert Trott

Fachgebiet Theoretische Physik II
Institut für Physik
Fakultät für Mathematik und Naturwissenschaften
Technische Universität Ilmenau

Betreuer: Prof. Dr. Philipp Maaß
Gutachter: Prof. Dr. Erich Runge
Gutachter: Prof. Dr. Hans Babovsky

Tag der Einreichung: 19. Juli 2011
Tag der Verteidigung: 16. November 2011

urn:nbn:de:gbv:ilm1-2011000472

Abstract

Today, computer simulations form an integral part of many research and development efforts. The scope of what can be modeled has increased dramatically, as computing performance improved over the last two decades. But with serial-execution performance of CPUs leveling off, future performance increases for computational physics, material design, and biology must come from higher parallelization. In particular, heterogeneous hardware designs are becoming increasingly important due to their, compared to conventional hardware, inherently larger power efficiency. The currently leading technology are GPU accelerated systems. Offering a five- to tenfold increase of cost- and power-efficency, GPU based supercomputers have proliferated rapidly since their introduction in 2008. Now, three of the five world's fastest supercomputers achieve their performance through GPUs. In order to take full advantage of such systems, new programming paradigms have to be applied.

This thesis presents a new code for Molecular Dynamics (MD) simulations. LAMMPS$_{CUDA}$ employs GPUs to accelerate simulations of many material classes, including bio-simulations, semiconductors, metals, inorganic glasses, polymer systems, and atomic liquids. Here, details of the implementation, as well as comprehensive performance evaluations will be presented, showing that LAMMPS$_{CUDA}$ can efficiently utilize single workstations and large clusters with hundreds of GPUs.

In the second part of this thesis, LAMMPS$_{CUDA}$ will be used to develop a new MD force field, called BC-U, for ion-conducting glasses of the general composition $x\mathrm{Li_2O} - y\mathrm{B_2O_3} - z\mathrm{Si_2O_4} - w\mathrm{P_2O_5}$. It will be shown that the new BC-U force-field reproduces composition-dependent, experimental data of structural and ion-dynamics properties for several binary glasses better than established potentials. For this demonstration, extensive simulations of lithium silicate, lithium borate, and lithium phosphate are carried out using several MD force fields. Numerous properties, including coordination numbers, thermal expansion coefficients and conductivity activation energies are determined for a wide range of compositions, and compared to experimental data. On a newly-introduced evaluation scale, the BC-U potential consistently has the highest overall score of all investigated force fields.

Moreover, in simulations of lithium borophosphate using the BC-U potential, the mixed network former effect in the activation energies is observed, in agreement with experimental findings. Because it does not occur when simulating borosilicates and phosphosilicates, MD simulations with the BC-U potential can be used to verify proposed microscopical causes of the mixed network former effect.

Summarizing, the new BC-U potential is suited to investigate structural properties as well as features of the ion-dynamics, and is the primary choice for future investigations of ion-conducting glasses.

Abstract

Computersimulationen sind heute ein fundamentaler Bestandteil von Forschung und Entwicklung. In den letzten Dekaden wurden die Grenzen dessen, was modelliert werden kann, durch die kontinuierliche Steigerung der verfügbaren Rechenleistung dramatisch erweitert. Aufgrund der stagnierenden seriellen Rechenleistung heutiger Prozessoren müssen zukünftige Leistungsgewinne mittels Parallelisierung erreicht werden. Insbesondere heterogene Hardware-Architekturen werden immer wichtiger, da sie inhärente Vorteile im Bereich der Energieeffizienz aufweisen. Zur Zeit wird dieser Bereich durch von Grafikkarten beschleunigte Systeme dominiert. Aufgrund ihres fünf- bis zehnfachen Energie- und Kosteneffizienz haben auf Grafikkarten basierte Supercomputer seit ihrer Einführung im Jahr 2008 schnelle Verbreitung gefunden. Mittlerweile beziehen drei der fünf schnellsten Computer der Welt einen Großteil ihrer Rechenleistung aus Grafikkarten. Um derartige Systeme effizient zu nutzen sind jedoch neue Programmierungsansätze notwendig.

In dieser Arbeit wird ein neuer Molekular-Dynamik-Code namens LAMMPS$_{CUDA}$ vorgestellt. LAMMPS$_{CUDA}$ verwendet Grafikkarten um Simulationen verschiedenster Materialklassen zu beschleunigen. Dazu gehören biologische Systeme, Halbleiter, Metalle, inorganische Gläser und Polymersysteme. Es werden sowohl Details der Implementierung vorgestellt als auch umfangreiche Leistungsuntersuchungen präsentiert. Diese zeigen, dass LAMMPS$_{CUDA}$ sowohl auf einzelnen PCs als auch auf Großrechnern mit hunderten Grafikkarten effizient laufen kann.

Im zweiten Teil der Arbeit wird LAMMPS$_{CUDA}$ verwendet, um ein neues Potential für ionenleitende Gläser der Zusammensetzung $x\mathrm{Li_2O} - y\mathrm{B_2O_3} - z\mathrm{Si_2O_4} - w\mathrm{P_2O_5}$ zu entwickeln. Wie sich zeigt, kann das neue Potential – das mit BC-U bezeichnet wird – nicht nur strukturelle sondern auch dynamische experimentelle Daten verschiedener binärer Gläser über den gesamten Kompositionsbereich sehr gut reproduzieren. Dafür werden umfangreiche Simulationen von Lithiumsilikaten, Lithiumboraten und Lithiumphosphaten durchgeführt und verschiedene Eigenschaften wie Koordinationszahlen, Ausdehnungskoeffizienten und Aktivierungsenergien bestimmt. Zum Vergleich wurden nicht nur experimentelle Befunde herangezogen, sondern auch Simulationen mit etablierten Potentialen durchgeführt. Im Ergebnis erreicht das neue BC-U-Potential für alle Systeme die höchste Punktzahl aller hier untersuchten Potentiale auf einer neu eingeführten Bewertungsskala.

Zudem tritt in Simulationen von Borophosphaten der sogenannte Mischnetzwerkformer-Effekt auf. Damit ist das BC-U Potential prädestiniert dafür, zur Verifizierung von Theorien der mikroskopischen Ursachen dieses Effektes eingesetzt zu werden.

Zusammenfassend lässt sich festhalten, dass das BC-U-Potential sowohl für Untersuchung von strukturellen als auch dynamischen Eigenschaften geeignet ist. Da es die meisten experimentellen Befunde besser reproduziert als die hier zusätzlich untersuchten etablierten Potentiale, ist es die erste Wahl für zukünftige Molekular-Dynamik-Studien an ionenleitenden Gläsern.

Contents

1 Introduction

During the last three decades, extensive numerical calculations have become the third pillar of modern research and development, next to the actual experiment and analytics. From the design of a new airplane [1] to the development of the latest cancer medicine [2], an ever increasing array of products is existing as virtual objects long before the first prototype is produced. Simulations are used to predict the future, be it the weather [3], the traffic in a metropolitan area [4] or the stock market [5]. And much of the modern architecture would never have come into existence, if it were not for the reliable and fast programs which are used to do the structural analysis behind the idea of an architect [6].

One class of simulations which is of great importance for material science and biology is the class of Molecular Dynamics (MD) simulations. They are used to model materials at a particle (i.e. atoms or molecules) level by calculating the motion of all particles based on their interactions with each other. Generally one can distinguish between classical MD and Quantum MD simulations. In the latter case quantum mechanical calculations are used to determine the forces on each particle, while in classical MD simulations effective force fields are used. Typical system sizes vary between 10^4 and 10^9 particles. The largest systems only run for a fraction of a nanosecond of simulated time, while the smallest systems can run for up to a millisecond. Although these numbers are already impressive compared to what was possible ten years ago, they fall well short of allowing scientists to model processes in macroscopic samples. For example in order to simulate the interaction of a virus with the surface of a cell, the modeling of billions of atoms for many milliseconds is required. The same is true for the modeling of crack propagation in materials. And even studying the influence of defects in crystals (i.e. doping in semiconductors) requires relatively large samples if the defect rate is small.

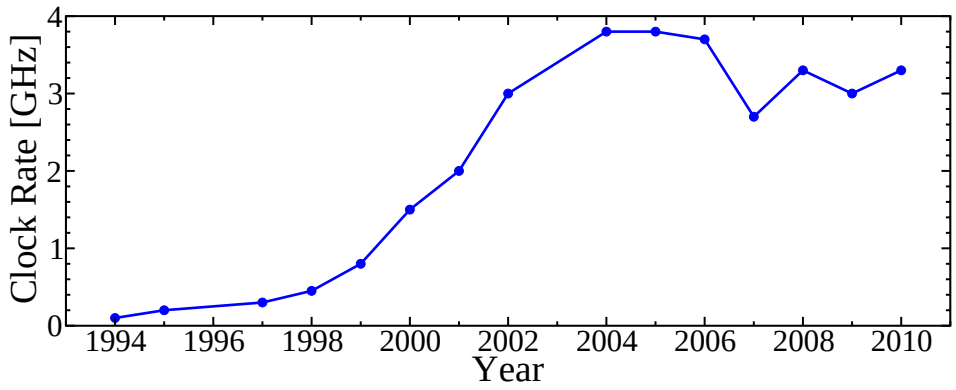


Figure 1: Historical clock rates of Intel CPUs [7].

Between 1990 and 2004, the limit of what was possible with MD simulations simply grew with the increasing clock rates of the Central Processing Units (CPUs) (see Fig. 1).

Without developing a new code, the available computational power on a CPU doubled about every two years. But by the middle of the last decade, this path did not go any further. The Intel Pentium 4, clocked at 3.8 GHz, was the chip with the highest frequency ever built for the mass market. Power leakage and corresponding heat issues prevented faster processor cores. So in order to increase the available computational power, chip designs moved towards multi-core implementations, with which several instructions can be performed simultaneously. At the same time high performance computers started to be dominated by clusters - a large number of more or less conventional servers based on PC technology connected via a fast network. Given a suitable program those clusters allow the usage of hundreds or even thousands of processors to solve a numerical problem. While the necessary parallelism is not given in every case, many scientific problems can be split up in independent operations.

Graphic processing units (GPUs), typically used to run ever more realistic visuals of computer games, are small clusters in themselves. Featuring dozens or even thousands of processing cores, they are designed to execute the same set of instructions on a large data array. This is in stark contrast to the common processor, which typically has a very diverse load of instructions at any given time. In a modern operating system dozens or even hundreds of programs run at the same time, sharing the resources of the processor. GPUs on the other hand have to perform the same operation (i.e. putting an overlay effect on a pixel) a thousand times before proceeding to the next operation. This allows them to focus much more on actual computations than on data management and instruction order optimization. Hence about three quarters of the transistors of a GPU are dedicated to computations compared to only about one quarter in normal CPUs. As a consequence, GPUs can be much more powerful and energy efficient than CPUs, with the peak performance of a modern GPU being 30 times as high as that of a modern CPU.

Being aware of the huge computational power of GPUs, scientists started to explore the possibility of using them for serious computations [8, 9]. By mapping mathematical algorithms on graphic operations, such as adding a light effect to a texture, they were able to use the graphics engine for simulations. Though the results were promising, it was never widely adopted due to the complex but limited programming model. In 2007, NVIDIA, the largest producer of dedicated GPUs, changed that situation dramatically. By providing the CUDA toolkit, they allowed for a programming of their GPUs with what amounts to simple C code with some GPU specific extensions. Over time NVIDIA added development tools to the toolkit, such as debuggers, memory check programs, and profiling tools, which were key to making CUDA usable for a wide array of developers. NVIDIA introduced also a new line of GPU products: TESLA cards are specifically designed for data crunching - the second installment, the C1060, not even had a display connector - and provide features such as Error Correction Code (ECC) memory, which are not needed in classical GPUs.

Combining GPUs with the successful idea of clusters allows for the deployment of supercomputers with a much higher energy and cost efficency than before. This lead to a rapid deployment of large installations. Now, only 4 years after the introduction of the

CUDA ecosystem, 3 of the 5 most powerful supercomputers in the world are featuring NVIDIA Tesla GPUs. The largest one, Tianhe-1A, employs 7.168 NVIDIA M2050 GPUs, which combined with its 14336 Hexcore Intel CPUs provide a peak performance of more than 2.5 PFlop/s [10].

When designing a software for such systems, multiple problems have to be solved. First there are at least two layers of parallelization: the compute cores on the GPU and the level of using multiple compute nodes. On top of that, a single GPU already consists of a number of so-called multiprocessors each of which has a number of compute cores (or shaders). Furthermore a compute node typically has multiple GPUs and CPUs. Finding an optimal partitioning of a computational task into these parallelization layers therefore is a first difficulty. Beside the algorithmical problems posed by these different layers, which all come with differing bandwidths to their respective memories, they require different programming languages or libraries. So the GPU code is written with the CUDA model, while parallelization on one node is most efficiently done with a threading model like OpenMP or pthreads. Inter-node communication on the other hand is almost always handled by the Message Passing Interface (MPI) set of routines.

With LAMMPS$_{CUDA}$, the MD package which will be presented in the first half of this thesis, many of those challenges are met. It will be shown that LAMMPS$_{CUDA}$ not only brings cluster level performance to single workstations, but that it is well suited for using next generation GPU clusters. Designed as an general purpose MD code, it supports the whole range of material classes which are currently simulated with MD codes. This includes among others atomic liquids, bio-molecules, polymers, granular systems, semiconductors and metals. For all of these material classes typical node performance increases are in the range of a factor of 5 to 10.

The code is now part of the main LAMMPS program as the so-called "USER-CUDA" package [11], and can be downloaded from `http://lammps.sandia.gov`. Being one of the most used MD codes besides Amber, NAMD and Gromacs, LAMMPS is a vital tool for hundreds of scientific groups around the world[1].

In the second half of this thesis, LAMMPS$_{CUDA}$ will be used to parameterize a new force field for ion-conducting glasses of the general composition:

$$x\mathrm{Li_2O} - y\mathrm{B_2O_3} - z\mathrm{Si_2O_4} - w\mathrm{P_2O_5}.$$

Inorganic glasses such as this are of great interest for a wide range of applications including solid electrolytes for batteries, nuclear waste storage, anodic bonding material in semiconductor devices, and use as bioactive material for artificial bone replacements [12, 13, 14, 15]. Almost all of these applications rely on more or less complex mixed glass systems with multiple components. In contrast, basic research - especially theoretical research - has focused on simple glass systems. The great majority of all molecular dynamic simulations of ion-conducting glasses has been performed for lithium and sodium silicate. One reason for this limitation is that systems as simple as possible have to be investigated, in order to isolate a feature from additional influences. Only then, it is possible to establish causal relations.

[1] According to Web of Science, LAMMPS has been cited nearly 400 times in 2010.

There are phenomena though, e.g. the mixed network former effect and the mixed alkali effect in ion-conducting glasses, which only occur in the more complex compositions. It can also be imagined that with sufficiently accurate simulations of mixed systems, it would be possible to first determine candidate compositions for a special purpose with simulations and then synthesize only these few compositions, instead of systematically producing samples for all possible variations of ingredients. The key word here is "accurate".

In the context of MD simulation the "accuracy" or reliability of the simulation largely depends on the quality of the interaction model. Developing and verifying better force fields is one of the key challenges. Most of the potentials which are currently in use for network glasses are based on simple quantum mechanical calculations or were fitted to properties of crystals with the same composition. But these approaches are hard to use when dealing with complex compositions.

Therefore a different method is used here. Instead of trying to find parameters directly for one of these complex systems, a gradual extension of an existing force field is employed. By starting with a reliable force field of a simple system and adding new atom species one by one, only few parameters have to be determined in each step. These parameters can be determined by a brute force method. For each candidate set of parameters a number of test simulations is performed, and the properties of the resulting glass are compared to experimental data.

Using about 30 GPUs LAMMPS$_{CUDA}$ not only enables scanning of a large parameter space for optimal values, but it also allows us to perform a comprehensive characterization of the new force field and several benchmark potentials.

In total 50 different compositions of the three binary lithium glasses $x$$Li_2O$ - B_2O_3, $x$$Li_2O$ - Si_2O_4, and $x$$Li_2O$ - P_2O_5 as well as the three mixed network former glasses 0.4Li_2O - $y$$B_2O_3$ - $(1-y)$$Si_2O_4$, 0.4$Li_2O$ - $y$$B_2O_3$ - $(1-y)$$P_2O_5$, and 0.4$Li_2O$ - $y$$Si_2O_4$ - $(1-y)$$P_2O_5$, will be investigated. Calculated properties such as density, coordination numbers, and activation energies of these glass systems are then compared to experimental data if available. Also a quantitative assessment scheme is introduced, which enables the assignment of a quality number to each force field. This allows for a summarizing of a force field evaluation in an efficient way, and provides a general assessment of the force fields overall quality.

In total more than ten billion simulation steps with systems of 8,000 particles or more have been performed for the present work. This is about two to three orders of magnitude more than what was typically done in previous studies of single materials, and is only possible since LAMMPS$_{CUDA}$ accelerates this particular kind of simulation by as much as a factor of 15 comparing a single GPU to a modern conventional workstation.

Much of this work would not have been possible without the people I have been collaborating with. In particular two people have contributed directly to this thesis.

Lars Winterfeld was strongly involved into the early development process of LAMMPS$_{CUDA}$. Much of the original data management classes were developed by him. He was also instrumental in evaluating the possibility of using a cell list based approach to pair interaction calculation, which ultimately turned out to be inferior to

the neighborlist approach now employed in LAMMPS$_{CUDA}$ Further information can be found in his bachelor thesis [16].

Martin Körner was an invaluable help in performing the thousands of MD runs of the ion-conducting glasses. After setting up the input scripts he took over from me and managed to keep our compute resources busy.

Parts of sections 2.2, 2.3.2, and 2.4 were taken from a manuscript, which was submitted to Computer Physics Communications and is currently under review [17].

2 LAMMPS$_{CUDA}$ - a new general purpose Molecular Dynamics Package for GPU clusters

LAMMPS$_{CUDA}$ is implemented as an extension to the widely used MD code LAMMPS [11]. With its 26 different force fields, LAMMPS$_{CUDA}$ can model atomic, polymeric, biological, metallic, granular, and coarse-grained systems up to 20 times faster than a modern quad-core workstation by harnessing the power of recent GPUs. At the same time it offers unprecedented multi-GPU support for an MD code. By providing very effective scaling of simulations on up to hundreds of GPUs, LAMMPS$_{CUDA}$ enables scientists to harness the full power of the world's most advanced supercomputers, such as the world's (currently) second fastest supercomputer, the Tianhe-1A at the Chinese National Supercomputing Center in Tianji [10].

While the host code is written using C++ the GPU methods are written using CUDA.

LAMMPS was chosen as a basis for this project mainly for three reasons. Its biggest advantage is that it is meant to be easily extentable. Indeed not even a single line of the existing code needs to be changed in order to add a new pair force field, or some analysis routine. As a consequence, it is realtively easy to add GPU support for certain features one after another without breaking the whole code.

Additionally LAMMPS already runs exceptionally well on large clusters. Using the CPU data partition and communication algorithms of LAMMPS the GPU version is automatically a multi GPU code. Improving the communication routines later by adapting them to the special situation of GPU clusters is much more efficient than writing these parts from scratch.

Last but not least LAMMPS has a sophisticated input script parser, which allows for very complex simulations without the need to write extra code for every new investigation. This makes LAMMPS one of the most flexible tools for particle simulations available.

In 2010 the source code of LAMMPS exceeded 200,000 lines of C++ code. With the modifications of LAMMPS$_{CUDA}$ another 50,000 lines of C++ and CUDA code were added.

2.1 A short repetition of the CUDA programming model

While it is beyond the scope of this work to give an exhaustive description of CUDA programming, a short introduction shall be provided. The aim of this introduction is to give a reader unfamiliar with CUDA a grasp of the basic concepts, which are necessary to understand this thesis. In order to avoid confusion the description here will be simplified in some points, with more detailed explanations given in footnotes.

More comprehensive tutorials for CUDA are readily available on the internet, with NVIDIA providing a good compilation of some of the best guides on its website at `http://developer.nvidia.com/cuda-training` including multiple recorded university lectures. Especially the "Dr. Dobbs" article series can be recommended. Also the docu-

mentation of NVIDIA itself is well worth reading. The "CUDA programming guide" is very detailed, and a "Best practise" document describes the most important optimization strategies.

2.1.1 Terms and definitions

In this section terms which are used in the GPU programming community are defined. It is assumed that the reader is familiar with the meaning of basic hardware component names such as CPU, RAM and cache.

Hardware-related items:

- **Host:** Refers to the conventional hardware base of motherboard, RAM and CPU. Running code on the host means that it is executed by the CPU.
- **Device:** Refers to the GPU.
- **Multiprocessor:** A GPU consists of several multiprocessors each of which has a number of cores. Each multiprocessor has its own register file as well as its own shared memory.
- **Core/Shader:** The actual processing units of a GPU which execute operations. Several cores form a multiprocessor. During one cycle, all cores of a multiprocessor perform the same instruction.
- **Device memory:** The dedicated main memory of a GPU. It is shared by all multiprocessors. Typical sizes are in the range of 512 MB to 3 GB for consumer level GPUs and 2.5 GB to 6 GB for professional products.
- **Shared memory:** A fast read-write cache shared by all cores of one multiprocessor. Its size can be set to either 16 KB or 48 KB. Usage must be explicitly managed by the program.
- **Texture memory:** A read-only cache optimized for spatial ordered accesses. It caches up to 768 KB of the device memory. The working set per multiprocessor is 6 KB or 8 KB.
- **Constant memory:** A read-only cache. It caches up to 64 KB of the device memory and is common to all multiprocessors. The working set per multiprocessor is 8 KB. As the shared memory and texture memory it must be explicitely managed by the program.
- **Level 2 cache:** A read-write cache on Fermi level GPUs (see below). A program cannot explicitely access this cache, rather all accesses to the device memory go through this cache. Its cacheline size is 128 bytes.

- **Pinned memory:** Pinned memory refers to an address region of the RAM which is not masked by the virtual address space[2]. This enables direct memory access by hardware components such as the GPU and the network adapter (e.g. infiniband cards) without going through the CPU. As a consequence, data transfer between pinned memory and the GPU is up to two times faster than to memory regions which are not pinned. The amount of memory which can be pinned is limited and allocating too much of it reduces the operating system performance.

Software-related items:

- **Host function:** A function which is executed on the host.
- **Global function:** A function which is launched from an host function and executed on the GPU. A global function needs to have the identifier "`__global__`".
- **Device function:** A function which is called from a global or another device function and is executed on the GPU. A device function needs to have the identifier "`__device__`".
- **Kernel:** Refers to a global function and all the device function called by it. Upon kernel launch all its instructions are loaded in the instruction memory of each multiprocessor.
- **Thread:** In parallel programs a thread refers to one of the parallel executed instruction sets. In CUDA a thread is executed by one core.
- **Warp:** Currently 32 threads form a warp. The threads of a warp are executed at the same time. On pre-Fermi GPUs this takes four cycles, on Fermi Level GPUs one cycle.
- **Block:** Multiple threads form a block. Each block is executed by one multiprocessor. All threads of one block share a common shared memory. The threads of one block have a 3D index. In a program the number of threads per block must be explicitely given for each kernel launch. The maximal number of threads per block depends on the GPU and is 512 for Fermi GPUs. Blocks cannot generally synchronize with each other, since a block has to be finished before the next one can be executed by the multiprocessor.
- **Grid:** As the threads of a block, the blocks have a 3D index themselves. Currently the number of blocks is limited to 65536 in the x and y direction and 1 in the z direction. This arrangement is called a grid. The grid size has to be given in the program for each kernel launch.

[2] A virtual address space allows the operating system to move chunks of allocated memory within the actual physical RAM in order to avoid fragmentation, while a program uses the same pointer to that memory no matter where it actually resides.

- **Atomic functions:** These functions allow thread safe write operations on data. While one thread executes an atomic function on a data element, every other thread trying to modify the same data element is halted until the first thread is done.

2.1.2 The CUDA programming model

Programming GPUs requires a fundamentally different approach from writing efficient code for CPUs. The main reason is that GPUs have more in common with vector processors than with the CPUs found in desktop machines, servers and high performance computing clusters. A GPU is built up from a number (1-30) of small vector processors, often referred to as multiprocessors. Each multiprocessor has 8 to 48 compute cores, a dedicated shared memory as well as its own chunk of constant and texture cache[3]. The vector processor aspect comes into play because each multiprocessor has only one instruction unit[4], which in a given clock cycle feeds the same instruction to each of the compute cores.

Different multiprocessors are in principle independent from each other. But all of them access the same device memory through a common L2 cache. This basic layout of a Fermi generation NVIDIA GPU is depicted in Figure 2.

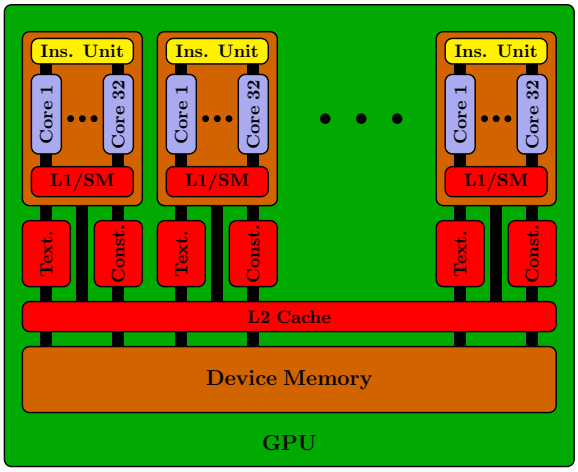


Figure 2: (Simplified) hardware layout of an NVIDIA Fermi GPU.

[3]Each multiprocessor also has its own register file, 16 load/store units for memory access, 4 special function units, and an instruction cache. The compute cores themself have a floating point and an integer operation units.

[4]Indeed the Fermi architecture has two instruction units, which can each issue an instruction of one warp to either a group of 16 cores, the 16 store/load units of an multiprocessor, or the four special function units. This implies that up to 5 warps can be actively worked on in a given clock cycle - three using the up to 48 normal compute cores, one doing memory access, and one using the special function units.

In the CUDA programming model the actual hardware specifications are hidden behind an abstraction layer. Instead of just running a number of homogenous threads, a hierarchical structure of threads is used, which can be mapped to the hardware. Multiple threads form one block, with one block being executed on a single multiprocessor of the GPU. This means that the threads of one block work synchronously (because there is just one instruction unit issuing the commands to the compute cores) and that they have access to a common shared memory. The blocksize - that is the number of threads in a block - is usually chosen to be much larger than the number of compute cores of a multiprocessor, with typical values lying between 128 and 256. For each kernel launch on the GPU the number of blocks and the blocksize has to be given. While it is often enough to have a simple 1D enumeration scheme, internally CUDA works with 3D grids of blocks and 3D layouts of threads within a block. As a consequence, each thread has 6 coordinates, which identify the block which it belongs too and the position within that block.

As long as the resources requested by each block (how much shared memory it is using, how many registers are needed) are few enough multiple blocks can be launched on a multiprocessor at the same time. This helps to optimally use the available hardware resources in order to get closer to the theoretical peak performance.

If insufficient resources are available to launch all blocks simultaneously, blocks are going to be worked on sequentially. So, only after a block has finished, the next one can be issued to a multiprocessor. As a consequence, global synchronization between blocks is not possible within a kernel, since some blocks might not even run yet[5]. Generally, it is a good practice to launch as many blocks and threads as possible in order to give the various schedulers enough flexibility to utilize all available resources in the most efficient way.

This abstraction layer has also numerous advantages in itself. E. g. a code does not need to take explicit care of the type of GPU it is running on. For the implementation it does not matter if the program will be run on a GPU with 30 multiprocessors with 8 cores each or on a GPU with 1 multiprocessor of 32 cores. It also simplifies the actual parallelization scheme. Instead of using every thread to work on multiple input data elements, one can just start one thread for each of the data elements. Despite this being easier to implement, it also serves the purpose of allowing for a better saturation of the GPU hardware.

The following code is a CUDA implementation of a reduction algorithm[6].

```
1 #include <stdio.h>
  #include <stdlib.h>
  #define N 4096

  __global__ void cuda_reduce(int* a,int* b)
```

[5] It is possible to implement global synchronization if the number of blocks is small enough, so that all blocks can run at the same time. This can be used to implement persistent thread schemes. Usually a strategy to do that is to use "lock-flags" in the device memory. Such an implementation, though, is very much dependent on the particular hardware and is prone to errors.

[6] One commonly refers to a reduction as the summation of the elements of a vector.

```
 6 {
     a+=blockIdx.x*blockDim.x;
     for(int i=2;i<=blockDim.x;i*=2)
     {
       if(threadIdx.x<blockDim.x/i)
11     a[threadIdx.x]+=a[threadIdx.x+blockDim.x/i];
       __syncthreads();
     }
     if(threadIdx.x==0) atomicAdd(b,a[0]);
   }
16
   int main (int argc, char *argv[])
   {
    int a[N];
    for(int i=0; i<N; i++) a[i]=1;
21
    int sum = 0;

    int* a_dev;
    int* b_dev;
26
    int nblocks=N/256;

    cudaSetDevice(4);
    cudaMalloc(&a_dev,sizeof(int)*N);
31  cudaMalloc(&b_dev,sizeof(int));
    cudaMemset(b_dev,0,sizeof(int));
    cudaMemcpy(a_dev,a,sizeof(int)*N,cudaMemcpyHostToDevice);

    cuda_reduce<<<nblocks,256>>>(a_dev,b_dev);
36
    cudaMemcpy(&sum,b_dev,sizeof(int),cudaMemcpyDeviceToHost);

    printf("\nTotal Sum %i\n",sum);
   }
```

The first function `cuda_reduce` is the actual kernel which is executed on the GPU. In it each block (identified by the index `blockIdx.x`) with `blockDim.x` threads will sum up its part of the array `a` and put the result into its corresponding element in `b`. Note the use of the `atomicAdd` function in line 13 in order to ensure that every block is able to add its value without interference of other blocks.

The `main` function first sets up the data on the host, activates the GPU afterwards and then copies the initial data to the device. In line 36 the kernel is called with a syntax resembling a templated function call. The parameters in brackets give the number of blocks and threads to be used. The kernel execution is actually asynchronous, so the CPU returns after submitting the kernel and continues until the next CUDA command. In this case, it happens to be the next line in the code, which copies the result back to the host. This general scheme of initializing data on the host, copying it to the device, doing some calculations on the GPU, and copying data back is found in almost every CUDA program.

2.1.3 Bottlenecks

Much of the development history of modern CPUs was spent on eliminating bottlenecks. Most of their logic is dedicated to large caches and the functionality needed to fill it in the most efficient way. CPUs are also able to do out-of-order executions - i. e. the processor can change the order in which instructions are executed (if their interdependency allows it) to facilitate increased utilization. As a consequence, the various compute units of a CPU can be kept busy even if the code is not optimally written. The cost for this benefit is that only a minority of the transistors of the processor are actually doing calculations.

GPUs are different in that most of their die space is dedicated to the actual compute cores. On the other hand they lack many of the advanced features of modern CPUs, making it much more likely to run into bottlenecks which limit the utilization of the compute cores. The most important class of bottlenecks is related to access to the various memory regions on the GPU.

The fundamental limit of global memory access is the memory bandwidth. On the most powerful GPUs this bandwidth is 150-200 GB/s. This means that up to 50 billion single precision floats can be loaded per second. Considering that the same GPUs can perform about 1 trillion floating point operations per second, this means that a code needs at least 20 operations per accessed float, in order not to be limited by the memory bandwidth.

The theoretical memory bandwidth can only be reached under certain conditions. The most important one is that access to the memory is coalesced. This means that consecutive threads of a warp need to access memory elements within the same 128 bytes segment of the main memory. The reason is that the memory controller needs to issue one fetch operation per 128 bytes segment which is accessed. If the threads access elements in different 128 bytes segments these memory fetch operations will be serialized. In the worst case (i.e. random memory access with each thread of a warp accessing a 4 byte float of a different 128 bytes segment) this means that the actual bandwidth is divided by 32. Given cases in which uncoalesced access cannot be avoided the usage of caches is often beneficial.

In addition to the bandwidth, latency limits the actual data throughput. Issuing a memory fetch operation is associated with a latency of about 400 cycles. Considering that a multiply or an addition takes only 4 cycles this means that up to 100 calculations could be performed in the time until one memory access is serviced. Fortunately a compute core has not to wait until a requested data element is actually provided to a thread. Instead it can work on other threads. This allows an effective masking of memory access operations as long as enough threads are available per compute core. Therefore one should generally launch at least one order of magnitude more threads than there are compute cores - for the top line Fermi cards this means at least 5000 threads.

Caches can be used to mitigate the effects of sub-optimal memory access. On Fermi GPUs there are five different caches: L1, L2, texture, constant cache and shared memory. All caches have in common that their effect on performance increases with increasing reusage of data. When a data element is only used by a single thread, caching offers no

performance benefit.

Since the introduction of the Fermi chip, all global memory accesses are cached. There is a global level 2 cache of up to 768 kB and a local level 1 cache of either 16 kB or 48 kB which is specific to each multiprocessor. Both caches are managed by the hardware and all global memory accesses go through them. So it is not possible to manually manage it in order to enhance data reuse and minimize cache flushes. The cacheline size (i.e. the size of an atomic memory transfer block) is 128 bytes. This means that for purely random access of 4 byte variables (such as floats and integer) only 1/32nd of the cache bandwidth is used.

In these cases texture cache can help, which is the method of choice to optimize random memory access on large data sets. It is a global 768 kB cache shared by all multiprocessors and can be used to cache data sets of arbitrary size. The texture cache is optimized for spatiality, that means that it is faster to access data elements which are close together on a 2D representation of an array, than accessing random elements. Its two most important differences from the level 2 cache are its cacheline size of 32 byte and the data, which shall be cached, needs to be specified.

The constant memory is most useful for accessing parameters which are used by many threads. If all threads of a warp access the same constant memory element and no cache miss occurs, it will be as fast as a register access. Access to different elements by threads of the same warp is serialized in as many elements as are requested. Even so, considering that a register access costs one cycle only, this is a very fast way to access data compared to the 400 cycles latency of the global memory, even if a lot of serialization occurs. Unfortunately, the constant cache is only 8 kB per multiprocessor. Therefore it cannot generally be used to cache data of particles. In LAMMPS$_{CUDA}$ the constant cache is most prominently used to cache potential parameters if the number of particle types allows to fit the whole parameter list into the constant cache. It is also used to store all relevant pointers to data, reducing the registers that are used by threads. Which data is cached must be defined a priori of a kernel launch - that is the data is loaded into the constant cache at kernel launch time.

The last memory regime is the shared cache. This manually managed cache, i.e. one has to explicitely transfer data to it, is local to each Multiprocessor and has a size of 16 kB or 48 kB. The size can be set during runtime for each kernel launch. This is because the shared memory is actually using the same 64 kB of memory banks as the level 1 cache. Depending on the kernel, it is either beneficial to have a larger level 1 cache or a larger shared memory. A possible performance hindrance using shared memory are bank conflicts. Shared memory is separated into 32 banks, where consecutive 32-bit words are in different banks. Each bank can serve only one access request per clock cycle. Therefore multiple requests for the same bank have to be serialized. As an example, if the thread i of a warp accesses element $a[i]$ of an array of integers the access can be served in one operation. If the threads access only every other element (thread i reads $a[2 * i]$) first threads 1 to 16 will get their data, and after that threads 17 to 32. The worst case would be if thread i accessed $a[32 * i]$. This requires 32 separate shared memory reads, therefore it would take also 32 times as long as a bank conflict-free access

pattern. An exception to the rule that two threads belonging to the same warp should not access the same bank, is if they access the same element, since broadcast operations are possible. When thread i reads $a[i\%4]$, no bank conflict occurs.

Another bottleneck in GPU programs is the data transfer between host and device through the PCIe bus. Assuming that the motherboard provides full 16 PCIe lanes for a GPU, the maximum throughput is about 6 GB/s. If it only assigns 8 lanes (often the case on multi GPU boards) the bandwidth is halved to 3 GB/s. Consequently in an ideal usage scenario all data can be kept on the GPU for the whole program runtime. Another model to mitigate the impact of slow data transfers is to overlap them with calculations. Using asynchronous data transfers and streams, it is possible to execute a kernel while concurrently copying data to and from the device. On Fermi-level Tesla products, which feature two memory transfer controllers, it is even possible to upload and download data at the same time in addition to performing calculations.

Double-precision floating-point operation throughput is much lower than that of single precision floating points. With the exception of professional Fermi level GPUs (such as C2050 and Quadro 6000 where it is 1/2) double-precision performance is 1/8th of the single precision performance. Often this is not as restricting as one might think, since the other limitations are more important.

As CPUs, GPUs have special function units (SFUs) to execute transcendental instructions such as sine, cosine, reciprocal, and square root. While there are only four SFUs per multiprocessor (so eight cycles are needed to execute a warp), these units can work independent of the standard cores. As a consequence, special function operations can actually be hidden behind normal operations if there are enough of those in a kernel and the number of running threads per multiprocessor is large.

Atomic functions are used to solve the problem of write conflicts, when multiple threads try to alter the same memory element at the same time. Generally, it is only guaranteed that one thread succeeds in executing its operation. For example, if all threads of a warp add a value to a global memory element, only one of the values will be actually added. Atomic operations solve this problem by locking the memory element until a thread has finished its operation. All other threads must wait until the current operation is finished. Therefore atomic operations are extremely slow even compared to random device memory access and should be avoided if possible.

A last thing to keep in mind is the register latency. If a value is written to a register, the next read of the same register can only occur after 6 cycles. To avoid idling cores due to this requirement up to 192 threads (6 warps) need to be launched per multiprocessor. The following code illustrates this point:

```
1  int a = input1+input2;
2  int b = a*4;
```

A single warp has to wait 6 cycles after line 1 in order to execute line 2. That means there is time to execute line 1 with 5 other warps before the first one can continue.

2.1.4 Multi-GPU execution models

Developing code for multi-GPU systems requires another parallelization method in addition to the CUDA programming model. Depending on the hardware system a number of options exist. The most commonly used method for single node multi-GPU systems (i.e. a workstation with 2-4 GPUs) are pthreads and OpenMP, which work relatively similarly. A single host process launches multiple threads, which share a common working memory. Typically the number of launched threads is equal to the number of processor cores on the node, a significant difference from the CUDA model where many more threads are launched, than there are compute cores on the GPU. Much of the parallelization of an algorithm with OpenMP can be done with compiler directives. The reduction example from the small CUDA introduction could look like this using OpenMP:

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N 4096

int main (int argc, char *argv[])
{
 int np = 2;
 if(argc>1) np = atoi(argv[1]);
 omp_set_num_threads(np);

 int a[N], my_sum, sum=0;

 for(int i=0; i<N; i++) a[i]=1;

 #pragma omp parallel private(my_sum)
 {
   my_sum = 0;

   #pragma omp for schedule(static,1)
   for(int i = 0; i < N; i++)
       my_sum = my_sum + a[i];

   printf("MySum: %i\n",my_sum);

   #pragma omp critical
   sum = sum + my_sum;
 }
 printf("\nTotal Sum %i\n",sum);
}
```

Except for the compiler directives, this code is the same as a serial program. These directives do three things: the first one in line 16 marks the starting point from which on multiple threads are used. It also declares `my_sum` as a private variable, which is not shared between all threads (i.e. it can have a different value for each thread). The second directive tells OpenMP to split the following `for`-loop between all threads. E. g. if two threads are launched the first one will process the contents of the `for`-loop for $i = 0$ to $i < N/2$ and the second will do it for $i = N/2$ to $i < N$. The third and last

directive in line 26 tells OpenMP to use a thread-safe access for the next operation (the equivalent to the `atomicAdd()` call in the CUDA example). Otherwise write conflicts could result in only one thread successfully adding its value to the total sum.

For multi-node systems, OpenMP cannot be used, since threads cannot share a common main memory area[7]. In these situations most often MPI (Message Passing Interface) employed. Here independent threads with their own separate memory exchange data via explicit communication routines [8]. The following code is the already-discussed reduction example implemented with MPI.

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#define N 4096

int main (int argc, char *argv[])
{
 long sum = 0;
 int my_sum,id;

 int num_threads;

 MPI_Init(&argc,&argv);                  /* starts MPI */
 MPI_Comm_rank(MPI_COMM_WORLD, &id);   /* get current process id */
 MPI_Comm_size(MPI_COMM_WORLD, &num_threads); /*get number of process */
 MPI_Status Stat;

 int nblocks = N/256/num_threads;
 int chunk = N/num_threads;

 int* a = new int[chunk];
 for(int i=0; i<chunk; i++) a[i]=1;

 for(int i=0; i<chunk; i++) my_sum += a[i];

 printf("my_sum: %i %i\n",id,my_sum);
 if(id == 0)
 {
   sum = my_sum;
   for(int i=1;i<num_threads;i++)
   {
     MPI_Recv(&my_sum, 1, MPI_INT, i, 1, MPI_COMM_WORLD,&Stat);
     sum += my_sum;
   }
   printf("\n %li\n",sum);
 }
```

[7] There are a number of efforts to implement OpenMP for distributed memory systems. Mostly they are based on so-called Distributed Shared Memory concepts, which work by keeping a copy of the whole shared memory on each node, and synchronize it as needed. Typically this becomes rather inefficient when the number of nodes grows.

[8] Many MPI implementations provide shared memory optimizations for situations in which communication occurs between threads on the same node. In these cases, data transfers can be done using the common main memory of the node.

```
  else
    MPI_Send(&my_sum, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);

40 MPI_Finalize();
}
```

Note the MPI library calls to set up the communication network and the explicit use of `MPI_Send()` and `MPI_Recv` operations. Another difference from the OpenMP model is that each thread owns only one part of the total array `a`, whereas in OpenMP every thread has access to the full array.

Both, OpenMP and MPI, can be used in conjunction with CUDA. Each OpenMP or MPI thread will request its own GPU and offload work to it. Data synchronization between GPUs now involves one more step than synchronization between host threads, because the data has to be copied back from the device to the host. Combining the previous examples with the CUDA code of Sec. 2.1.2 in order to perform a reduction in parallel over multiple GPUs is straightforward. Basically one has to replace the part where the actual reduction takes place (lines 20 to 22 in the OpenMP code and line 24 in the MPI program) with the GPU setup function calls and the kernel launch of the CUDA program. This leads to the following code using OpenMP:

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
4 #define N 4096

int main (int argc, char *argv[])
{
  ...
9
 #pragma omp parallel private(my_sum)
 {
   my_sum = 0;

14   int nblocks=N/256/np;
     int chunk = N/np;

     int id = omp_get_thread_num();

19   int* a_dev;
     int* b_dev;

     cudaSetDevice(id);
     cudaMalloc(&a_dev,sizeof(int)*chunk);
24   cudaMalloc(&b_dev,sizeof(int));
     cudaMemcpy(a_dev,a+id*chunk,chunk*sizeof(int),cudaMemcpyHostToDevice);

     cuda_reduce<<<nblocks,256>>>(a_dev,b_dev);

29   cudaMemcpy(&my_sum,b_dev,sizeof(int),cudaMemcpyDeviceToHost);
     printf("MySum: %i\n",my_sum);
```

```
    #pragma omp critical
    sum = sum + my_sum;
34  }
   printf("\nTotal Sum %i\n",sum);
}
```

The omitted parts are the same as in the pure OpenMP example and the CUDA kernel is the same as in the pure CUDA reduction code. Note that the variables declared within the `#pragma omp parallel` clause (e.g. `a_dev` and `b_dev`) are private to each thread by definition. A difference from the kernel launch call of the pure CUDA program is that fewer blocks are used and each OpenMP thread copies another part of the array `a` to the device.

The MPI multi-GPU example is slightly more complex. In particular, the CUDA functions are put into an extra file and are called as an '**extern** "C"' function within the main program. While this is not necessary, the separation of CUDA and MPI parts into different files has one distinct advantage. It allows the usage of the MPI-compiler wrappers commonly employed to compile MPI programs. They are almost a necessity on large clusters, because they set all the correct compiler flags and paths automatically. These settings often are specific for each cluster in order to make optimal use of the available network features. But the CUDA code still has to be compiled with `nvcc` and as long as these compiler wrappers do not support CUDA directly, the CUDA parts of an MPI program have to be compiled separately.

```
#include <cstdlib>
extern "C" int cuda_reduce_host(int id,int chunk,int nblocks,int* a);

 4 int cuda_reduce_host(int id,int chunk,int nblocks,int* a)
   {
    int* a_dev;
    int* b_dev;
    int my_sum;
 9  cudaSetDevice(id);
    cudaMalloc(&a_dev,sizeof(int)*chunk);
    cudaMalloc(&b_dev,sizeof(int));
    cudaMemcpy(a_dev,a,chunk*sizeof(int),cudaMemcpyHostToDevice);

14  cuda_reduce<<<nblocks,256>>>(a_dev,b_dev);

    cudaMemcpy(&my_sum,b_dev,sizeof(int),cudaMemcpyDeviceToHost);

    return my_sum;
19 }
```

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#define N 4096
5 extern "C" int cuda_reduce_host(int id,int chunk,int nblocks,int* a);

int main (int argc, char *argv[])
{
```

```
    long sum = 0;
10  ...

    my_sum = cuda_reduce_host(id,chunk,nblocks,a);

    printf("my_sum: %i %i\n",id,my_sum);
15  ...
    MPI_Finalize();
}
```

The modification of the main routine is minor compared to that of the pure MPI program. The '**extern** "C"' function `cuda_reduce_host()` is declared and the loop to add the elements of `a` is replaced by the call to that function. The function `cuda_reduce_host()` resembles closely the original pure CUDA program discussed in Sec. 2.1.2.

In principle, one can now also combine all three concepts. On a node OpenMP can be used to parallelize over multiple GPUs, while MPI is employed for inter node communication. The advantage over a pure MPI program would be the potential reduction of network traffic. But the worthiness of the additional development effort depends largely on the problem itself.

To summarize, it is, in principle, not more complicated to make an MPI or OpenMP program use GPUs with CUDA than it is to rewrite a serial code to use CUDA. Though a straight forward approach often will not give the most efficient program, since an optimal partition of a problem onto multiple OpenMP or MPI threads is not necessarily an optimal partition for multiple GPUs. In fact, this is especially true for OpenMP.

Because OpenMP is used for shared memory systems, the cost for a thread to access the memory working set of an other thread is the same as accessing its own memory. This is not true with multiple GPUs since transferring data from one GPU to another always involves the latency and limited bandwidth of the PCIe express bus. Even though CUDA 4.0 now provides a unified address space over all GPUs of one node (so they can be used as a shared memory system), this transfer cost remains. As a consequence, algorithms which depend on a shared memory architecture with uniformly fast data access wont work well for multiple GPUs using OpenMP. On the other hand, it is usually assumed that data transfer between threads in MPI programs is slow compared to accessing the own memory. Hence, these programs tend to be optimized for minimal data transfers, and are thus better suited for a direct adaption into a multi-GPU code.

The full versions of all reduction examples, together with compile instructions, can be found in Appendix B without line numbers, for convenient copying and pasting.

2.2 Overview of LAMMPS$_{CUDA}$

2.2.1 Design principles

Numerous GPU-MD codes have been under development during the past four years. Some of those are new codes (HOOMD [18], AceMD [19], HALMD [20]), others are extensions or modifications of existing codes (e.g. NAMD [21], Amber [22], LAMMPS [23]). Most of these projects are yet of limited scope and cannot compete with the

rich feature sets of legacy CPU-based MD codes. This is not surprising considering the amount of development time which has been spent on the existing codes; many of them have been under development for more than a decade. Furthermore, some of these GPU-MD codes have been written to accelerate specific compute-intensive tasks, limiting the need to implement a broad feature set.

Our goal is to provide a GPU-MD code that can be used for simulation of a wide array of materials classes (e.g. glasses, semiconductors, metals, polymers, biomolecules, etc.) across a range of scales (atomistic, coarse-grained, mesoscopic, continuum).

LAMMPS can perform such simulations on CPU-based clusters. It is a classical MD code, that has been under development since the mid-1990s, is freely-available, and includes a very rich feature set.

As building such simulation software from scratch would be an enormous task, we instead leverage the tremendous effort that has gone into LAMMPS, and enable it to harness the compute power of GPUs. We have written a LAMMPS "package" that can be built along with the existing LAMMPS software, thereby preserving LAMMPS' rich feature set for users while yielding tremendous computational speedups. Other important LAMMPS features include an extensive scripting system for running simulations, and a simple-to-extend and modular code infrastructure that allows for easy integration of new features. Most importantly it has an MPI-based parallelization infrastructure that exhibits good scaling behavior on up to thousands of nodes. Finally, starting with an existing code like LAMMPS and building GPU versions of functions and classes one by one allows for easy code verification.
Our objectives can be summarized as follows (in order of decreasing priority):

(i) maintain the rich feature set and flexibility of LAMMPS,

(ii) achieve the highest possible speed-ups,

(iii) allow good parallel scalability on large GPU-based clusters,

(iv) minimize code changes,

(v) implement easily maintainable code,

(vi) make the GPU capabilities easy for LAMMPS users to invoke.

All of these design objectives have implications for design decisions, yet in many cases they are competing objectives. For example objective (i) implies that the different operations of a simulation have to be done by different modules, and that the modules can be used in any combination requested by the user. This in turn means that data, such as the particle positions, are loaded multiple times during a single simulation step from the device memory, which results in a considerably negative effect on the performance of the simulation. Another slight performance hit is caused by the use of templates for the implementation of pair forces and communication routines. While this greatly enhances maintainability, it adds some computational overhead. By keeping full compatibility with LAMMPS we were able to minimize the GPU-related changes that users will need

to make to existing input scripts. In order to use LAMMPS$_{\text{CUDA}}$ it is often enough to add the line "package cuda" at the beginning of an existing input script. This triggers the use of GPUs for all GPU-enabled features in LAMMPS$_{\text{CUDA}}$, while falling back to the original CPU version for all others.

The limiting factors of the targeted architecture also greatly influence design decisions. Since those have been discussed in detail elsewhere [24], here we only list the most important factors:

(a) in order to use the full GPU, thousands or even tens of thousands of threads are needed,

(b) race conditions are a problem when multiple threads access the same data,

(c) data transfer between the host and the GPU is slow,

(d) the ratio of device memory bandwidth to computational peak performance is much smaller than on a CPU,

(e) latencies of the device memory are large,

(f) random memory accesses on the GPU are serialized,

(g) 32 threads are executed in parallel.

As a consequence of (b), LAMMPS$_{\text{CUDA}}$ generally uses full neighbor lists, i.e. each interaction between two particles i and j is calculated twice (for particle i the force f_{ij} is calculated and for particle j the force f_{ji}). That way a force f_i can be updated by a single thread (or block), avoiding racing conditions and write conflicts.

Considering (c), we decided to minimize data transfers between device and host by running as many parts of the simulation as possible on the GPU. This decision distinguishes our approach from other GPU extensions of existing MD codes (e.g. the regular GPU package for LAMMPS), where only the most computationally-expensive pair forces are calculated on the GPU.

2.2.2 Features

Currently LAMMPS$_{\text{CUDA}}$ supports 26 pair force styles; long range Coulomb interactions via a particle-particle/particle-mesh (PPPM) algorithm; NVE, NVT, and NPT integrators; and a number of LAMMPS "fixes." In addition, pair force calculations on the GPU can be overlapped with bonded interactions and long range Coulomb interactions if those are evaluated on the CPU. All of the bond, angle, dihedral, and improper forces available in the main LAMMPS program can be used. Simulations can be performed in single (32-bit floats) and double (64-bit floats) precision, as well as in a mixed precision mode, where only the force calculation is done in single precision while the time integration is carried out in double precision. In addition to the requirements of LAMMPS, only the CUDA toolkit (available for free from NVIDIA) is needed. Currently only NVIDIA

GPUs with a compute capability of 1.3 or higher are supported. This includes GeForce 285, Tesla C1060 as well as GTX480 and Fermi C2050 GPUs.

Since July 2011, the package has been part of the main LAMMPS distribution. LAMMPS is available under the GNU Public License and can be downloaded from `http://lammps.sandia.gov/`, where detailed installation instructions and feature lists can be found. LAMMPS$_{\text{CUDA}}$, which is encapsulated in the USER-CUDA package of LAMMPS, should not be confused with LAMMPS' "GPU" package, which has some overlapping capabilities (see Figures 6 and 8) and is also available from the same website.

2.2.3 The program work-flow

Before discussing a detailed execution in the next chapter, we will describe the general principles using a simplified version. The latter is of interest for the general reader, while the former is targeted at potential developers and provides in-depth details of the execution order.

The simplified work-flow chart of our implementation is shown in Figure 3.

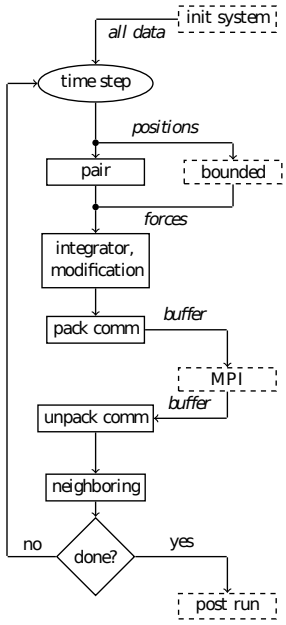


Figure 3: LAMMPS$_{CUDA}$ work-flow, dashed boxes are done on the CPU, while solid boxes are done on the GPU.

Initializing the process (e.g. setting up the MPI communication routines, reading the input script, generating atoms, etc.) is done on the CPU. At this state it is generally assumed that the CPU has the correct set of data. So if during the setup a GPU routine has to be called (as for example an initial force evaluation) all data is copied from the host to the GPU and results are transferred back. Before beginning the loop over all

time-steps, all data is put on the device. From this point on, it is assumed that the GPU owns the correct data.

Since LAMMPS is using spatial decomposition in order to parallelize simulations over multiple nodes, the positions of all atoms for which the distance to a domain border is less than the interaction radius, must be known to the neighboring process. The corresponding data transfer is performed using MPI. In LAMMPS$_{CUDA}$ the communication buffer is build on the GPU and subsequently transferred to the host ("`pack comm`"). It is then directly sent to the neighboring process. Correspondingly, the received buffer is directly copied to the GPU and is opened only there ("`unpack comm`").

LAMMPS uses neighbor lists for evaluating the pair interactions [25]. That means for every atom, a list with its neighbors is kept in memory. This approach allows to only evaluate a fraction of the interatomic pair distances in order to know which atoms are within the interaction cutoff of each other. As a consequence, the pair force evaluation scales linearly with the number of total atoms. Reneighboring is the process of rebuilding the neighbor lists. By using a neighbor radius that is larger than the interaction radius, it is possible to avoid the need to reneighbor in every time-step. As already stated in Sec. 2.2.1 LAMMPS$_{CUDA}$ uses full neighbor lists instead of half neighbor lists like it is usually done in the classical LAMMPS code. Therefore each atom keeps a list of all its neighbors. This approach requires roughly two times as much memory and also doubles the numbers of interactions which are calculated.

Also in all time-steps where reneighboring occurs, atoms are exchanged between processes. So each atom which has crossed the domain border of a process is packed into a buffer during the communication phase and sent to the neighboring process.

Pair forces are evaluated on the GPU (more on that in section 2.3.2). If bonded interactions are used, those will be computed on the CPU at the same time as the GPU is calculating pair forces. To facilitate this, the positions of all atoms are transferred from the GPU to the host before evaluating forces. After the interactions are computed, bonded forces are copied back from the host to the GPU and summed onto the forces stemming from the pair interactions. The rational behind this approach is that evaluation of bonded interactions usually takes much less time than pair force computation. Hence, the CPU can often perform this task within the time the GPU needs for pair force evaluation. As a consequence, bonded interactions can be used with only minor performance costs along side pair forces, when using LAMMPS$_{CUDA}$. An exception are some bio-simulations in which heavy usage of bonded interactions is made. In these cases the GPU has to wait for the CPU to finish its task. Therefore, it might be worth to implement bonded interactions on the GPU as well in the future.

2.3 Implementation details

2.3.1 File and object relations

LAMMPS$_{CUDA}$ is an object based C++ code which is using external C routines to invoke CUDA Kernels. In order to keep the code structure clear, the naming of files follows the class names - e.g. the class "FixNVE" is found in the files fix_nve.h and fix_nve.cpp.

For each class of the original LAMMPS, which needs to call its own CUDA kernels, a new class is created with the name extension "Cuda". Those classes also own three more files in addition to "class_name_cuda.h" and "class_name_cuda.cpp". The first is "class_name_cuda_cu.h" and contains the headers for any functions compiled by NVCC which are called from C++ routines (typically within the "class_name_cuda.cpp"). The second file, named "class_name_cuda.cu", entails the host side routines compiled with NVCC. The actual CUDA kernels are found in the third file: "class_name_cuda_kernel.cu". During compilation for each file "class_name_cuda_kernel.cu" an object file is created.

An exception to this scheme are the pair force classes. Their CUDA part is compiled into a single object file named "cuda_pair.o" since they all use the same template based kernels found in "cuda_pair.cu and cuda_pair_kernel.cu.

There are four main groups of classes in LAMMPS: base classes, pair_forces, fixes and computes. Each type of two body potential is encapsulated in its own class. For example the classical Lennard Jones potential with a cutoff is found in the class PairLJCut, while the Morse potential is found in the class PairMorse. The CUDA versions of the pair_force classes are generally derived from the standard classes. This eliminates the need to duplicate functions such as the parameter interpretation or output of pair_style specific data into restart files.

Fixes are operations which are called either every time step or every nth time step during the integration loop of the LAMMPS execution. This includes operations such as the time integration itself, external forces as gravity or an electric field, and things like explicit moment conservation.

Computes are classes which serve the purpose of calculating a quantity from the system state. For example temperature, pressure and mean square displacement are calculated through computes. Computes are not automatically executed but called by other classes such as fixes.

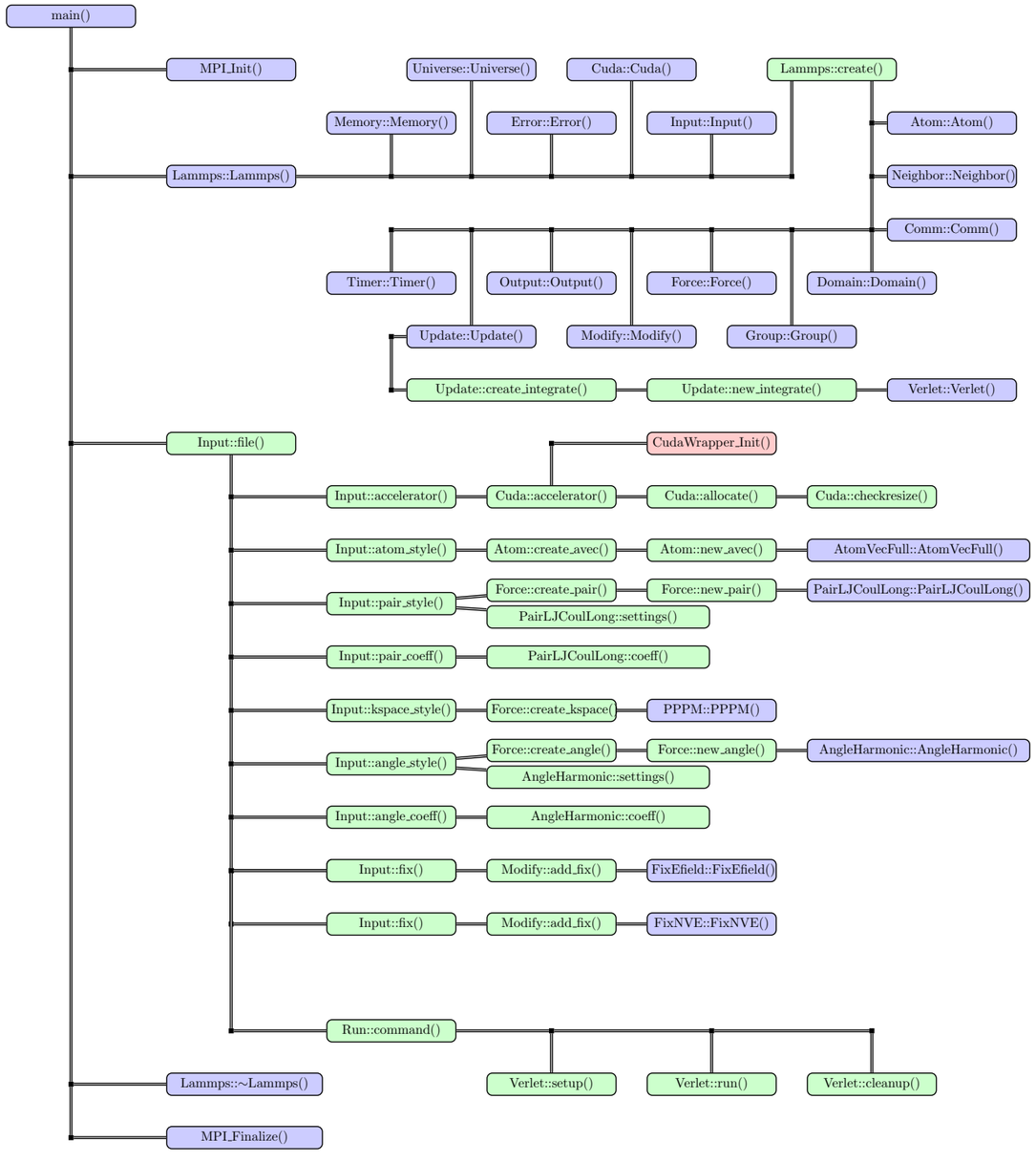
The most important base classes are the neighbor class and the comm class. The neighbor class provides various routines to generate neighbor lists. A class which needs a neighbor list (e.g. a pair_force) requests its calculation from the neighbor class. The comm class entails the routines to communicate between MPI processes. It will be discussed in some more detail in Sec. 2.3.3.

The following execution plans picture the relations between the various classes. It shows where functions are called and can also be read as a workflow plan. Since it would be close to impossible to picture the complete class function hierarchy of LAMMPS, this plan is limited to what is actually called during the execution of the following script:

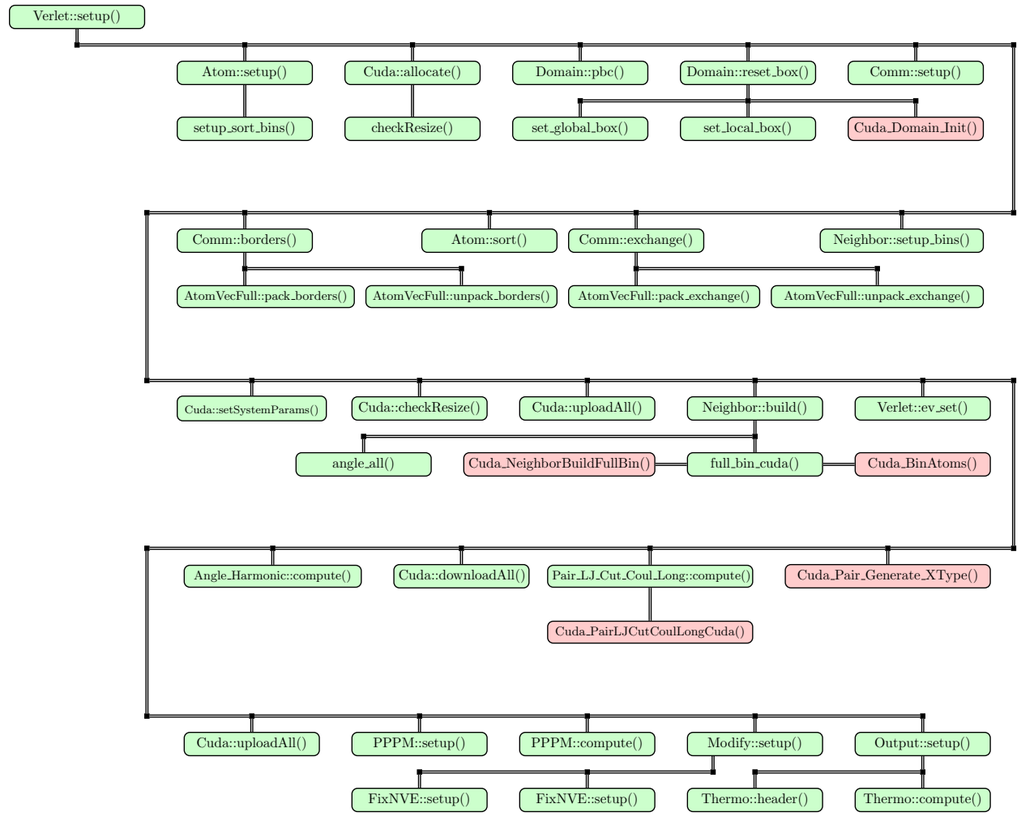
```
package cuda gpu/node 3

atom_style full

read_data silicate.data
pair_style lj/cut/coul/long 10.0
pair_coeff 1 1 20117023442.1621 0.0657  0.0
pair_coeff 1 2 247244.172408168 0.20851 1631.1766
pair_coeff 2 2 40514.4105526705 0.35132 4951.9369

kspace_style pppm/cuda 1e-5

angle_style harmonic
angle_coeff 59.7114742 109.47

fix 1 efield 1.0 0.0 0.0
fix 2 nve
run 1000
```

The nodes are named following the scheme `ClassName::FunctionName` for host side functions. Functions found in the `liblammpscuda.a` are given by their name. In order to read the plan according to the sequence of execution one generally follows the paths and takes every branch. For example it begins with the `main()` routine, which calls first the `MPI_Init()` and after that the constructor of the `LAMMPS` class. In that constructor, first some other top level classes are created (e.g. `Memory` and `Input`) before the `create` function of the `LAMMPS` class is called. That function in turn creates some more instances of various top level objects. Next the `main()` routine calls the input script interpreter found in `Input::file()`.

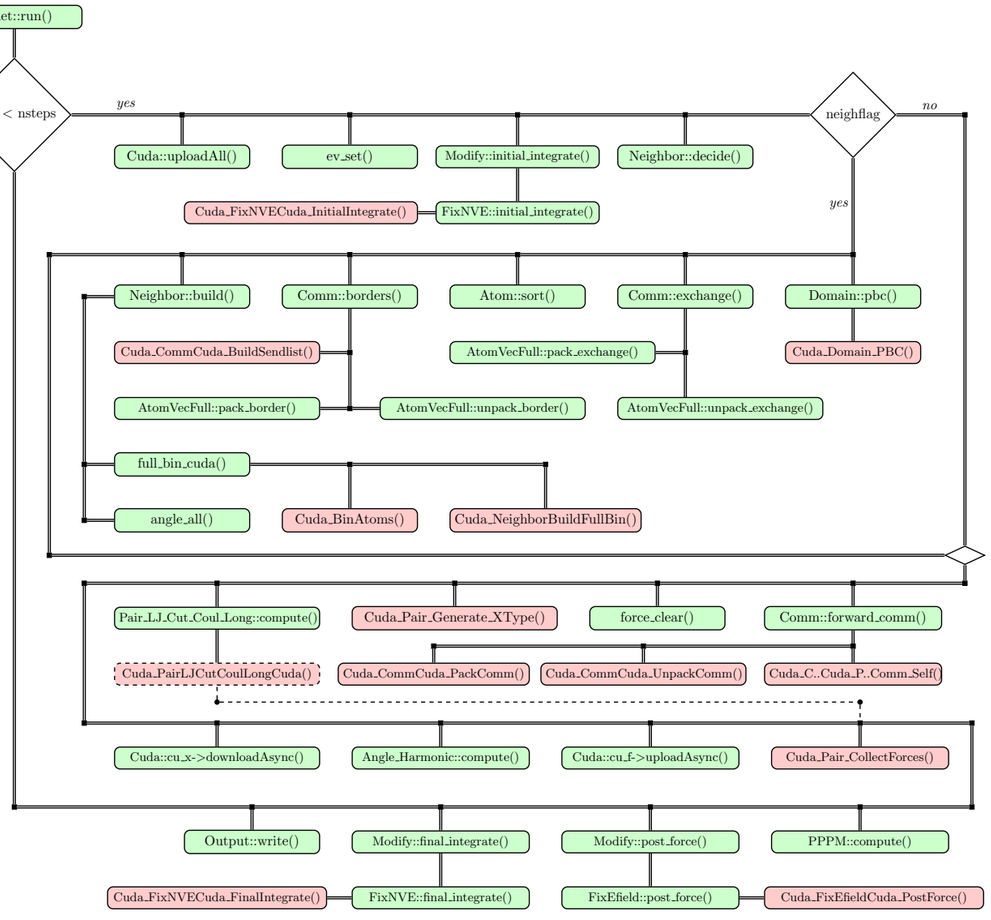
LAMMPS startup, class construction, and script parsing.



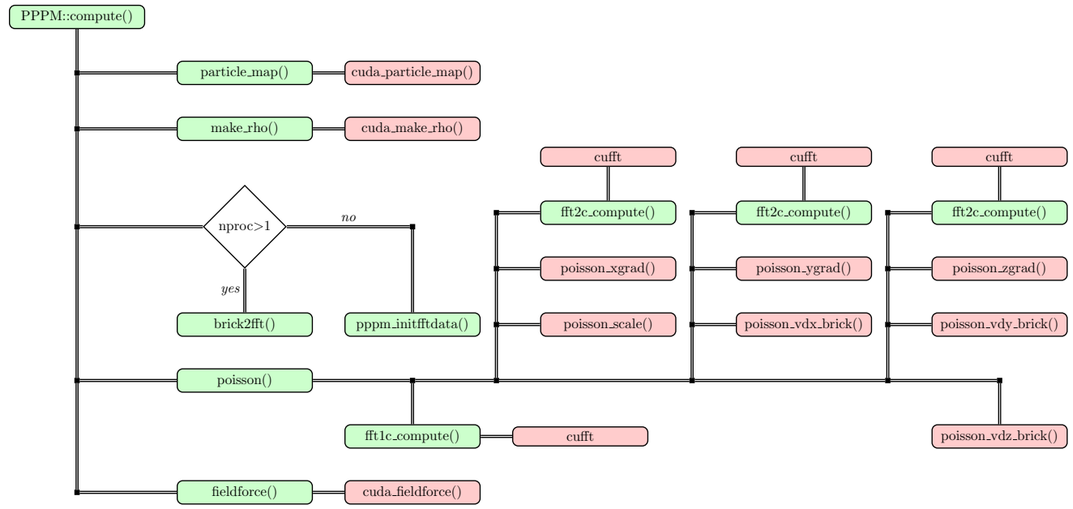
System and run setup.

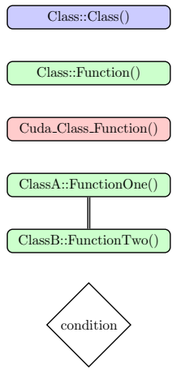


Time integration loop.



PPPM compute function.





Constructor of a class. The implementation can be found in the source file `class.cpp`.

A class function which is executed on the host. The implementation can be found in the source file `class.cpp`.

A function which is executed on the GPU. The implementation can be found in the source file `class.cu`.

`FunctionOne` of `ClassA` calls `FunctionTwo` of `ClassB`. If `ClassB` os ommitted, `FunctionTwo` is a member of `ClassA` as well.

A conditional branching point.

2.3.2 Pair interactions

There are three commonly used algorithms for calculating non bonded pair interactions in MD simulations. The first possibility is to use direct summation, in which each of the N atoms in a system interacts with each other. This method is often referred to as the N^2 approach, since it amounts to N^2 force calculations. While being the easiest to implement, this approach has the disadvantage of being very slow for large systems.

Often it is not necessary to calculate each possible interaction if far away particles do not influence each other much. In these cases the force calculations are only performed if two particles i and j are not farther apart from each other than a given cutoff r_c. This generally limits the number of interaction partners for each atom to a maximum number of neighbors given by the density of the system making it an $O(N)$ problem. The prerequisite is that the force is short range in nature, i.e. it goes faster to zero than r^{-3}, with r being the interatomic distance[9]. Most potentials fulfill this requirement, with the notable exception of gravity and the Coulomb force. Those have to be handled by an N^2 method or more sophisticated algorithms, which usually employ some degree of approximation (e.g. Ewald summation, PPPM, and Multipol extension [26]).

Calculating the interactions exclusively between atoms in proximity requires nevertheless some method to decide which atoms are close enough after all. Two approaches are typically used: cell lists and neighbor lists [25].

In the cell list approach the simulation box is divided into small sub domains called cells. For each cell a list is made of which atoms belong to it. By choosing the cells as large as the interaction radius[10] only the pair forces between atoms of neighboring cells need to be calculated. Since that cell list approach is discussed in detail in the Bachelor Thesis of Lars Winterfeld [16], we shall only state that it is inferior to the neighbor list approach.

The third class of algorithms (and the most commonly used ones) for calculating non-bonded pair interactions are neighbor list methods. These methods are based on maintaining a list of neighbors for each atom. Generally, neighbor list solutions tend to be faster than the other methods (see the discussion in the thesis of Lars Winterfeld and in [17] as well as) but they also use much more memory. While a cell list or N^2 method rarely needs more than 150 bytes per particle, a program using neighbor lists often requires 300 to 4000 bytes per atom. These numbers are plausible when comparing the number of neighbors of an atom - which typically is in the range of 50 to 1000 - with the relative low number of properties and states which must be saved for each particle.

It is worth noting, that for building the neighbor lists a cell list approach is used in LAMMPS. Moreover the range up to which two atoms are considered neighbors is set to

[9] The requirement for a potential $U(|\vec{r}|)$ to be short ranged is, that $\forall \varepsilon > 0 : \exists r_c \in \mathbb{R}^+ : \int_V U(|\vec{r}|) \cdot H(|\vec{r}| - r_c) dV < \varepsilon$, where H is the Heavyside step function. Expressing the integral in spherical coordinates one gets $\epsilon > 2\pi \int_{r_c}^{\infty} U(r) r^2 dr$. For the purpose of MD simulations it is often enough to make this requirement for the forces instead of the potential.

[10] For reasons discussed in Lars Winterfelds Bachelor thesis, one actually chooses a cell size smaller than the neighbor cutoff and calculates interactions between cells which are not nearest neighbors as well.

r_{neigh}, a value larger than the actual interaction cutoff r_{cut}. That way it is not necessary to rebuild the neighbor list every time step, but only if an atom has moved more than $0.5 \cdot (r_{\text{neigh}} - r_{\text{cut}})$ since the last reneighboring.

In designing a neighbor list approach that uses blocks of threads as given by the CUDA model, it becomes clear that there are two main ways that the force calculation work can be divided up among the threads. The first possibility is to use one thread per atom (TpA), where the thread loops over all of the neighbors of the given atom. The second possibility is to use one block per atom (BpA), where each of the threads in the block loops over its designated portion of the neighbors of the given atom. In the following, pseudo-code for both algorithms are given:

TpA algorithm:

```
i = blockId*ThreadsPerBlock+threadId;
load(i) // coalesced access
for(jj = 0; jj<numneigh[i];
      jj++) {
  j <- neighbors[i][jj]
  load(j) // random access
  ftmp+=calcPairForce(i,j)
}

ftmp -> f[i] // coalesced access
```

BpA algorithm:

```
i = blockId
load(i) // coalesced access
for(jj = 0; jj<numneigh[i];
      jj+=ThreadsPerBlock) {
  j <- neighbors[i][jj]
  load(j) // random access
  ftmp+=calcPairForce(i,j)
}
reduce(ftmp)
ftmp -> f[i] // coalesced access
```

Both algorithms ostensibly have the same number of instructions. However, when considering looping it becomes clear that the BpA algorithm requires the execution of a larger total number of lines of code. Additionally the BpA algorithm uses the relatively expensive reduction of $ftmp$ that is not needed by the TpA algorithm. BpA also requires the use of a much larger total number of blocks. For further clarification, Table 1 lists the number of times each line of code is executed, taking into account the number of blocks used, and considering that 32 threads of each block are executed in parallel.

While this seems to indicate that TpA would always be faster, cache usage has to be taken into account as well. In order to reduce random accesses in the device memory while loading the neighbor atoms (limiting factor (e)), one can cache the positions using the texture cache. (We also tested global cache on Fermi GPUs, but it turns out to be slower due to its cache line size of 128 bytes.) This strategy improves the speed of

Table 1: Number of executions per line for the BpA and TpA algorithms

Lines	TpA	BpA
1,2,9,10:	natoms/32	natoms
5,6,7:	(natoms/32)*nneigh	natoms*(nneigh/32)

both algorithms considerably, but it helps BpA more than TpA. The underlying reason is that less atoms are needed simultaneously with BpA than with TpA. As a result, BpA allows for better memory locality, and therefore the re-usage of data in the cache is increased (assuming atoms are spatially ordered). Revisiting table 1 makes it evident that this better cache usage becomes increasingly important with an increasing number of neighbors, corresponding to an increased pair cutoff distance.

Consequently, one can expect a crossover cutoff for each type of pair force interaction, where TpA is faster for smaller cutoffs and BpA for larger. Unfortunately it is hard to predict where this cutoff lies. It not only depends on the complexity of the given pair force interaction, but also on the hardware architecture itself. Therefore, a short test is the best way to determine the crossover cutoff. The timing ratios shown in Figure 4 indicate that the force calculation time can depend significantly on the use of BpA or TpA. Generally, the differences are larger when running in single precision than when running in double precision. For the LJ system, an increase of the cutoff from $2.5\sigma_0$ to $5.0\sigma_0$ can turn the 30% TpA advantage into a 30% BpA advantage. Therefore, we decided to implement both algorithms, and allow dynamic selection of the faster algorithm using a built-in mini benchmark during the setup of the simulation. This ensures that the best possible performance is achieved over a wide range of cutoffs. While our particular findings are true only for NVIDIA GPUs, one can expect that similar results would be found on other highly parallel architectures with comparable ratios of cache to computational power.

2.3.3 Communication

In order to facilitate its parallelization on clusters, LAMMPS uses spatial decomposition. That means each subprocess governs its own spatial domain of the simulation box. Since particles within the pair force cutoff distance to the border of their domain need to know the positions of the nearby atoms of the neighboring domain, LAMMPS uses the concept of ghost atoms. Therefore a process does not only keep its own particles in the memory but also the atoms which are close enough to its domain.

The positions of these ghost atoms must be updated every time step, which constitutes the majority of all occurring communication. In contrast, constant properties of the atoms such as charge, mass and type only need to be transferred periodically. Transferring them just once is not enough, since the particles which are close enough to a border to be used as ghost atoms change over time. The third major communication

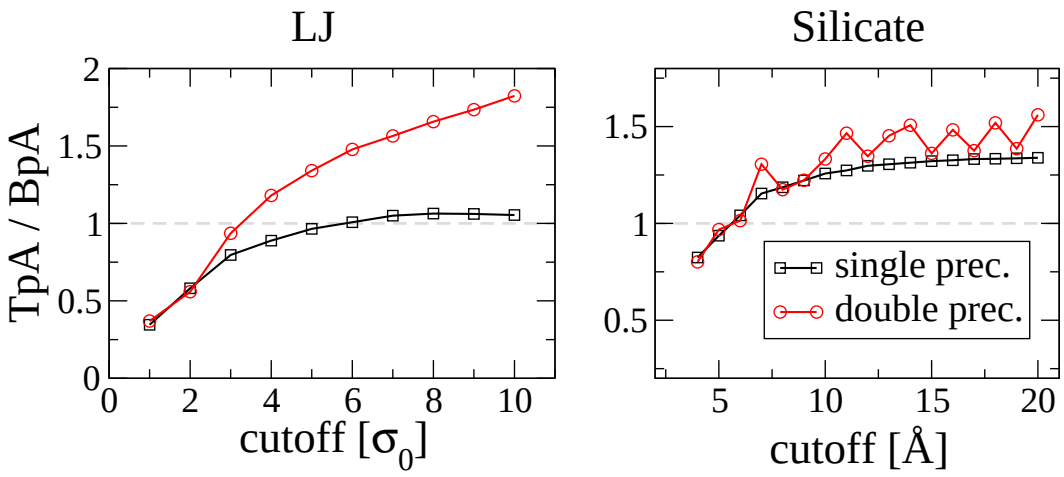


Figure 4: Computation time comparison of BpA and TpA algorithms for two different benchmark systems in single and double precision. The ratio of the force calculation time using the TpA algorithm and the force calculation time using the BpA algorithm is shown as a function of the cutoff. System: LJ (32k atoms), Silicate (12k atoms) (see Appendix C); Hardware: CL (see Appendix D)

routine facilitates the transfer of atoms from one box to another, when they cross the border.

Exchange of atoms and border atoms only takes place in timesteps when reneighboring occurs (see Sec. 2.3.1). First the data of all atoms, which are outside the local box is put into a list, with a separate list made for each dimension in space. The local copy of those atoms is deleted in that process. Then, the lists are sent to the respective neighboring processes, where the atoms are added to the local list of particles[11].

After all atoms have reached the MPI process to whoms sub domain they belong, a list of the ghost atoms is built for each dimension in space. An atom belongs to that list if it is within the communication cutoff, which is commonly set to the same value as the neighbor cutoff. According to that list, the properties of each atom added are added to a buffer, which will be needed for the force evaluation. While this commonly includes type, positions and charge, values such as mass and velocity are left out in order to minimize network traffic. The ghost atom list is also used for the forwarding of atom positions at each time step.

In simplified pseudo code the communication routine looks like this:

```
if(not do_reneighbor) {
```

[11] In fact, before adding an atom to the local list, it is checked whether that atom is actually within the local box. This has a twofold purpose: (i) it is needed if the sub boxes are so small that the interaction range of one sub box spans multiple neighboring domains in a given direction and (ii) if an atom moved too far away to be added to any of the neighboring boxes the simulation will stop with a "lost atom" message. This is a sensible thing to do, since in that case the interaction radius or the so-called "neighbor cutoff" was chosen too small.

```
    for(int dim = 0; dim<3 ; dim++) { /*for each dimension*/
      commbuffer=cuda_pack_forward_comm(ghostlist[dim],dim);
      MPI_SendRecv(commbuffer);
      cuda_unpack_forward_comm(commbuffer);
    }
  } else {
    for(int dim = 0; dim<3 ; dim++) { /*for each dimension*/
      commbuffer=cuda_pack_exchange_comm(dim);
      MPI_SendRecv(commbuffer);
      cuda_unpack_exchange_comm(commbuffer,dim);
    }
    for(int dim = 0; dim<3 ; dim++) { /*for each dimension*/
      cuda_build_ghostlist(ghostlist[dim],dim);
    }
    for(int dim = 0; dim<3 ; dim++) { /*for each dimension*/
      commbuffer=cuda_pack_ghosts_comm(ghostlist[dim],dim);
      MPI_SendRecv(commbuffer);
      cuda_unpack_ghosts_comm(commbuffer);
    }
  }
```

The actual code can be found in the `VerletCuda::run()` function.

While all these functions have already been implemented in the CPU version of LAMMPS, it is unfortunately not trivial to adapt them for efficient execution on a GPU. Most of the problems are caused by racing conditions. As an example of a typical difficulty the parallel building process of the list of ghost atoms shall be discussed here. The serial code is relatively trivial[12]:

```
int nsend = 0;                      /*number of atoms which are added*/
for(int i=0; i<nlocal; i++) {       /*loop over all atoms in domain*/
  if (x[i] >= lo && x[i] <= hi) {   /*check if atom is close to border*/
    ghostlist[nsend] = i;           /*add atom to list of ghost atoms*/
    nsend++;                        /*increase counter*/
  }
}
```

Doing the same in parallel can be simple as well if atomic operations are used:

```
/*nsend is a pointer to a global variable*/
int i=blockIdx.x*gridDim.x+threadIdx.x     /*loop over all atoms in domain*/
if(i<nlocal) {
  if (x[i] >= lo && x[i] <= hi) {          /*is atom close to border?*/
    ghostlist[atomicAdd(nsend,1)] = i;     /*add atom to ghost atoms list*/
  }
}
```

The key operation here is the `atomicAdd()` function. It simultaneously increases the counter `nsend` by one and returns its previous value. As explained in section 2.1.3 usage of the `atomicAdd()` operation is necessary since a standard write operation would only be guaranteed to succeed for one thread at a time. So if 100 threads try to increase

[12]The code examples are somewhat simplified. Among other points, they are only given for one dimension in space.

`nsend` at the same time, without an atomic operation it might only be increased by one in total. While the code using `atomicAdd()` looks nice and simple, it is also very slow, because potentially thousands of threads will try to access `nsend` at the same time. This will result in serialized "lock, write, free" cycles, which in the end will be much slower than the serial code.

The trick is to use shared memory for a first reduction for each block. That way only one thread of each block has to use the costly `atomicAdd()` operation to update the global counter.

```
  /*nsend is a pointer to a global variable*/
  int i=blockIdx.x*blockDim.x+threadIdx.x;
3 bool add = false;
  if(i<nlocal)
    if (x[i] >= lo && x[i] <= hi) {
      add=true;
    }
8
  shared[threadIdx.x]=add?1:0;

  /* example state of shared with blockDim.x=8: [0 1 1 0 0 1 0 1 ?]*/
  __syncthreads();
13
  int mynsend = 0;
  if(threadIdx.x==0) {
    for(int k=0;k<blockDim.x;k++) {
      if(shared[k]) {mynsend++; shared[k]=mynsend;} /* add up contributios,
18                and note which internal position in the list an atom gets */
    }
    shared[blockDim.x]=atomicAdd(nsend,mynsend);
  }
  /* state of shared is now: [0 1 2 0 0 3 0 4 nsend]*/
23 __syncthreads();

  mynsend=shared[blockDim.x]+shared[threadIdx.x]-1;
  /* mynsend ranges from nsend to nsend+3 */
  if(add&&mynsend<maxlistlength)
28   list[mynsend] = i;
```

The reader might notice that no parallel reduction was used in lines 14-23. That choice was made because it is not enough to just determine the number of atoms which need to be added to the list. Every thread of a block also has to know where it should put its atom.

2.3.4 Long range Coulomb interactions

LAMMPS provides two modules to do long range Coulomb calculations, one is using the Ewald algorithm and the other the Particle Particle - Particle Mesh (PPPM) method [27]. Both algorithms are so-called "KSpace" methods. Here only the basic concepts shall be repeated without giving a full derivation (those can be found in [27] and [26] as well as in a number of textbooks). The idea of both methods is to split the Coulomb

sum into two parts. The first part must be short ranged, so that a cutoff can be used to make the calculation of it an $O(N)$ problem. The second part must be short ranged after Fourier transformation. Commonly this splitting is achieved by adding and subtracting gaussian counter charge densities for each point charge q_i at position $\vec{r}_i$:

$$U = \sum_i \sum_{j,i<j} \sum_{\vec{L}} \frac{q_i q_j}{\vec{r}_{ij} + \vec{L}} \Rightarrow \sum_i \sum_{j,i<j} \sum_{\vec{L}} \left[\frac{q_i q_j}{\vec{r}_{ij} + \vec{L}} - \iint \frac{\rho_i(\vec{r})\rho_{j+L}(\vec{r}')}{\vec{r} + \vec{r}'} d\vec{r} d\vec{r}' \right] + \sum_i \sum_{j,i<j} \sum_{\vec{L}} \iint \frac{\rho_i(\vec{r})\rho_{j+L}(\vec{r}')}{\vec{r} + \vec{r}'} d\vec{r} d\vec{r}', \quad (1)$$

where the sum over $\vec{L}$ stems from the periodic boundary conditions of the MD simulation and is the lattice vector of the simulation box. The charge density $\rho_i(\vec{r})$ is given by:

$$\rho_i(\vec{r}) = q_i \left(\frac{G^2}{\pi} \right)^{\frac{3}{2}} e^{G(\vec{r} - \vec{r}_i)}. \quad (2)$$

After combining the first two terms and Fourier transformation of the last term in Eq.1 one arrives at the usual Ewald formula:

$$U = \sum_i \sum_{j,i<j} \sum_{\vec{L}} \frac{q_i q_j \mathrm{erfc}(G|\vec{r}_{ij} + \vec{L}|/\sqrt{2})}{\vec{r}_{ij} + \vec{L}} + \sum_{\vec{k} \neq 0} \frac{4\pi}{L^3 k^2} e^{-k^2/2G^2} |S(\vec{k})|^2 - \frac{G}{\sqrt{2\pi}} \sum_i q_i^2, \quad (3)$$

with $S(\vec{k}) = \sum_{j=1}^{N} q_j e^{i\vec{k}\cdot\vec{r}_j}$ and $\vec{k} = \frac{2\pi}{L}\vec{n}$, $n = 0..\infty$. The first term of Eq. 3 is short ranged with respect to $|\vec{r}_{ij} + \vec{L}|$ and the second is short ranged with respect to k, so for both sums a cutoff can be used.

In the PPPM method one does not use the exact charge densities $\rho(\vec{r})$ for the Fourier transformation, but one approximates it with a discrete density $\rho(\vec{r}_g)$. The advantage is, that the Fourier transformation can now be done with a Fast Fourier Transformation (FFT) algorithm.

In summary the PPPM algorithm consists of four basic steps:

i Generate the discrete charge density $\rho(\vec{r}_g)$.

ii Fourier transform the charge density to get $\rho(\vec{q})$.

iii Calculate the electric field $E(\vec{r}_g)$ on the grid by inverse Fourier transformation of $i\vec{q}T_n(\vec{q})\rho(\vec{q})$.

iv Calculate the force on the particles by the grid through interpolation of the electric Field.

For the most part, adapting the CPU algorithm for a GPU is straightforward. The two exceptions are the generation of the discrete charge density, and the 3D FFT when using multiple MPI processes.

Distributing the particle charges onto a grid in parallel potentially leads to write conflicts, which can be avoided by using atomic operations. Unfortunately at the time of first implementation, the GPUs only supported `atomicAdd()` for integer variables and even now Fermi level GPUs support it only for single precision floating point variables in addition. The solution employed here is the use of fixed precision arithmetic. Basically, the contribution of each particle to the charge density at a grid point is multiplied by a fixed scaling factor and only the integer part is added. Thus one can use `atomicAdd()` and avoid write conflicts. After the discrete charge density is calculated in fixed precision, it is converted back into floating point variables by dividing it again by the scaling factor. In order to maximize the valid digits in this scheme, it is checked that at least one of the three highest bits of any of the grid points is used. Also it is necessary to prevent an overflow. If either non of the bits is set or an overflow occurred, the scaling factor is increased or reduced by a factor of two respectively and the calculation is repeated. After finding a good scaling factor during the first time step, an adjustment of the factor (and thus a repetition of the charge density calculation) rarely occurs. The pseudo code for this operation (using simple `for` loops instead of kernel calls on the GPU) is:

```
  overlow = 0;
2 highbit = 0;
  while(overflow || (not highbit)) {
    highbit = 0;
    overflow = 0;
    #do parallel on GPU for nlocal threads
7   for(i = 0; i<nlocal; i++) {
      cells = determine_cells_of_atom(i);
      for(cells) {
        scaled_charge = Scale*part_charge(i);
        tmp_charge = atomicAdd(cell_charge_int , int(scaled_charge));
12      if(scaled_charge+tmp_harge > MAXINT) overflow++;
        if(tmp_charge>MAXINT/32) higbit++;
      }
    }
    if(overflow) Scale/=2;
17  else if(not highbit) Scale*=2;
  }

  #do parallel on GPU for nlocal threads
  for(allcells) {
22  cell_charge = cell_charge_int;
  }
```

In terms of precision, through this scheme at least 29 bits are used for valid digits (one is used for the sign, and two are allowed to be unused). Using single precision floats one has only 24 bits for valid digits. So as long as the charge values of two different grid points do not differ by a factor larger than 2^5, the fixed precision algorithm uses at least as many valid digits per grid point as an equivalent single precision floating point implementation. This can always be achieved by increasing the real space cutoff and simultaneously decreasing the number of grid points used in the PPPM method.

The second problem is the parallel 3D FFT. It is so communication intensive that

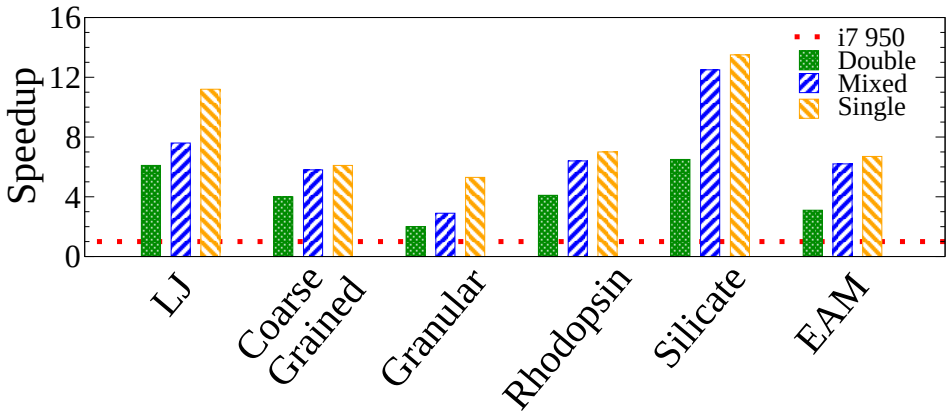


Figure 5: Typical speed-up when using a single GTX 470 GPU running LAMMPS$_{CUDA}$ versus a quad-core Intel i7 950 running four threads of LAMMPS for various system classes. Systems: see Figure (see Appendix C); Hardware WS$_B$ (see Appendix D)

copying the whole charge grid to each GPU and doing the full 3D FFT there turned out to be the fastest solution. This reduces the scalability of the PPPM implementation in LAMMPS$_{CUDA}$ significantly. It might be possible to improve on this situation by employing more sophisticated comunication-calculation overlapping schemes in addition to using the new multi-GPU features offered by CUDA 4.0 - but that must be left for future work.

2.4 Performance evaluation

2.4.1 Single-GPU performance

To assess the possible performance gains of harnessing GPUs, we have performed benchmark simulations of several important classes of materials. Both the regular CPU version of LAMMPS and LAMMPS$_{\text{CUDA}}$ were run on our workstation B (WS$_B$) with an Intel i7 950 quad-core processor and a GTX 470 GPU from NVIDIA. Simulations on the GPU were carried out in single, double, and mixed precision. We compare the loop times for 10,000 simulation steps. The results shown in Figure 5 are proof of an impressive performance gain. Even in the worst-case scenario, a granular simulation (which has extremely few interactions per particle), the GPU is 5.3 times as fast as the quad-core CPU when using single precision and 2.0 times as fast in double precision. In the best-case scenario the speed-up reaches a factor of 13.5 for the single precision simulation of a silicate glass involving long range Coulomb interactions. Single precision calculations are typically twice as fast as double precision calculations, while mixed precision is somewhere in between. It is worthy to note that this factor of two between single and double precision is reached on consumer grade GeForce GPUs, despite the fact that their double precision peak performance is only 1/8th of their single precision peak performance. This is a strong sign that LAMMPS$_{\text{CUDA}}$ is memory bound.

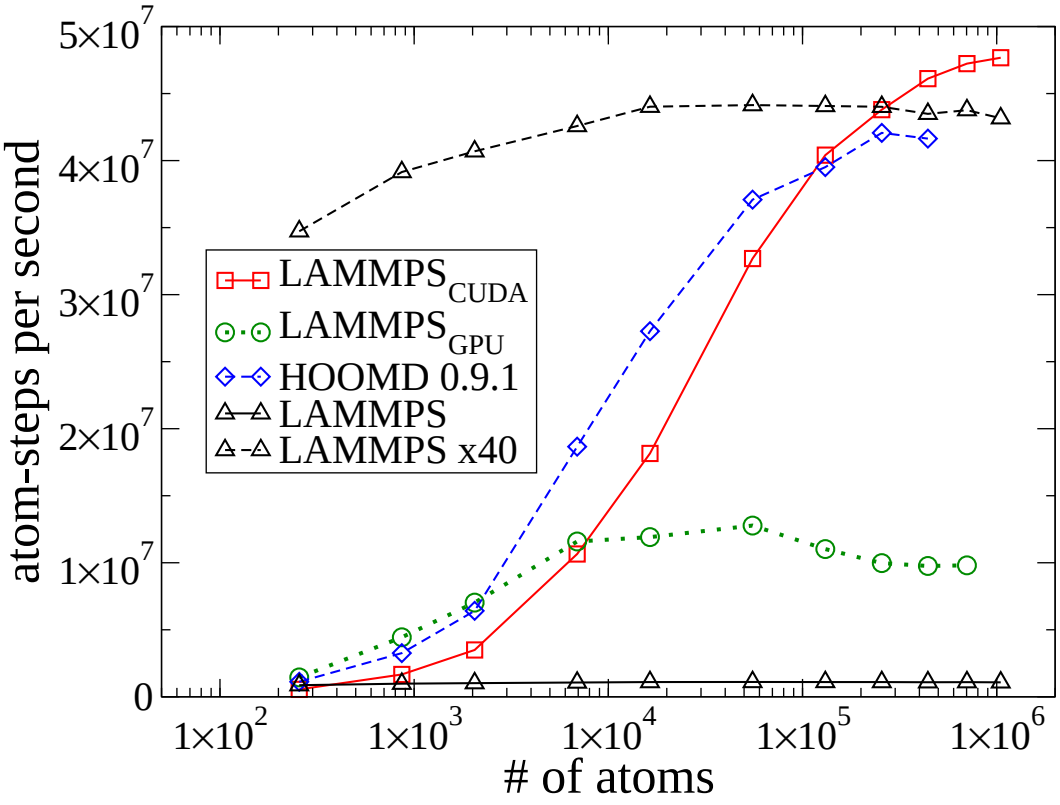


Figure 6: Performance in number of atom-steps per second of a GTX 470 GPU and a single core of an i7 950 CPU. LAMMPS$_{\text{CUDA}}$ approaches its maximum performance only for system sizes larger than 200,000 particles. The same system was also run using HOOMD version 0.9.1 and the "GPU" package of LAMMPS. The CPU curve has been plotted with a scaling factor of 40 as well to make it easier to see. System: LJ (see Appendix C); Hardware: WS$_B$ (see Appendix D)

Generally, the speed-up increases with the complexity of the interaction potential and the number of interactions per particle. Additionally the speed-up also depends on the system size. As stated in section 2.2.1 the GPU needs many threads in order to be fully utilized. This means that the GPU cannot reach its maximum performance when there are relatively few particle-particle interactions. This point is illustrated in Figure 6, where the number of atom-steps per second is plotted as a function of the system size. As can be seen, at least 200,000 particles are needed to fully utilize the GPU for this Lennard-Jones system. In contrast the CPU core is already nearly saturated with only 1,000 particles. All systems used to produce Figure 5 were large enough to saturate the GPU.

We have also plotted the performance curves for the GPU-MD program HOOMD (version 0.9.1) and the "GPU" package of LAMMPS in Figure 6 for comparison purposes. The characteristics of HOOMD are very similar to LAMMPS$_{\text{CUDA}}$. It reaches its top

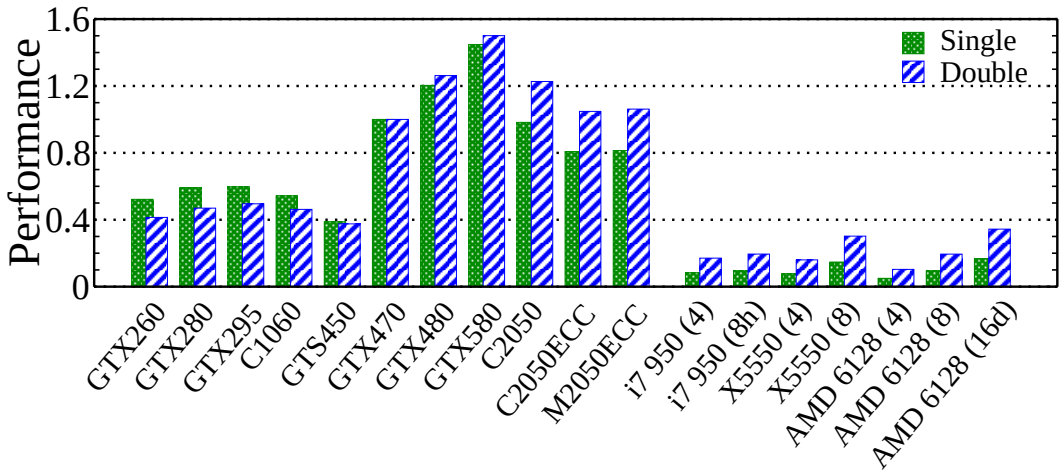


Figure 7: Relative performance of various GPUs and CPUs in single and double precision, in relation to a GTX470. The numbers are the average of the performance ratio of the given processor to a GTX 470 GPU for the Lennard Jones melt and the silicate glass benchmark (see Appendix C). Note that the CPU version of LAMMPS always uses double precision.

performance at about 200,000 particles. Interestingly HOOMD is somewhat slower than $\text{LAMMPS}_{\text{CUDA}}$ at very high particle counts, while it is significantly faster at system sizes of 16,000 particles and below. This can probably be explained by the fact that HOOMD is a single GPU code, whereas $\text{LAMMPS}_{\text{CUDA}}$ has some overhead due to its multi-GPU capabilities. The "GPU" package of LAMMPS reaches its maximum performance at about 8,000 particles. While it is faster than $\text{LAMMPS}_{\text{CUDA}}$ for smaller systems (and even faster than HOOMD for fewer than 2,000 particles), it is significantly slower than $\text{LAMMPS}_{\text{CUDA}}$ and HOOMD for this LJ system at large system sizes. The reason is most likely that the "GPU" package of LAMMPS only off-loads the pair force calculations and the neighbor list creation to the GPU, while the rest of the calculation (e.g. communication, time integration, thermostats) is performed on the CPU. This requires a lot of data transfers over the PCI bus, which reduces overall performance and sets an upper limit on the speed-up. On the other hand, at very low particle counts the CPU is very efficient at doing these tasks that are less computationally demanding and memory bandwidth limited. While a GPU has a much higher bandwidth to the device memory than does the CPU to the RAM, the whole data set can fit into the cache of the CPU for small system sizes. So for the smallest system sizes, the CPU can handle these tasks more efficiently than the GPU, leading to the higher performance of the "GPU" package.

Fig. 7 shows the relative performance of various GPUs and CPUs compared to a GTX470. The performance numbers were determined from running the Lennard Jones melt benchmark system and the silicate glass benchmark, since those two are largely unaffected by the host performance. On the GPUs one MPI thread has been used, while

4 to 16 threads have been used for the CPU runs. Two of the CPU runs were performed using two processors on dual socket boards, one with two X5550 Intel quad-cores and one with two AMD6128 octo-cores. Generally, the performance of the GPUs scales as one would expect from theoretical performance numbers (memory bandwidth and peak compute performance). Indeed the relative numbers reflect what is commonly found in gaming benchmarks.

One outlier are the C2050/M2050 GPUs, which provide more double precision performance than the equivalent GTX470 consumer GPU. Nevertheless the 20% advantage is far away from the factor of four suggested by the theoretical double precision peak performance numbers. This shows that LAMMPS$_{CUDA}$ is not much limited by double precision computations. It is worth noting that a low to mid range GPU as the GTS450, which currently can be bought for as few as 80 €, can beat a dual Xeon server even in double precision. The latter system currently costs about 2,300 €for the mainboard and the two processors alone.

2.4.2 Multi-GPU performance

In order to simulate large systems within a reasonable wall clock time, modern MD codes allow parallelization over multiple CPUs. LAMMPS's spatial decomposition strategy was specifically chosen to enable this parallelization, allowing LAMMPS to run efficiently on modern HPC hardware. Depending on the simulated system, it has been shown to have parallel efficiencies[13] of 70% to 95% for up to several ten thousand CPU cores. To split the work between the available CPUs, LAMMPS's spatial decomposition algorithm evenly divides the simulation box into as many sub-boxes as there are processors. MPI is used for communication between processors. During the run, each processor packs particle data into buffers for those particles that are within the interaction range of neighboring sub-boxes. Each buffer is sent to the processor associated with each neighboring sub-box, while the corresponding data buffers from other processors are received and unpacked.

While the execution time of most parts of the simulation should in principle scale very well with the number of processors, communication time is a major exception. With an increasing number of processors, the fraction of the total simulation time which is used for inter-processor communication grows. This is already bad enough for CPU-based codes, where switching from 8 to 128 processors typically doubles or triples the relative portion of communication. But for GPU-based codes, the situation is even worse since the compute-intensive parts of the simulation are executed much faster (typically by a factor of 20 to 50). It is therefore understandable why it is essential to perform as much of the simulation as possible on the GPU. Consider the following example: In a given CPU simulation, 90% of the simulation time is spent on computing particle interaction forces. Running only that part of the calculation on the GPU, and assuming a 20-fold

[13] We define atom-steps per second as number of simulated steps s times the number of atoms n divided by the wall clock time t: $k = s \cdot n \cdot t^{-1}$. Let k_s denote the atom-steps per second for a single CPU run, and let k_m denote the atom-steps per second for an m CPUs run. Parallel efficiency, p, is then the ratio of k_s to k_m multiplied by m: $p = \frac{k_s}{k_m} m$.

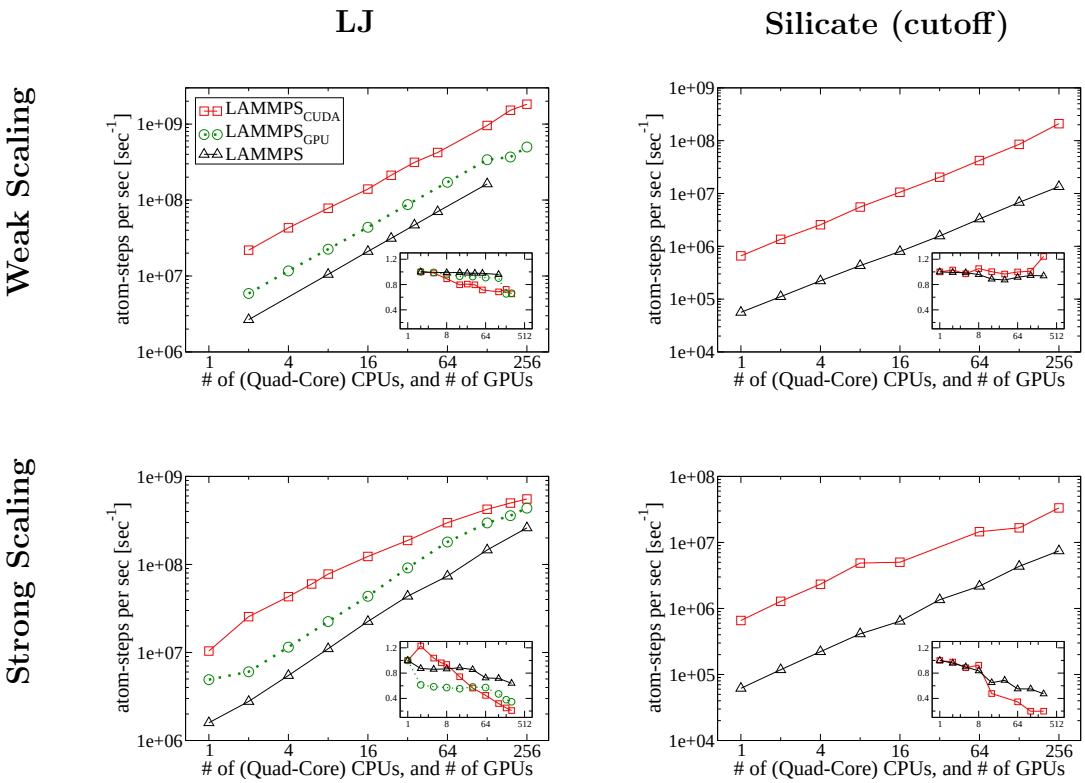


Figure 8: Multi-node scaling comparison for a fixed system size setup (strong scaling), and a constant number of atoms per node setup (weak scaling). High parallel efficiency is harder to achieve for the strong scaling case. The insets show the parallel efficiencies. Each node includes 2 GPUs and 2 quad-core CPUs. Systems: LJ, silicate (cutoff) (see Appendix C); Hardware: Lincoln (see Appendix D)

speed-up in computing the forces, the overall speed-up is only a factor of 6.9. If we then assume that with an increasing number of processors the fraction of the force calculation time drops to 85% in the CPU version, then the overall speed-up would be a factor of 5.2 only. So on top of the usual parallel efficiency loss of the CPU code, additional parallel efficiency is lost for the GPU-based code when only the pair forces are calculated on the GPU.

If one processes the rest of the simulation on the GPU as well, the picture gets somewhat better. Most of the other parts of the simulation are bandwidth bound, thus typical speed-ups are around 5. Taking the same numbers as before yields an overall speed-up of 15.4 and 13.8, respectively. So if parts of the code that are less optimal for the GPU are also ported, not only will single node performance be better, but the code should also scale much better. While the above numbers are somewhat arbitrary, they illustrate the general trend.

In order to minimize the processing time on the host, as well as minimize the amount of data sent over the PCI bus, LAMMPS$_{\mathrm{CUDA}}$ builds the communication buffers on the GPU. The buffers are then transferred back to the host and sent to the other processors via MPI. Similarly, received data packages are transferred to the GPU and only opened there.

Actual measurements have been performed on NCSA's Lincoln cluster, where up to 256 GPUs on 128 nodes were used (see Figure 8). We compare weak and strong scaling behavior of LAMMPS$_{\mathrm{CUDA}}$ versus the CPU version of LAMMPS for two systems. In the weak scaling benchmark, the number of atoms per node is kept fixed, such that the system size grows with increasing number of nodes. In this way, the approximate communication-to-calculation ratio should remain fairly constant, and the GPUs avoid underutilization issues. In the strong scaling benchmark, the total number of atoms is kept fixed regardless of the number of nodes used. This is done in order to see how much a given fixed-size problem can be accelerated. Note that in Figure 8, we plot the number of quad-core CPUs rather than the number of individual cores. Please also note that in general the Lincoln cluster would not be considered a GPU-"based" cluster since the number of GPUs per node is relatively small and two GPUs share a single PCIe2.0 8x connection. This latter issue represents a potential communication bottleneck since there are synchronization points in the code prior to data exchanges. Consequently, both GPUs on a node attempt to transfer their buffers at the same time through the same PCIe connection. On systems in which each of the (up to four) GPUs of a node has its own dedicated PCIe2.0 16x slot, the required transfer time would be as little as 1/4th of the time on Lincoln, thus allowing for even better scaling. Since Lincoln is not intended for large-scale simulations, it features only a single data rate (SDR) InfiniBand connection with a network bandwidth that can become saturated when running very large simulations.

Nevertheless, Figure 8 shows that very good scaling is achieved on Lincoln. There, the number of atom-steps per second (calculated by multiplying the number of atoms in the system by the number of executed time-steps, and dividing by the total execution time) is plotted against the number of GPUs and quad-core CPUs that were used. We

tested two different systems: a standard Lennard-Jones system (density 0.84 σ^{-3}, cutoff 3.0 σ_0), and a silicate system that uses the Buckingham potential and cutoff coulombic interactions (density 0.09 $Å^{-3}$, cutoff 15 Å). While keeping the number of atoms per node constant, the scaling efficiency of $\text{LAMMPS}_{\text{CUDA}}$ is comparable to that of regular CPU-based LAMMPS. Even at 256 GPUs (128 nodes), a 65% scaling efficiency is achieved for the Lennard-Jones system that includes 500,000 atoms per node. And a surprising 103% scaling efficiency is achieved for the silicate system run on 128 GPUs (64 nodes) and 34,992 atoms per node. This means that for the silicate system, 128 GPUs achieved more than 65 times as many atom-steps per second than 2 GPUs. In this case, a measured parallel efficiency slightly greater than unity is probably due to non-uniformities in the timing statistics caused by other jobs running on Lincoln at the same time.

We were also able to run this Lennard-Jones system with LAMMPS's "GPU" package. As already seen in the single GPU performance, the GPU package is about a factor of three slower than $\text{LAMMPS}_{\text{CUDA}}$ for this system. The poorer single GPU performance leads to slightly better scaling for LAMMPS's "GPU" package.

Comparing the absolute performance of $\text{LAMMPS}_{\text{CUDA}}$ with LAMMPS at 64 nodes gives a speed-up of 6 for the Lennard-Jones system and a speed-up of 14.75 for the silicate system. Translating that to a comparison of GPUs versus single CPU cores means speed-ups of 24 and 59, respectively.

Such larger speed-ups are observed up to approximately 8 nodes (16 GPUs) in the strong scaling scenario, where we ran fixed-size problems of 2,048,000 Lennard-Jones atoms and 139,968 silicate atoms on an increasing number of nodes. With 32 GPUs (16 nodes) the number of atoms per GPU gets so small (64,000 and 4,374 atoms, respectively) that the GPUs begin to be underutilized, leading to much lower parallel efficiencies (see Figure 6). At the same time, the amount of MPI communication grows significantly. In fact, for the silicate system with its large 15 Å cutoff, each GPU starts to request not only the positions of atoms in neighboring sub-boxes, but also positions of atoms in next-nearest neighbor sub-boxes. This explains the sharp drop in parallel efficency seen at 32 GPUs. In consequence, 256 GPUs cannot simulate the fixed-size silicate system significantly faster than 16 GPUs. On the other hand, those 16 GPUs on 8 nodes are faster than all 1024 cores of 128 nodes when using the regular CPU version of LAMMPS.

We also tested the "GPU" package of LAMMPS for strong scaling on the Lennard-Jones system. For this test, its parallel efficency is lower than that of $\text{LAMMPS}_{\text{CUDA}}$ for up to 32 GPUs. For more than 32 GPUs, the "GPU" package shows stronger scaling than $\text{LAMMPS}_{\text{CUDA}}$. This can be ascribed to $\text{LAMMPS}_{\text{CUDA}}$'s faster single node computations and subsequently higher communication-to-computation ratio. (Note that each of the versions of LAMMPS discussed here has the same MPI communication costs.) In $\text{LAMMPS}_{\text{CUDA}}$, the time for the MPI data transfers actually reaches 50% of the total runtime when using 256 GPUs.

A simple consideration explains why the MPI transfers are a main obstacle for better scaling. Since the actual transfer of data cannot be accelerated using GPUs, it constitutes the same absolute overhead as with the CPU version LAMMPS. Considering that the rest of the code runs 15 to 60 times faster on a process-by-process basis, it is obvious

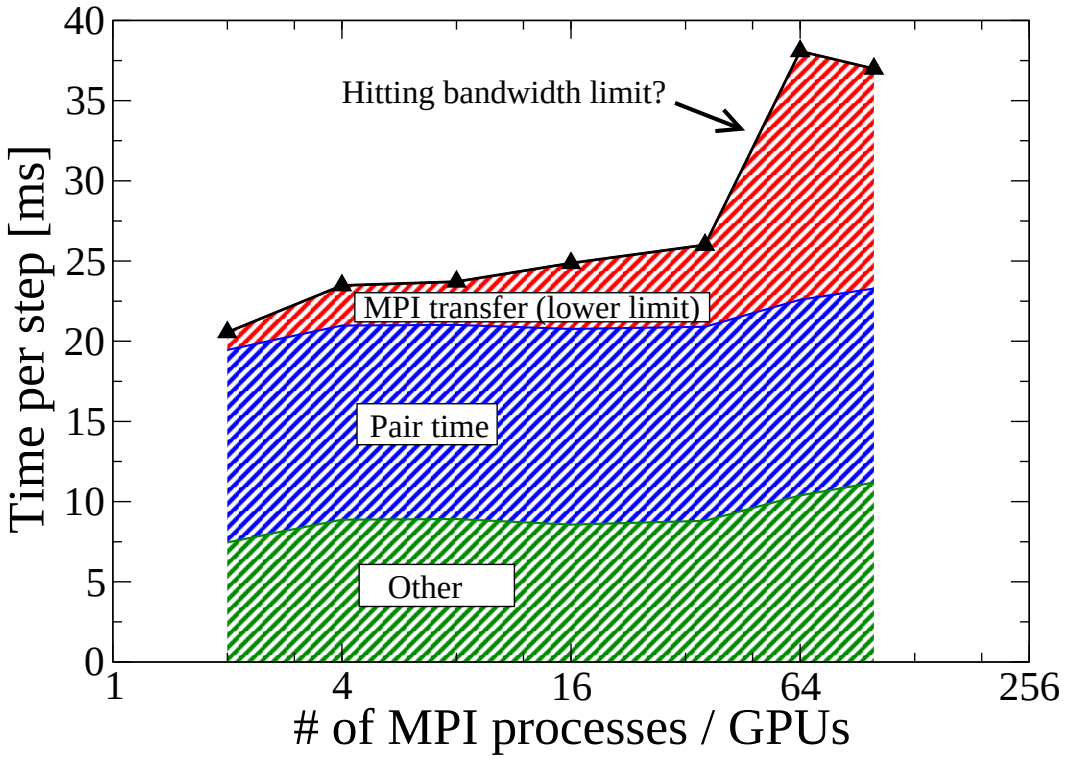


Figure 9: Portion of runtime spent on MPI transfers, pair force calculations, and other computations in a weak scaling benchmark. The increase of the total runtime is caused almost completely by the increase in the MPI transfer times. System: LJ (see Appendix C); Hardware: Lincoln (see Appendix D)

that if 1% to 5% of the total time is spent on MPI transfers in the CPU LAMMPS code, communication can become the dominating time factor when using the same number of GPUs with LAMMPS$_{\text{CUDA}}$.

That the MPI transfer time is indeed the main cause of the poor weak scaling performance can be shown by profiling the code. Figure 9 shows the total simulation time of the Lennard-Jones system versus the number of GPUs used. It is broken down into the time needed for the pair force calculation, a lower estimate of the MPI transfer times and the rest. The lower estimate of the MPI transfer time does not include any GPU↔host communication. It only consists of the time needed to perform the MPI send and receive operations while updating the positions of atoms residing in neighboring sub-boxes. All other MPI communication is included in the "other" time. Clearly, almost all of the increase in the total time needed per simulation step can be attributed to the increase in the MPI communication time. Furthermore, at 64 GPUs a sharp increase in the MPI communication time is observed. We presume that this can be attributed to the limited total network bandwidth of the single data rate InfiniBand installed in Lincoln. Considering the relatively modest communication requirements of an MD simulation (at

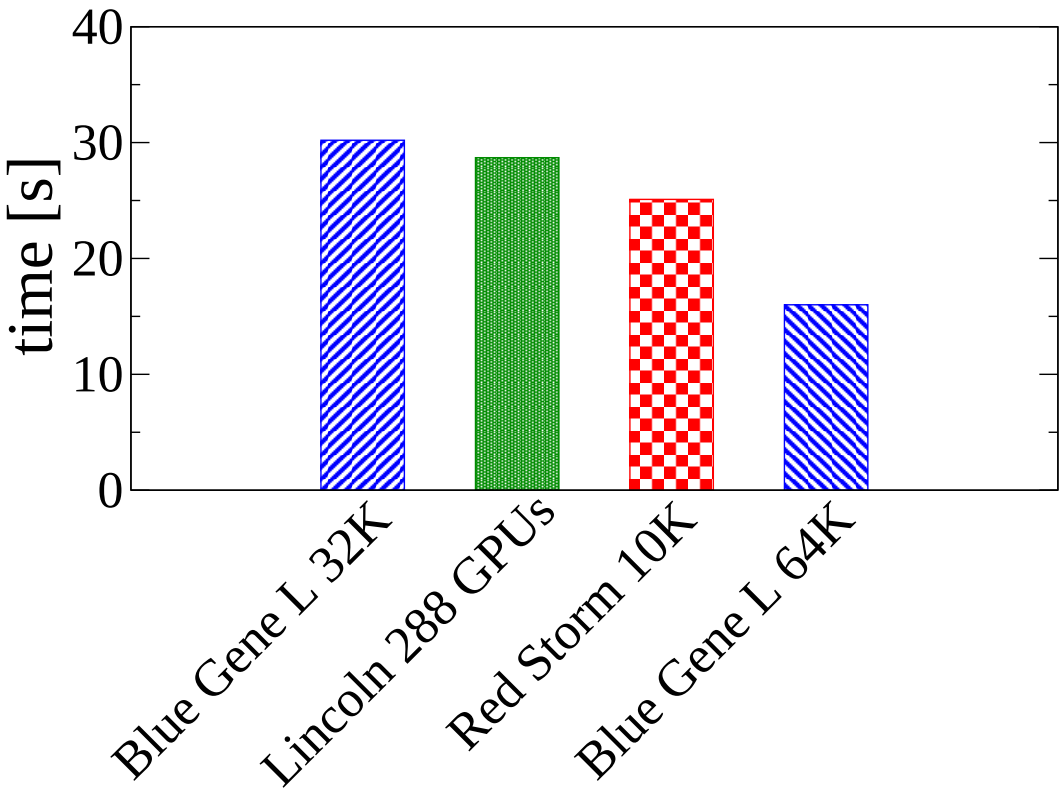


Figure 10: Loop time for 100 time-steps of a one billion particle Lennard-Jones system. On Lincoln, 288 GPUs were used. On the BlueGene/L system (with 32K processors and 64K processors) and RedStorm (with 10K processors), the regular CPU-based version of LAMMPS was used. System: LJ (see Appendix C); Hardware: Lincoln (see Appendix D)

least for this simple Lennard-Jones system), this finding illustrates how important high throughput network connections are for GPU clusters. In order to somewhat mitigate this problem, we have started to implement $\text{LAMMPS}_{\text{CUDA}}$ modifications that will allow a partial overlap of force calculations and communication. Preliminary results suggest that up to three quarters of the MPI communication time can be effectively hidden by that approach [28].

As a further example of what is possible with $\text{LAMMPS}_{\text{CUDA}}$, we performed another large-scale simulation. Using 288 GPUs on Lincoln, we ran a one billion particle Lennard-Jones system (Density: 0.844, Cutoff: 2.5 σ). This simulation requires about 1 TB of aggregate device memory. In Figure 10 loop times for 100 time-steps are shown for Lincoln, Red Storm (a Cray XT3 machine with 10368 processors sited at Sandia National Laboratories), and BlueGene/L (a machine with 65536 processor sited at Lawrence Livermore National Laboratory). The data for the latter two machines was taken from the LAMMPS homepage (`http://lammps.sandia.gov/bench.html`). Using 288 GPUs,

Lincoln required 28.7 s to run this benchmark, landing between Red Storm using 10,000 processors (25.1 s) and the BlueGene/L machine using 32 K processors (30.2 s).

2.5 Future of LAMMPS$_{CUDA}$ and outstanding problems

As of mid 2011 LAMMPS$_{CUDA}$ has reached a stable state, so that it can be employed for regular MD investigations. Considering its features, it is well positioned to establish itself as one of the major MD codes for current and future GPU cluster systems. A hint that this might indeed happen is the relatively large group of early adopters who are currently evaluating the capabilities of LAMMPS$_{CUDA}$.[14] This group includes the Sandia, Los Alamos and Oak Ridge National Laboratories, the HP Labs, the national supercomputing centers of Ireland (ICHEC) and Switzerland (CSCS), and a number of universities such as the ETH Zürich, the University of Western Sydney, the Imperial College London, and the Temple University (USA). In addition the US company Creative Consultans (from Albuquerque, NM) is already offering an online MD service using LAMMPS$_{CUDA}$ on their eStella cluster [29]. And last but not least a number of people at NVIDIA have been evaluating LAMMPS$_{CUDA}$ and are supporting customers with setting up systems to run simulations with it.

This trend of wide adoption is likely to accelerate soon, since LAMMPS$_{CUDA}$ will now be distributed as part of the main LAMMPS code. As a consequence, every user of LAMMPS will also download LAMMPS$_{CUDA}$ as part of it.

Regarding a future expansion of the feature list, there are four main areas of work:

(i) **Long range Coulomb scaling**

(ii) **Three body interactions**

(iii) **Bonded interactions**

(iv) **Overlapping communication**

Currently only the PPPM algorithm is implemented for solving the long range Coulomb problem on GPUs. Unfortunately it is not well suited for massively distributed computing, since it requires a 3D FFT operation. As already described in 2.3.4 the latter is a very communication intensive operation, so that it was better to run the full 3D FFT on each GPU separately. Consequently the PPPM part of a simulation severely limits the multi GPU scaling performance of LAMMPS$_{CUDA}$. There are at least two options to improve on the current situation.

First the latest CUDA 4.0 release added improved GPU to GPU communication routines. It might be possible to employ those in order to distribute the 3D FFT over multiple GPUs relatively efficiently.

A second alternative is to use a different algorithm. For example it might be wort it to look at so-called multipole methods. Their basic concept is to set up a hierarchy of

[14] The list of institutions using LAMMPS$_{CUDA}$ is based on the affiliations of people who actively requested support from me or gave general feedback.

increasingly large regions, and let sufficiently distant regions only interact via a multipole approximation (of their entailed subregions) instead of evaluating the interactions of the particles directly. While this method is not as effective as the PPPM algorithm it has less communication overhead and is an $O(N)$ method as opposed to $O(N \log N)$ for PPPM. Therefore it can potentially scale better on large multi-GPU systems [30].

Three body interactions are another important feature which is not yet available in LAMMPS$_{CUDA}$. As with pair interactions it is again important to avoid potential write conflicts. One possible solution is to calculate every force f_{ijk} for each particle individually at the cost of potentially increasing the number of computations by a factor of three.

A third important field that improvements could have a big impact on are bonded interactions. The current solution, where bonded interactions are computed on the host while the GPU evaluates the non-bonded interactions, works well if only a small number of explicit bonds are used for a simulation. But if the ratio of bonded interactions to non-bonded interactions is large (which often is the case in bio systems), the GPU might be finished with its work long before the host is done as well. We are now considering two approaches to solve this problem. Besides the obvious approach of doing the bonded interactions on the GPU, another alternative is to use more than one CPU core per MPI process to compute them. The latter approach would have the advantage of potentially higher speed up. It has also the appeal of using otherwise idling host resources. On the other hand using multiple CPU cores per MPI thread requires the introduction of another parallelization layer such as OpenMP. This might increase the complexity of the code by a significant amount.

Last but not least further development of the experimental capabilities for overlapping communication with computations is needed. While this is not a high priority for runs on workstations or small clusters (i.e. 2 to 16 GPUs), it might be worth the additional development time in situations where more than 64 GPUs are used.

3 Development of empirical force fields for mixed network former glasses

3.1 General structure of mixed network former glasses

The foundation of our current understanding of the structure of network glasses can be attributed to Zachariasen, who developed the "random network theory" for simple oxide glasses in 1932 [39]. Concluding from the similarities in mechanical properties of oxide glasses with crystalline materials of the same stoichiometry that both materials must be governed by the same interatomic forces, he postulated the existence of fundamental structural building blocks - the network former units (NFUs) - which form glasses and crystals alike. In glasses, Zachariasen suggested, these NFUs form a three-dimensional random network. He also proposed four rules for oxide glasses of the composition X_mO_n with X being B, Si, P, Ge, etc.:

(i) oxygen atoms are linked to one or two atoms X,

(ii) the oxygen coordination numbers of an atom X are small (usually three or four),

(iii) the NFUs share corners, but not edges or faces,

(iv) at least three corners are shared.

These rules allow the construction of both crystals and glasses (see Fig. 11). The difference is that in glasses the units are randomly connected to each other. The latter is mainly achieved through a broader distribution of X-O-X angles, while the NFUs are not disturbed much in comparison with the crystal.

The NFUs are either trigonal for compositions of the type X_2O_3 (for X = Al, B) or tetragonal for compositions of the type X_2O_n with $n > 3$ (for X = Si, Ge, P). If $n > 4$ so-called non-bridging oxygens occur - oxygens which are linked to only one atom of type X. Based on their number of non-bridging oxygens, the tetrahedral NFUs are classified as so-called Q^k species, where k is the number of bridging oxygens.

Upon addition of alkali oxide to these glasses, the network is modified, since more oxygen atoms must be accommodated. This leads to the creation of more non-bridging oxygens, and thus a violation of (iv) if the ratio of O to X exceeds 2.5. Indeed Hägg argued that the tendency to form a glass cannot rely on the development of a random network as Zachariasen suggested, citing the example of alkali-metaphosphate in which long chains of Q^2 species inhibit crystallization [40]. Nevertheless the general postulate that the same primitive structural units exist in crystals and glasses of the same composition is also true when adding alkali oxide. A description of the network now relies on the knowledge of the Q-species distribution and their connectivity. Information about both can be gained by nuclear magnetic resonance (NMR) experiments [41].

3.2 Origin of MD force fields for glass systems

A MD force field consists of two components: a mathematical function and a set of parameters in that function. Usually, "developing" a new MD force field is used syn-

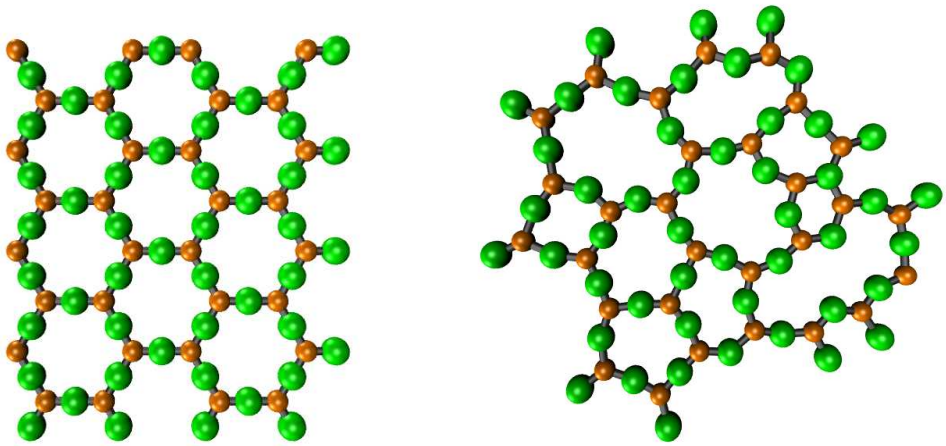


Figure 11: Schematic figure of a crystal (left) and a glass (right) created by using Zachariasens rules for a composition X_2O_3. Large green spheres are oxygen atoms, and small orange spheres are atoms of type X.

onymously with determining the parameters for the mathematical function. Before discussing the latter some considerations concerning the choice of the functional form shall be made.

A force field $U(r)$ (for a solid or liquid material) has to consist of at least two parts: an attractive force to keep the atoms together and a repulsive one to prevent atoms from overlapping. This will automatically provide a minimum at some distance r_e which is close to the equilibrium interatomic distance of that MD system. Typical choices for either part are power laws (most commonly r^{-12} and r^{-6}) and exponential functions. Their combinations already provide the three most often used pair-wise potential forms [42, 43, 44]:

- Lennard Jones : $U(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right]$
- Buckingham: $U(r) = Ae^{br} - \frac{C}{r^6}$
- Morse[15]: $U(r) = D\left(e^{-2a(r-r_e)} - 2e^{-a(r-r_e)}\right)$

A plot of those potentials is shown in Fig. 12 with parameter choices that result in the same equilibrium distances and binding energies. It is worth noting that by specifying depth and position of the minimum the Lennard Jones potential is fully determined. Both the Buckingham and the Morse potential have one more degree of freedom, which allows for an adjustment of the curvature at the minimum of the potential.

[15]Often a different form of the Morse potential is given by $V(r) = D\left(1 - e^{-a(r-r_e)}\right)^2$ which, except for a constant D, is equal to the form used here, i.e. $U(r) + D = V(r)$.

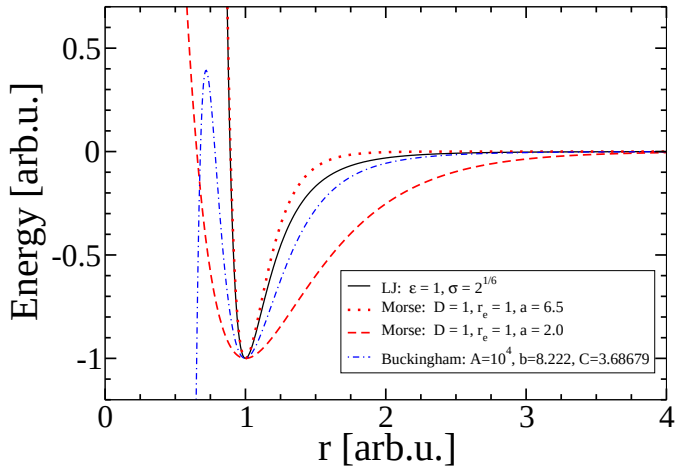


Figure 12: A Lennard Jones, Morse and Buckingham potential with a minimum at $r = 1$ and $E = 1$. The Buckingham potential has a singularity at $r = 0$.

Additionally, a term $U_{\mathrm{C}}(r) = \frac{q_i q_j}{4\pi\epsilon_0 r}$ for the coulomb interaction is generally used in systems with charged particles. It is worth noting that the Buckingham potential has a singularity at $r = 0$. While that is generally no problem if a system is close enough to equilibrium and the temperature is not too high, it can pose a challenge for generating glasses [16]. The latter typically involves heating the system to a few thousand Kelvin and cooling it down again. Sometimes an additional repulsive power law term is added to the Buckingham potential for these kind of simulations.

The next logical step in order to use more realistic potentials is the addition of angle potentials in structures with covalent bonds. In network glasses, they allow for a more realistic modeling of the bending modes of the units by specifying O-X-O angles (with X being the central atom, such as Si, B or P) and they provide the means to restrict the angles between two corner sharing units through specifying X-O-X angles. The most commonly used angle potential is a simple harmonic:

$$V_{\mathrm{A}}(\phi) = \frac{1}{2}K(\phi - \phi_e)^2, \tag{4}$$

where ϕ_e is the equilibrium angle and K the spring force constant. A more complex angle potential which is now increasingly used was proposed by Vessal *et al.* [45]:

$$V_{\mathrm{A}}(\phi) = K\left[\phi^n(\phi - \phi_0)^2(\phi + \phi_0 + 2\pi)^2 - \frac{n}{2\pi^{n-1}}(\phi - \phi_0)^2(\pi - \phi_0)^3\right] \tag{5}$$

[16] As long as the chosen parameters for a Buckingham potential generate a local minimum at a distance r_e, there is an energetic barrier which must be overcome for two atoms to be trapped in the singularity. The probability for one pair of atoms to overcome that barrier in a timespan t can be estimated by $t \cdot \nu \exp(\frac{E_b}{kT})$, where ν is an attempt frequency, E_b is the barrier height, k is the Boltzmann constant, and T is the temperature. Assuming ν to be 10^{12} Hz (the order of the fastest molecular vibrations) and $E_b = 5$ eV (roughly the binding energy of O_2) the probability to have such an event within a nanosecond is only $6 \cdot 10^{-23}$ at 1000 K. At 5000 K it is already one percent.

While more complex potentials are possible - non-fixed charges, polarizability and many body effects to include coordination dependency come to mind - those are rarely employed for the simulation of glasses. The reasons for that are manyfold. For one, many more parameters are needed for more difficult force fields, complicating every parameter fitting approach. Also, relatively long run times are needed to simulate glasses, since they must first be produced from the melt. And last but not least investigations into using more complex potentials failed to show substantial improvements over more simple force fields [46].

In the first MD simulations of glasses parameters were determined empirically. Often formal charges were used, and the minima of the potential were set so that they are in agreement with experimentally determined ion radii [47, 48, 49]. While these potentials already reproduced fundamental properties of glasses, now more sophisticated methods are used to develop reliable force fields.

There are two major approaches which are used (also in combination) to determine the parameters of a potential: (i) fitting data from quantum mechanical calculations and (ii) fitting experimental data (mainly elastic constants) in crystalline systems.

The most common approach to use quantum mechanical calculations for deriving potential parameters is to compute the energy and forces on a basic configuration found in the glass, and fit those with a classical force field ansatz [50, 51, 52, 53, 54, 55]. For example Tsuneyuki *et al.* calculated the total energy of a SiO_4^{4-} cluster with four positive point charges to mimic the surrounding bulk material [50]. Then they calculated the potential energy surface with respect to three vibrational modes: a symmetric breathing mode in which all four Si-O distances are varied together, the C_{3v} mode where only one of the Si-O distances is changed, and a D_{2d} bending mode in which O-Si-O angles are changed while the Si-O distance is kept constant. Since a subsequent fitting of the parameters for a Buckingham potential to these energy surfaces did not result in a unique solution, they performed MD calculations of crystalline silica to determine which of the parameter sets best reproduces the experimental data.

A very similar approach was taken by B. van Beest, G. Kramer, and R. van Santen who determined another Buckingham potential parameterization for silica [51]. But they argued that it is necessary to include bulk data into the fitting routine in addition to the quantum mechanical calculations, since they found that a number of good fits to the potential energy surface resulted in parameter sets which were not reproducing crystalline data. Also, they observed that even in a simple system, such as SiO_2, parameters are redundant, i.e. a change in one of them can be compensated by changing another. Nevertheless, their so-called BKS potential is one of the most widely used force fields to model silica.

Later, Habasaki *et al.* added alkali interactions to the potential of Tsuneyuki *et al.* by fitting the stretching mode of an e^+-O-Li-O-e^+ cluster modeled with Hartree Fock calculations.. Another approach to extend the potential of Tsuneyuki *et al.* was taken by W. G. Seo and F. Tsukihashi, who fitted additional parameters for sodium-oxygen interaction to structural, transport and thermodynamic data of calcium silicate crystals [56].

One possible criticism of this approach is the small size of the clusters for the quantum mechanical calculations. It is hard to imagine that the bulk material should not have a significant influence on the potential energy surface of a network former unit. After all, the four oxygen atoms of a SiO_4^{4-} tetrahedra are each part of a second SiO_4^{4-} unit as well. This argument weighs even more in a disordered material, where simply adding a positive charge on each oxygen is presumably an even less satisfying way to approximate the surrounding material.

An obvious way to address this criticism is to use bigger clusters, which was not possible when the aforementioned potentials were developed due to the constraints of available computational power at that time. Indeed, nowadays it is possible to produce larger and even "disordered" clusters of 50-100 atoms, as the author has shown in [57]. These clusters were validated by comparing experimental and calculated vibrational spectra, which were in good agreement.

Using those clusters we determined the energy surface along their vibrational modes in order to fit parameters of an empirical force field to it. While the fitting procedure succeeded, the resulting parameter sets were not able to reproduce a realistic glass structure when used for MD simulations. They even failed to produce correct coordination numbers for the network atoms.

A similar conclusion was made by Carre *et al.* [54]. They found that parameters obtained by fitting forces of short Car-Parinello quantum MD simulations [17] of liquid SiO_2 were not able to reproduce the structure of the liquid. Instead Carre *et al.* used the partial radial distribution functions of the quantum mechanical calculations as input for their fitting process. The resulting potential does reproduce some of the properties of the Car-Parinello system slightly better than the BKS potential, but it also shows a slightly larger disagreement with experimental data of crystalline phases.

The second approach for parameterizing force fields works identically to the first one, with the difference that the input data for the fitting process comes from experiments instead of *ab initio* calculations [59, 60, 45]. Most commonly elastic constants and lattice vectors are used as reference data. The reason for not using data from glass materials is twofold. Firstly, the experimental data on crystals is much more detailed and specific. Also, as non isotropic materials, crystals offer a richer feature set of basic mechanical properties. For example the elastic constants are tensors, whereas in isotropic glasses they are scalar values. Secondly, the fitting process is much simpler, because the MD structures are much more easily produced. While one can just use the correct crystal structure as a start configuration and relax it according to the atomic forces due to the guess potential, it is not as simple with glasses. It would be necessary to do a full cycle of equilibrating a melt, cooling it down, and equilibrating the glass, for each set of guess parameters. A cycle like that needs several orders of magnitude more simulations steps than the relaxation of a crystalline structure. In addition one would need to use

[17] The Car-Parinello method is a type of *ab-initio* MD simulation. In contrast to other *ab-initio* MD methods [58], the electrons are treated explicitely through additional degrees of freedom in the form of dynamical variables which are associated with a fictitious mass. In 2009 Roberto Car and Michelle Parrinello were awarded the Dirac Medal for the development of the method.

Table 2: Ion charges in elementary charges used for $x\mathrm{M_2O} - y2\mathrm{SiO_2} - z\mathrm{B_2O_3} - w\mathrm{P_2O_5}$

Li	O	Si	B	P
0.67	-1.34	2.68	2.01	3.35

relatively large systems (or many small) to have a good representation of the amorphous glass system. This is not the case with crystals, where a few atoms are enough to make a unit cell, which together with periodic boundary conditions is a good representation of a crystal.

3.3 Determination of potential parameters

The basic concept of the potential development approach employed here, is to use an existing successful force field and extend it through as little additions as possible. Junko Habasakis potential for alkali silicates [52] was chosen as a basis and force fields for borates and phosphates were developed, which are both compatible with each other and the silicate potential. The potential is of the Buckingham style using additional Coulomb interactions:

$$E(r_{ij}) = \frac{1}{4\pi\epsilon_0}\frac{q_i q_j}{r_{ij}} + A_{ij}e^{-r_{ij}/\sigma_{ij}} - \frac{c_{ij}}{r_{ij}^6} \tag{6}$$

where q_i and q_j are the charges of atoms i and j respectively, ϵ_0 is the vacuum permittivity, A_{ij}, σ_{ij}, and c_{ij} are type specific parameters and r_{ij} is the distance between atoms i and j. In the following the abbreviation BC-U (for Buckingham-Coulomb Unified) will be used for this new force field.

In order to allow for a more simple application of the potential to the different glass composition, the charges used by Habasaki are altered so that the basic building blocks of the glasses (e.g. $\mathrm{Li_2O}$ and $\mathrm{SiO_2}$) are charge neutral for themselves. This principle is also used to set the charges of boron and phosphorus. Note that after setting the charge for only one of the atom types, every other follows automatically. The particular choice of values was informed by weighting the tradeoff between increasing the oxygen-silicon interaction versus decreasing the lithium-oxygen interaction. In the end the latter was reduced by roughly 25% while the more important oxygen-silicon interaction was increased by 9% for the metasilicate composition. The values are given in Table 2.

For the borate and phosphate potentials the same alkali-alkali, alkali-oxygen and oxygen-oxygen interactions as in the alkali-silicate were used. Furthermore the same parameters as for the alkali-silicon interaction are used for alkali-boron and alkali-phosphorus and the silicon-silicon parameters are used for all interactions between silicon, boron and phosphorus atoms. This only leaves the boron-oxygen and the phosphorus-oxygen parameters to be determined. The rationale for this approach is that the interaction between the network atoms is the dominant factor in determining the properties of the glass. In fact, there are a number of force fields, where all interactions involving

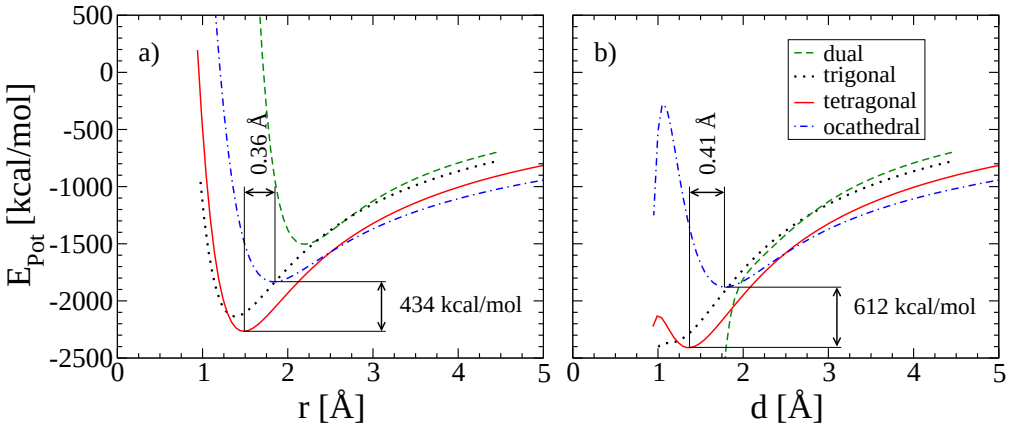


Figure 13: Binding energies of various hypothetical glass building blocks for a good parameter set (a) and one which lacks energy barriers towards low interatomic distances (b).

no oxygens are limited to Coulomb repulsion [59, 60].

The remaining parameters are determined by systematic variations. A first sanity check of a possible set of parameters is done by investigating the potential energy of symmetric XO_2, XO_3, XO_4, and XO_6 units in dependence of the X-O distance with X being boron or phosphorus. Considering that the Buckingham potential has a singularity at $r = 0$ Å, a good set of parameters should exhibit energy barriers at distances smaller than the typical bond lengths in order to prevent a network unit to collaps into a single point. Also distinct minima at about the targeted bond lengths are necessary. Note that due to interactions with the surrounding material the bond lengths of a network unit will generally be found at slightly different values than those minima. A third criterion, which can be checked for at this stage, is the relative stability of the various units versus each other. For example one would want that the energetic difference between BO_3 and BO_4 units is less pronounced than that of SiO_3 and SiO_4 units.

Figures 13 a) and b) show the potential energies of various network former units for the final set of parameters for the phosphate system and a set of parameters which lacks energy barriers towards low X-O distances.

If a candidate set passes this simple check a fast testrun is performed. A random starting configuration of about 2500 particles is first equilibrated at 1500 K and then cooled down to 400 K under constant pressure conditions (10,000 atm) with a cooling rate of 5 K/ps [18]. The resulting configuration is equilibrated for another 100 ps (with the pressure being reduced to zero) before determining some basic structural properties such as density, and the coordination numbers of the network atoms.

It turned out that it is beneficial, even necessary, to use angle potentials in addition to

[18] The systemsize and the choice of cooling rate is a compromise of choosing settings which are relatively close to the values employed in the final investigations and limiting the runtime. The temperature is chosen so that the system is in a liquid state at the beginning.

Table 3: Potential parameters for $x\mathrm{M_2O} - y2\mathrm{SiO_2} - z\mathrm{B_2O_3} - w\mathrm{P_2O_5}$. In the table X refers to Si, P and B. The same parameters are used for all interactions between atoms of those three types in any combination.

Pair	A_{ij}	σ_{ij}	c_{ij}
Li Li	$1.5478{\cdot}10^5$	0.14642	0.0
Li O	$5.5072{\cdot}10^4$	0.24887	764
Li X	$5514{\cdot}10^6$	0.10606	0.0
O O	$4.0514{\cdot}10^4$	0.35132	4952
O Si	$2.4724{\cdot}10^5$	0.20851	1631
O B	$2.5000{\cdot}10^6$	0.126	300.0
O P	$9.000{\cdot}10^4$	0.2035	0.0
X X	$2.0117{\cdot}10^{10}$	0.0657	0.0

the buckingham force field. For the borate system this fullfills the purpose of sharpening the otherwise very broad angle distributions.

For the phosphates, angle potentials are necessary in order to prevent over-coordination of the phosphorus atoms while keeping the equilibrium distances between phosphorus and oxygen atoms at the correct values. The underlying tendency for over-coordination (5x and 6x) of the phosphorus ions is due to the high number of non-bridging oxygens in the system, when only using tetrahedral units. Since non-bridging oxygens only exhibit one bond to a phosphorus atom, it is beneficial to transform it into a bridging one at the cost of increasing the coordination number of a phosphorus. Even though the PO_5 and PO_6 units are energetically less favorable than PO_4 units (as shown in Fig. 13), this can lead to a net gain in the binding energy. Adding an angle potential, which favors the PO_4 units, corrects this situation.

In order to avoid differentiating between O-B-O angles of BO_3 units (their equilibrium angle should be 120°) and BO_4 units (109.47°), it was decided to use a double minimum potential of the form

$$E(\phi) = K\left\{\left[1-\exp\left(-\sigma(\phi-109.47^\circ)^2\right)\right]\times \left[1-\exp\left(-\sigma(\phi-120^\circ)^2\right)\right]-1\right\} \tag{7}$$

for them. In the phosphate systems the potential for the O-P-O angles is

$$E(\phi) = K\left\{\left[1-\exp\left(-\sigma(\phi-120^\circ)^2\right)\right]-1\right\}. \tag{8}$$

K was chosen to be 10 kcal/mol and σ=0.1. The final set of parameters can be found in Table 3.

3.4 Sample preparation

The glasses are generated from random start configurations by constant pressure simulations with a cooling rate of 1 K/ps and an intermediate equilibration at 1100 K. The procedure can generally be divided into five stages:

- **Stage 1:** A large cubic box is filled with randomly positioned particles of the desired composition of a density which is 50 to 100 times lower than that of the target system. The low density prevents the ions from overlapping too much in the initial configuration. At the same time putting the particles at random positions avoids the possibility to carry over structural features from the starting configuration (for example a crystal) to the final glass system. In order to avoid problems with the singularity of the Buckingham potential, the random start configuration is relaxed using a simple Lennard Jones interaction model with $\sigma = 2.5$ Å and $\epsilon = 200$ kcal/mol (the same parameters are used for all interaction pairs). First the ensemble is run for 20 ps with a time-step of 1 fs using a restricted NVE[19]-integrator – i.e. the maximum movement distance of a particle per time-step is limited to 0.5 Å. Additionally the velocities are rescaled every time-step so that the temperature is 1,000 K. This is necessary, since due to the random starting configuration many atoms are extremely close to each other. As a result the initial total energy is typically positive and extremely large, so that the system is not in a bound state. Then, the system is propagated for another 2 ps using a time-step of 0.1 fs with an NPT-integrator at a constant target pressure of 10,000 atm and a temperature of 1,000 K. The chosen relaxation times for the temperature and the pressure were 0.1 ps and 1.0 ps respectively. In the cause of this short run under constant pressure the density is typically increased by an order of magnitude, but it is still well below the target density.

- **Stage 2:** The resulting configuration is taken as a starting point for a first quick relaxation (20 ps at 1,000 K with a restricted NVE-integrator) using the real pair interaction model and its parameters. This includes the activation of Coulomb interactions. Angle potentials are not yet used at this stage, which is carried out under constant volume conditions. Then the density is adjusted again with a 200 ps long run under constant pressure of 10,000 atm at 1,000 K. Both Stage 1, and Stage 2 are used to eliminate non-physicaly situations in the system, coming from the random start configuration. This includes too small interatomic distances as well as clusters of equally charged particles.

- **Stage 3:** Before beginning to cool down the system, it is equilibrated at 1,500 K for 200 ps at a pressure of 10,000 atm. At this point the angle potential is activated. Which atoms form an angle is determined every 500 time-steps according to a

[19] In molecular dynamics the integrator types are often referred to by the conserved properties. So NVE means an integrator which conserves the number of particles, the volume and the total energy – i.e. a micro canonic ensemble is simulated. Other typical chocies are NVT and NPT, where the number of particles, the temperature and the volume, respectively the pressure, are conserved.

simple distance criterion. As the cutoff distance for atoms to be close enough to form an angle, 1.9 Å was used. At the end of this stage the system is in a physicaly meaningful state for the first time[20].

- **Stage 4:** The actual cooling process consists of three phases. First the system is cooled down from 1,500 K to 1,100 K with a rate of 1 K/ps. During this period the pressure is kept at 10,000 atm. Then an intermediate equilibration of 100 ps is performed using the NPT-integrator in order to reduce the pressure to 1,000 atm, followed by 1,000 ps with the NVT-integrator. In the end the system is again cooled down from 1,100 K to 400 K with a cooling rate of 1 K/ps at a constant pressure of 1,000 atm. Every 100 ps (every 100 K) a restart configuration is saved.
- **Stage 5:** A final equilibration at the target temperature consists of a 200 ps run for reducing the pressure to 0 atm, a 200 ps run with a constant volume using the NVT-integrator and 600 ps in the NVE-ensemble. If angle potentials are used a NVT integrator is used for the last 600 ps also, since the dynamic creation and destruction of angles violates energy conservation.

The final structures are used for the determination of structural properties, and serve as a starting point for the determination of diffusion constants. During the latter simulations the momentum of the system is kept explicitely at zero. Otherwise small errors can lead to significant contributions to the mean square displacement, in particular at low alkali ion concentrations and low temperatures. Generally, the list of angles is rebuild every 0.5 ps.

For systems where the density has to be set to the correct value , the equilibrated states at 400 K of the constant pressure runs are taken and scaled to the correct density. These scaled configurations are then used as a starting point for stages 3 to 5, whith all NPT runs being replaced with NVT runs.

3.5 Evaluation procedure

In order to have a (simple) method for comparing the quality of different potential models for ion-conducting glasses, a set of evaluation criteria is chosen. Depending on the grade of agreement with experimental data, either zero, one or two points are awarded. A potential earns two points if it generates systems in good agreement with experimental values (trends are correct and actual values are close to experimental ones), one point if the fundamentals are reproduced (e.g. values are within the range typically measured for such systems even though the values of the particular glass composition are not matched) and zero if it is unrealistic. Two levels of criteria are distinguished, with primary features contributing to the overall score with a triple weight.

While a more elaborate scheme is possible, there are a number of reasons to keep it simple. First, the main goal of awarding something equivalent to grades to each potential is to express the “gut feeling”, which most researchers on this particular field

[20]Physical meaningfull not necessaraly implies that the system is a good representation of any real material .- but it is in an equilibrium state with respect to the employed interaction model

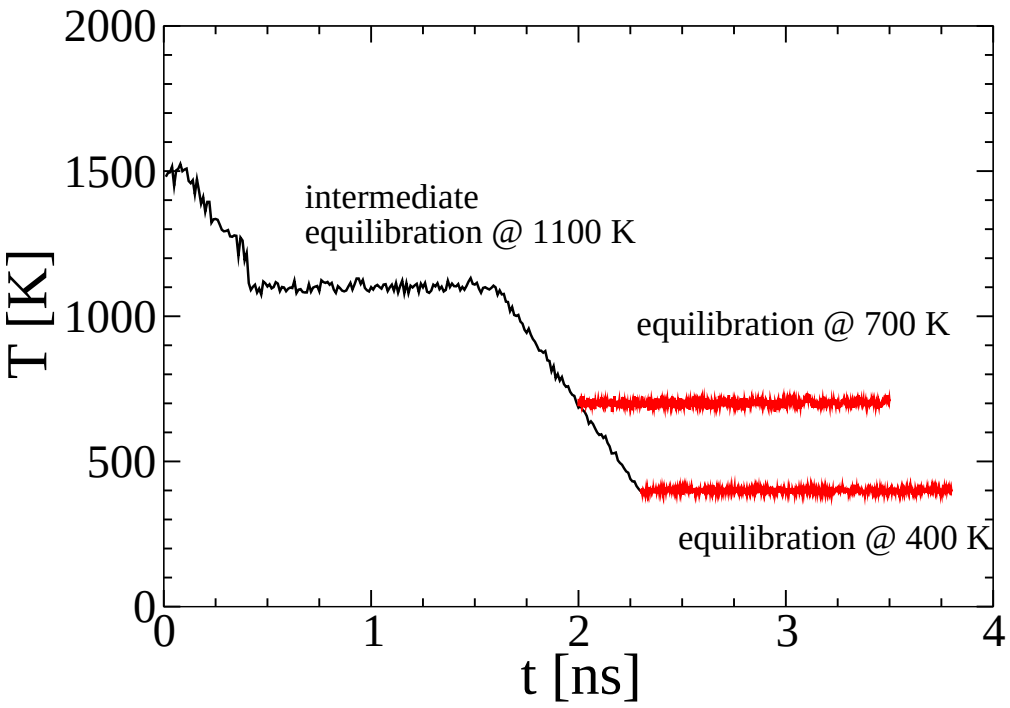


Figure 14: Cooling cycle for the preparation of the glass systems. The cooling rate is 1 K/ps. At 1100 K an intermediate equilibration takes place, and at each target temperature another equilibration is performed.

have, in numbers. Secondly, many experimental results are not as definite as it might seem at first glance. The problem is not so much that a single experiment does not reproduce exactly the same values again upon repitition, but that glass properties are sensitive to their processing history. This can lead to seemingly impossible deviations between experimental values reported by different groups. An example is the density of lithium metaphosphate, where reported values range from 2.29 g/cm^3 to 2.56 g/cm^3 [61]- [99]. In the case of mixed network former glasses, it is also difficult at best to find a complete set of experimental values for a force field evaluation, because the range of possible compositions is so large. Often only data of similar – not equal – compositions exists, a fact that renders more elaborate evaluation procedures meaningless. Additionally, the process-history of MD systems differs substantially from that of real systems. Considering that simulating a single nanosecond can take a few hours to a few days, it is practically not possible to anneal a glass for several hours of simulated time - as it is routinely done when preparing real glasses. Also, cooling rates are several orders of magnitude larger in MD simulations than in experiments. As a consequence, different properties can be expected even if a potential is a good representation of the actual physics governing glasses.

Taking the weighting factors into account, a total of 28 points can be achieved over

Table 4: Quality criteria for MD potentials for ion-conducting glasses. Two importance levels are distinguished with different weighting factors (3 and 1). In each criterion a potential can earn two points. As subcategories structural properties (s) and features which are associated with the particle dynamics (d) are distinguished.

Level	Feature	Points
Primary	Coordination numbers of network atoms	2 (s)
	Decoupling of mobile ions from the glass network	2 (d)
Secondary	Density	2 (s)
	Bond lengths	2 (s)
	Compressibility	2
	Thermal expansion coefficient	2
	RDF / Structurefactor	2 (s)
	Conductivity	2 (d)
	Activation Energy	2 (d)
	Vibrational Spectra	2 (d)
	Structural	12
	Dynamical	12
	Total	28

all criteria, with 16 attributed to structural properties and 12 to features of the particle dynamics. While a simple number, as the sum of these points, cannot give a complete picture of how well a given potential is suited for a specific kind of investigation, it does give a general idea of how well behaved the potential is.

- **The coordination number** of an atom is the number of neighboring atoms in the first neighbor shell. In MD simulations coordination numbers are readily defined through simple distance criterion based on the first minima in the type specific pair distribution functions. Experimental data for the network atoms, gained from Nuclear Magnetic Resonance (NMR) experiments, is also available for many glass compositions. The coordination numbers of the network atoms are the dominant factor in determing (or from another point of view: the most defining feature of) the glass network structure. In Reverse-Monte-Carlo simulations it was shown that these coordination numbers, together with minimal inter-atomic distances, produce realistic structures with a substantial agreement of radial distribution functions [38]. Hence, coordination numbers serve as one of the main benchmarks for the quality of MD potentials in producing glass structures.

- **Decoupling** refers to the separate time-scales on which diffusion of mobile ions

and the network atoms occurs. In order to investigate transport processes it is necessary that the glass network is stable at least on the time-scale where the mobile ions reach the diffusive regime. In real glass systems this is indeed the case, as apparent from the fact that the mobile ions have diffusion constants many orders of magnitude larger than those of the network atoms. It will be shown later, that this is not the case in all computer glass models.

- **The density** is one of the most fundamental properties of a material, nevertheless many potentials for network glasses do not reproduce those on their own. Often MD simulations are performed with fixed volume settings, where the density is arbitrarily set to match the experimental one. This leads to unrealistic high pressure in the order of several ten thousand atmospheres.

- **Bond lengths** of the network atoms are another defining microscopic feature of the structure of network glasses. Experimental data is often available from neutron and X-ray scattering experiments.

- **Compressibility and the thermal expansion coefficient** are the rates of volume change with varying pressure and temperature respectively.

- **The radial distribution function** (RDF, also known as $G(r)$) is the particle – particle pair correlation function. Through a fourier transformation it is connected to the structure factor, which can be determined from defraction experiments. In Appendix E the necessary formulas for calculating the structure factor of a MD system are given.

- **The conductivity** is one of the features most often measured in experiments, since the actual applications of ion-conducting glasses rely on it. In MD simulations the conductivity can be calculated from the slope s of the mean square displacement (MSD) of the mobile ions. It is related to the diffusion constant $D = s/6$. Using the Nernst-Einstein equation this allows for the calculation of the conductivity with:

$$\sigma = \frac{nq^2}{kT} D, \tag{9}$$

  where n is the number density of lithium ions, q is their charge (we assume $q = 1$ e), k is the Boltzmann constant and T is the temperature.

- **Conductivity activation energies (E_a)** describe the change of the conductivity σ with temperature T. For ion-conducting glasses it follows an Arrhenius law $\sigma = \sigma_0 e^{\frac{E_a}{k_B T}}$.

- **Vibrational spectra** can be calculated from the velocity auto correlation function. Experimental data is readily available for many glass compositions from Raman and infrared measurements.

The glass transition temperature T_g is not taken into account as a criterion by purpose. Generally, T_g is a function of the cooling rate. Since MD glasses are prepared with extremely high cooling rates in the order of 10^{12} K/s as compared to 1 K/s in experiments, it is questionable if a comparison of experimental values with calculated ones is meaningfull. One might even argue that an MD-glass with T_g close to experimental values is showing wrong physical behavior.

In order to model mixed network former glasses it is not enough to have reliable force fields for the sub systems, since those are generally not compatible. Even if the potential type is the same, the parameters usually are not, resulting in conflicting definitions of various interactions such as those between oxygen atoms. Furthermore most of the available force fields work with fixed density simulations at extremely high pressures in the range of 10^4 to 10^5 atm. In a very practical sense that external pressure is another parameter of the empirical force field. It not only sets the correct density, but can also have a large influence on the structural features. For example the ratio of tetragonal BO_4 units to trigonal BO_3 units very much depends on the external pressure in the Li_2O-B_2O_3 system using the potential model of Verhoeff and den Hartog [49] (more details in section 3.6.2). It is not clear how such an indirect parameter can be used in mixed network former glass systems.

In the following sections a number of selected force fields from the literature will be compared to the new BC-U force field. For this purpose most of the aforementioned properties will be determined. To the best knowledge of the author, there is no other MD study of ion-conducting glasses, which provides a similar comprehensive overview in terms of investigated features and compositions.

3.6 Results for binary network former glasses

3.6.1 Lithium Silicates

The first system which will be evaluated is the lithium silicate glass. Besides sodium silicates, it is the most well studied ion-conducting glass system by MD simulations [32, 33, 35, 38, 45, 52, 101, 60, 102, 103, 104, 105, 106, 107]. Consequently, various force fields can be found in the literature, where many properties have been determined and compared to experimental values. Here, the new BC-U potential will only be compared with a force field developed by Pedone *et al.* [60], in order to put more focus on the less explored borates and phosphates. It will be abbreviated MC-P. The Pedone system was chosen as a comparison, since it deviates from the usual Buckingham potential style by using a Morse potential. It was also parameterized for phosphates, and thus potentially allows the modeling of phosphosilicates.

The density of lithium silicate is better reproduced with the MC-P potential. As can be seen in Fig. 15 the BC-U force field overestimates it significantly, in particular at low lithium concentrations. Also, both force fields do not show the same trend as the experimental data. While the latter increases with increasing lithium content, the MD systems behave the other way around.

This differing behavior is also present for the average Si-O bond lengths, although

in a counterintuitive way. While both MD systems show a decreasing bond length when proceeding from $x = 0.0$ to $x = 1.0$, experimental data suggests an increase of the bond length [108].

One could speculate that the reason for the deviations of the MD systems from experimental observations are some fundamentally wrong structures, but both potentials generate systems which almost exclusively consist of SiO_4 tetrahedra like the real lithium silicate. The MC-P potential delivers slightly better results, with more than 99.5% of all silicon atoms being coordinated with four oxygens. When using the BC-U force field, up to 5% of all silicon atoms are coordinated with five instead of four oxygens for $x = 0.5$.

Nevertheless, an excellent agreement of the experimental and calculated structurefactor for $x = 1.0$ is achieved with both potentials. Note though, that at $x = 1.0$ the densities of the MD systems are relatively close to the experimental value. For small lithium concentrations the deviations are larger, and hence one has to expect larger differences of the diffraction data as well.

The thermal expansion coefficients, shown in Fig. 19, are better reproduced by the BC-U potential. While it matches the experimental values [100] almost perfectly, the MC-P system shows expansion coefficients, which are at least twice as large as the experimental values.

Both force fields reproduce the experimental observation of increasing conductivity with increasing lithium content (see Fig. 21). They also show very good decoupling of the mobile ions from the glass network, with diffusion constant ratios being larger than 1,000. Nevertheless, substantial differences in the conductivity are observed, with the MC-P system exhibiting more than an order of magnitude higher conductivity then the BC-U system. This is also reflected in the activation energies. The MC-P potential shows activation energies which are 0.2 eV lower than the experimental values. The activation energies observed with the BC-U potential are only too low by 0.1 eV.

Summarizing, both the MC-P and the BC-U potentials reproduce the experimental data well. Both potentials are well suited to investigate structural, as well as dynamic behavior. Using the evaluation scheme, the BC-U potential and the MC-P force field achieve 22 and 20 points respectively, out of 26 possible points for the features investigated here. A detailed list of the assigned points can be found in Sec. 3.8.

$x\mathbf{Li_2O}$ - $\mathbf{Si_2O_4}$

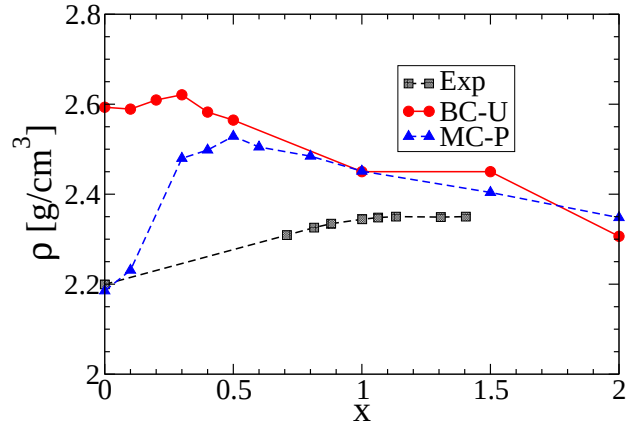


Figure 15: Density at ambient pressure. Experimental data is taken from [100].

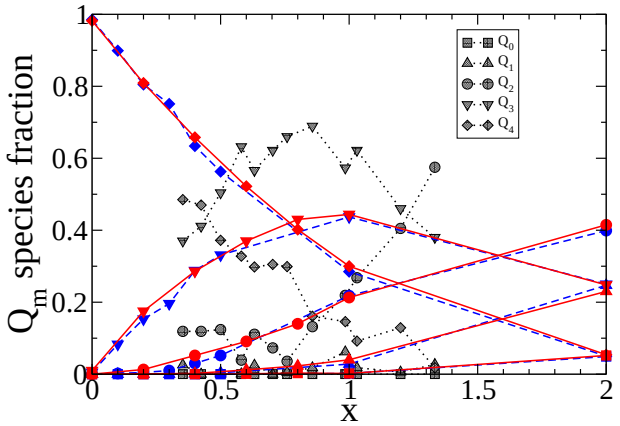


Figure 16: Q-Species distribution. Red symbols are from BC-U systems, blue symbols from MC-P systems and black symbols are experimental data from [109].

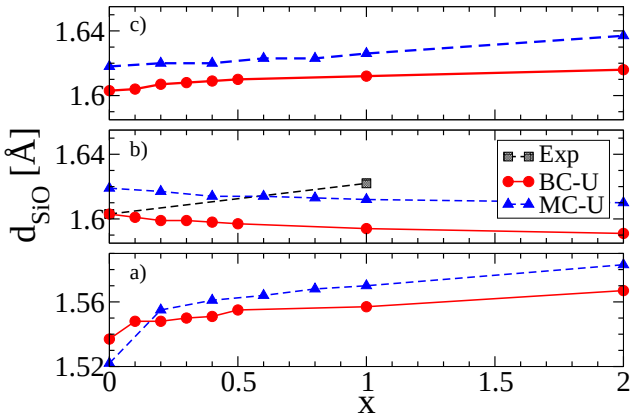


Figure 17: Si-O bond lengths for bridging oxygens (a), the average (b), and non bridging oxygens (c). Experimental values were found in [108].

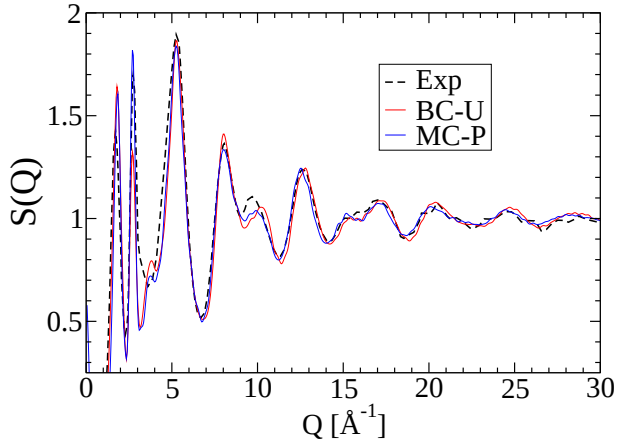


Figure 18: Calculated neutron scattering structurefactor of Li_2O - Si_2O_4. Experimental curves are reproduced from [110].

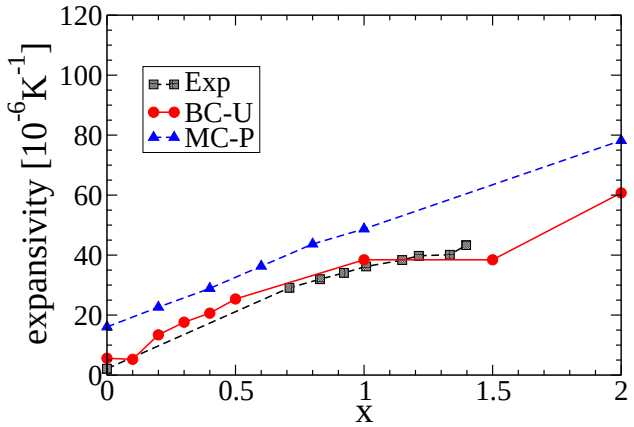


Figure 19: Thermal volume expansivity at ambient pressure. Experimental data was taken from [100].

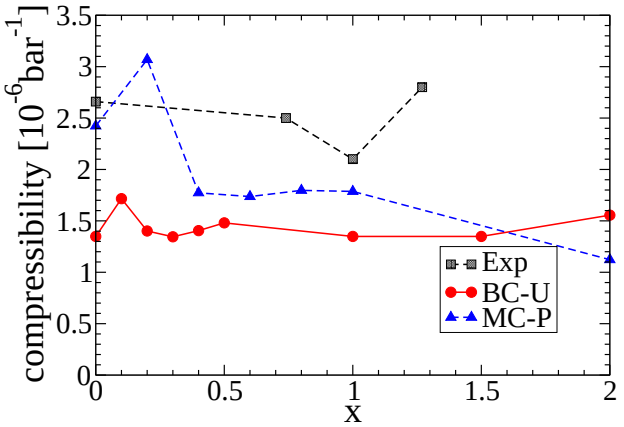


Figure 20: Volume compressibility (bulk modulus) as determined from MD simulations and experimental data [112].

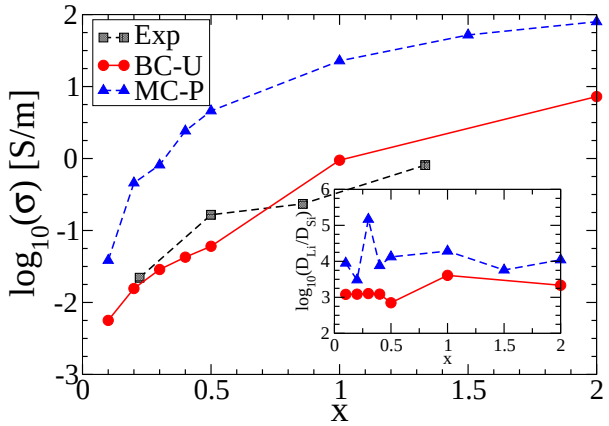


Figure 21: Ionic conductivity at 700 K. Experimental data calculated from [113]. The inset shows the ratio of lithium and silicon diffusion constants.

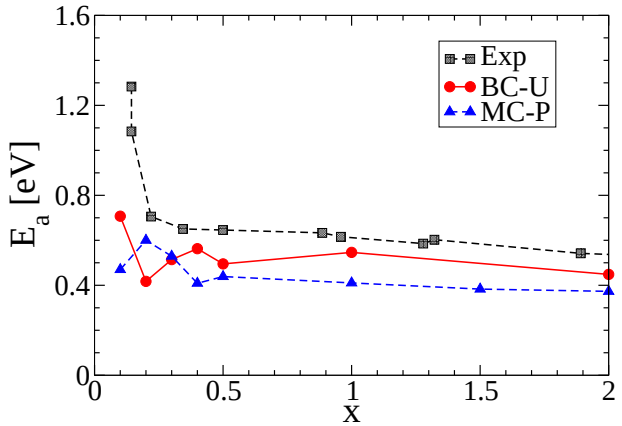


Figure 22: Activation energy in eV. The experimental values are taken from [113, 114].

3.6.2 Lithium Borates

The borate system $x\mathrm{Li_2O}$ - $\mathrm{B_2O_3}$[21] is a special case of network glasses since it exhibits two different geometries for the network former units: trigonal $\mathrm{BO_3}$ units and tetragonal $\mathrm{BO_4}$ units. As such it poses a particular challenge for modeling it with simple spherical pair potentials. The most commonly used force field for MD simulations of this system, was parameterized by Verhoeff and den Hartog [49], in the following referred to as BC-VH. It is also a Buckingham type potential (see Eq. 6) and uses additional angle terms for O-B-O triplets in order to stabilize the trigonal and tetragonal network former units. The employed angle potentials are of the form:

$$E(\phi) = K(\phi - \phi_0)^2 \tag{10}$$

where $K = 119.4$ kcal/mol and $\phi_0 = 120°$ for trigonal $\mathrm{BO_3}$ units and $K = 59.7$ kcal/mol and $\phi_0 = 109.47°$ for tetragonal $\mathrm{BO_4}$ units.

This potential is known for showing good agreement with experimental NMR data on coordination numbers of the network atoms [117]. But there are also considerable shortcomings, as will be made clear by the following data.

A first problem is that it can only be used with fixed density simulations. In Fig. 23 the densities of lithium borate systems with varying lithium content are shown for the BC-VH potential, as well as for the new BC-U force field. While the latter reproduces the experimental data [112] very well, the systems generated with the BC-VH potential exhibit densities which are too low by far. This is a common problem of many force fields for network glasses. Indeed Takada *et al.*, who developed three force fields for the pure borate system, concluded that their potentials produce reasonable densities in contrast to most published results [53]. Their densities are between 1.15 g/cm^3 and 1.63 g/cm^3, compared to a value of 1.89 g/cm^3 for systems simulated with the BC-U potential and an experimental value of 1.84 g/cm^3 [112].

As a consequence, the coordination numbers are substantially wrong as well, when performing constant pressure simulations with the BC-VH potential (see Fig. 24). Setting the density to the correct experimental values solves this problem so that the numbers are the same as what is found with the BC-U force field and what was reported in the literature [115]. It is worth noting that virtually no defects (i. e. boron atoms with coordination numbers other than three and four) occurred. Lithium coordination numbers are in good agreement with experimental values as well. Using 2.6 Å as cutoff distance, the mean coordination number is 4.98 with the BC-U potential and 4.57 with the BC-VH potential for $x = 0.5$. The experimental value reported in [118] is 4.9. In subsequent analysis only constant volume simulations have been used for the BC-VH borate systems. Compressibilities for the BC-VH potential have been determined by using the equilibrium pressures at the correct densities as base values. Thermal expansion coefficient have not been determined for the BC-VH potential.

Both potentials reproduce B-O bond lengths reasonably well (see Fig. 25). Nevertheless, the BC-U potential performs slightly better. Not only does it reproduce the B-O

[21] The bulk glass forming range of this system is $0.0 \leq x \leq 0.39$ [111]. While it is possible to extend that in certain cases considerably (e.g. thin film glasses), here investigations were limited to $x \leq 0.5$.

distance in BO_3 units more closely, but it also matches the experimental difference of B-O distances in BO_3 and BO_4 units of about 0.08 Å [119]. The latter is roughly 0.14 Å when using the BC-VH potential. Lithium-oxygen distances (not shown) however, are overestimated with both force field models. While experimental data suggests 2.0 Å as mean Li-O distance ($x = 0.5$, [118]), here a value of 2.20 Å was found with the BC-U and 2.12 Å with the BC-VH potential.

It is noteworthy that the experimental structurefactor S(Q) [120] is significantly better reproduced by the BC-VH potential. The more pronounced peaks of the curve for the BC-U force field indicate that the latter produces systems, which are slightly more ordered than those simulated with the BC-VH potential.

Seeing Fig. 29, it is evident that both MD simulations overestimate the conductivity at low lithium concentrations. Still, the BC-U force field fares considerably better than the BC-VH potential. The most striking difference is the behavior at high values of x. While the BC-U potential reproduces the fundamental feature of increasing conductivity with increasing lithium content (and does give the correct conductivity for $x = 0.5$), the conductivity decreases at $x = 0.5$ using the BC-VH potential. From the inset of Fig. 29 another advantage of the BC-U force field becomes apparent. The inset shows the logarithm of the ratio of the lithium diffusion constants to the boron diffusion constants. When using the BC-VH potential, there is just slightly more than an order of magnitude difference in the diffusion of the mobile lithium ions and the network atoms. As a consequence, the glass network is not stable on the timescale, which the lithium ions need to reach the diffusive regime. The BC-U potential achieves a decoupling of two to four orders of magnitude, which is generally enough to guarantee a stable network while investigating transport phenomena. It is notable that the diffusion constants of boron are extremely small when using the BC-U. Even after 100 ns, the increase of the MSD is typically only 0.1 $Å^2$. It is most likely that the real diffusion constants of boron are overestimated by several orders of magnitude.

A similar problem exists for the lithium diffusion constants for $x = 0.1$ and $x = 0.2$. Considering the relatively small total MSD of about 8 Å and 15 Å respectively, these values should be taken as estimates at best. It is very likely that the estimated diffusion constants are considerably too large. This assumption is supported by earlier findings of Heuer *et al.*, that backward jump dynamics decrease the slope of the MSD curve over long timescales [35]. Activation energies are shown in Fig. 30. At lithium concentrations of $x \leqslant 0.3$, the BC-U potential is in good agreement with the experimental data taken from [114]. At lower lithium concentrations the MD simulation exhibits too low activation energies. This is coherent with the observation that the conductivities determined from the MSD is too large for those systems. It can be assumed that the overestimation of the conductivity is larger at lower alkali contents and lower temperatures (an assumption again supported by the findings of Heuer *et al.* [35]). This would lead to the observed behavior that activation energies are too low for small x.

Summarizing, the BC-U potential achieves 23 of 26 points on the evaluation scale, while the BC-VH force field achieves 12 points. The low score is mostly due to dynamic properties, where the BC-VH potential earns only 2 out of 10 possible points.

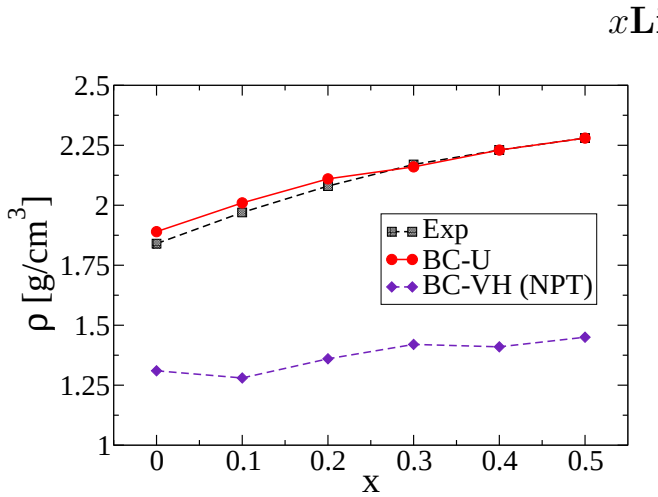


Figure 23: Densities at ambient pressure. Experimental data was taken from [112].

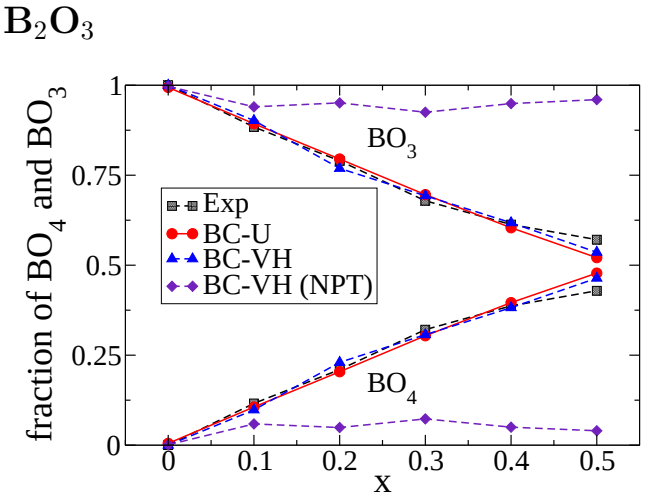


Figure 24: Fraction of BO_3 and BO_4 units as determined from MD simulations and NMR measurements [115].

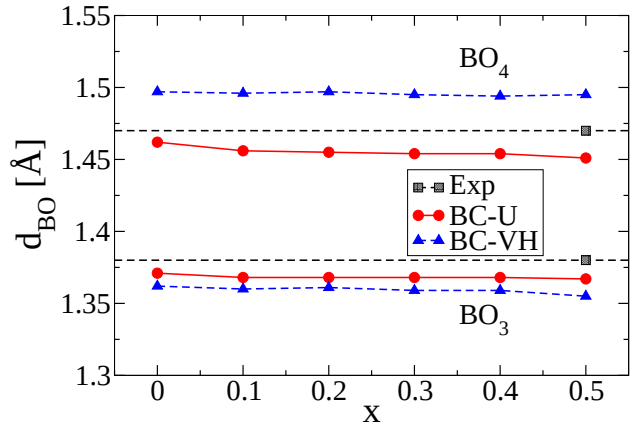


Figure 25: Bond lengths of oxygens in BO_3 (upper curves) and BO_4 units (lower curves). Experimental values are taken from [119] and were measured for $x = 0.5$.

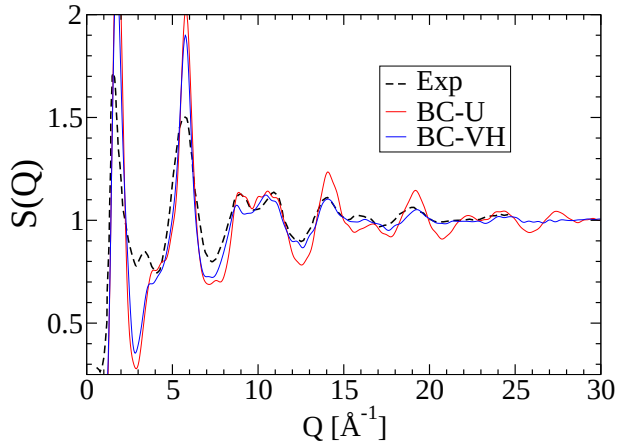


Figure 26: Neutron diffraction structurefactor. The experimental curve was taken from [120].

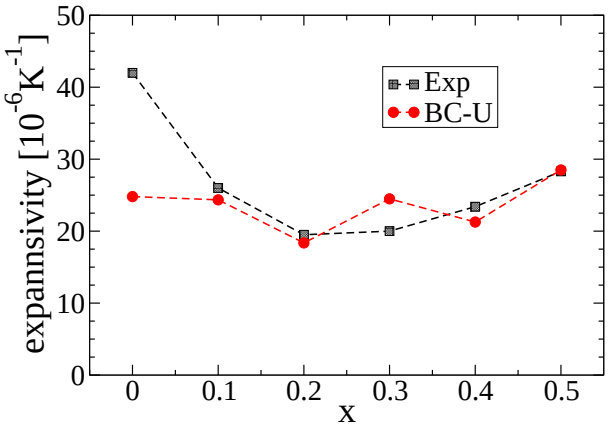


Figure 27: Thermal volume expansivity at ambient pressure. Experimental data was taken from [122].

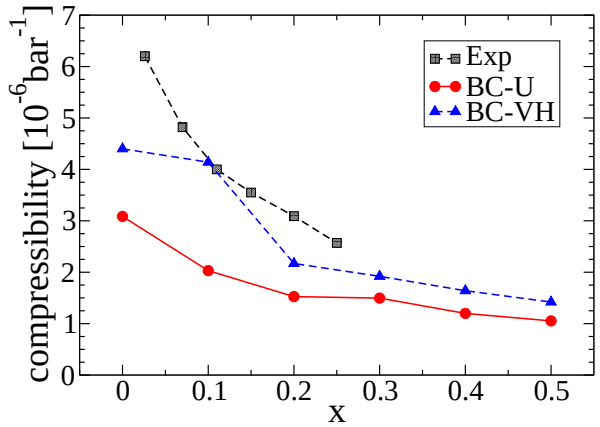


Figure 28: Volume compressibility (bulk modulus). Experimental data was found in [121].

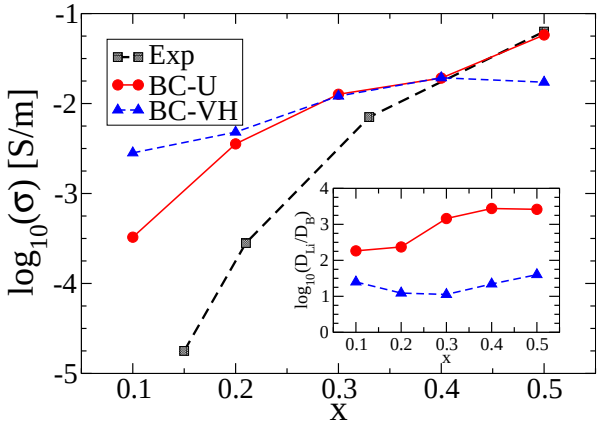


Figure 29: Conductivity at 700 K. The experimental values are extrapolated from Fig. 1 of [114]. The inset shows the quotient of lithium and boron diffusion constants.

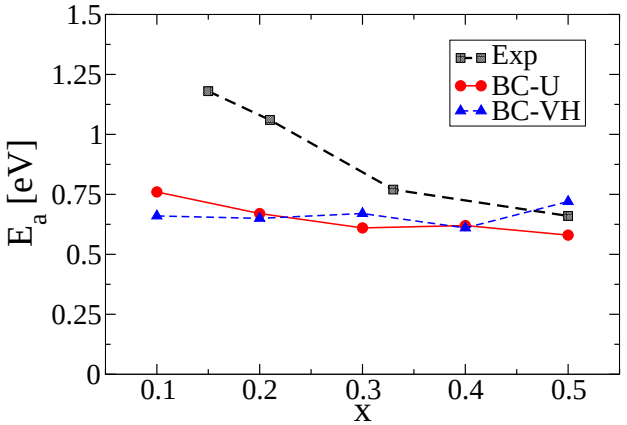


Figure 30: Activation energy in eV. The experimental values are taken from [114].

3.6.3 Lithium Phosphates

For the phosphate system $x\mathrm{Li_2O}$ - $\mathrm{P_2O_5}$, three published potentials are considered in addition to the BC-U force field. One is based on a potential of Karthikeyan *et al.* which was later modified by M. Vogel in order to optimize lithium-oxygen distances [123, 124]. It is another Buckingham type potential and uses no additional angle terms. The second force field is a Lennard-Jones potential with Coulomb interactions and terms for O-P-O as well as P-O-P angles. It was parameterized by Liang *et al.* in 2000 [59]. The third potential is the Morse-type force field parameterized by Pedone *et al.* which was already used as one of the references for the silicate systems [60]. In the following the three potentials will be referred to as BC-V, LJC-L and MC-P respectively.

For an overview of experimental data concerning structural properties of phosphate glasses, refer to the review article of Brow [116] A short summary of that review will be given considering the properties which were investigated in this study.

The general model for alkali phosphates is that it consists of cornersharing $\mathrm{PO_4}$ tetrahedra with alkali ions in the empty spaces in between. One classifies $\mathrm{PO_4}$ tetrahedra as Q^n species, where n is the number of bridging oxygens. In the pure v-$\mathrm{P_2O_5}$ system, every $\mathrm{PO_4}$ tetrahedra belongs to the Q^3 species, in which the non-bridging oxygen has a double bond to the phosphorous atom. With increasing lithium content the number of non-bridging oxygens increases, so that firstly Q^2 species and later Q^1 and Q^0 species occur. Evidence suggests that the non-bridging oxygens are indistinguishable from each other, i.e. that there is no difference between a double bonded oxygen and an O^- non-bridging oxygen [116].

With deviations of less than 10% from experimental values, densities are too small but in a good range for both the LJC-L and the BC-U potential. The LJC-L force field generates systems which deviate more from expected values (see Fig. 31). Both potentials also generate metaphosphate systems ($x = 1.0$) with a density at the lower end of reported experimental values.

Systems produced by using the BC-V potential exhibit much too small densities, with values dropping as low as 1.25 g/cm^3 for the metaphosphate system. As a consequence, subsequent analysis of BC-V systems were performed using fixed density simulations. The MC-P force field generates systems with a density which is generally too large. An analysis of the coordination numbers of the network former units reveals that this can be attributed to a substantial amount of over-coordinated phosphorous atoms - i.e. up to 28 % of all phosphorous atoms have five or six oxygens within a distance of 1.9 Å in the vitreous phosphate. The fraction of over-coordinated phosphorous atoms declines with increasing lithium content. For the metaphosphate it is only 4.2%.

In contrast, no $\mathrm{PO_5}$ and $\mathrm{PO_6}$ groups can be found when using the other potentials. Virtually all phosphorous atoms are coordinated with 4 oxygen atoms in systems generated with the LJC-L ($\geq 99.3\%$) and the BC-U ($\geq 99.9\%$) force field. Systems prepared by using the BC-V potential show a significant amount of $\mathrm{PO_3}$ units with up to 10% for $x = 0.1$. Again, the amount of $\mathrm{PO_3}$ units decreases with increasing lithium content, reaching 2.1% for the orthophosphate system, which coincidentally is the most often employed system in MD studies using the BC-V potential.

Since the fundamental structure of lithium phosphate is not reproduced at ambient pressure with the BC-V and MC-P potential, it was decided to set the density to the correct value by adjusting the volume. While this increases the fraction of PO_4 units only slightly for the BC-V systems (no gains for high alkali content, up to 2.5% for low alkali content), it significantly improves the situation for the MC-P systems. For $x = 0.0$ the PO_4 fraction increases from 72% to 94%. The largest deviations are now found for $x = 0.1$ with 89%.

Determing the Q^n species distribution of our simulations reveals that they are generally broader (see Fig. 39). For example only 70% of all phosphate units of the v-P_2O_5 MD system prepared with the new BC-U potential are Q^3 units, with the rest being divided between Q^4 and Q^2 units. Generally, the LJC-L potential creates the systems which are closest to the experimental data, with the BC-U force field being the second best. In addition to the occurrence of PO_3 units the BC-V potential generates Q-species distributions which are the broadest among the systems simulated here.

As a consequence of the deviation in Q-species distribution from experimental values, properties such as ring-size and chain-length distributions are most likely not correct either.

Bond lengths between oxygen and phosphorous atoms show the expected behavior for each of the potentials: non-bridging oxygens have a significantly smaller bond length than bridging oxygens, and those bond lengths increase with increasing lithium constant (see Fig. 8 in the review of Brow [116]). Nevertheless there are differences between the various systems as is shown in Fig. 33. Considering an experimental value of $\sim$1.52 Å for the average P-O distance, it is evident that the BC-V potential exhibits too large bond lengths. In contrast both the LJC-L and the BC-U force field show good agreement with that experimental value. While the average value determined from systems generated with the MC-P potential is close to the expected value as well, the values for the non-bridging and the bridging oxygens are too high[22]. The latter even decreases with increasing lithium content. This can be attributed to the over-coordination of phosphorous, which is more pronounced at small lithium concentrations.

Conductivities, as determined from the slope of MSD plots, are in good agreement with experimental data for the BC-V, the MC-P and the new BC-U potential. Also, for these three potentials, the decoupling factors are large enough to guarantee stable networks on timescales, where the lithium ions reach the diffusive regime. The LJC-L force field on the other hand shows much too large conductivities. Indeed in Fig. 37, the conductivity at 500 K was plotted for the LJC-L potential, while the other curves are 700 K data. Despite that 200 K difference the conductivity of the LJC-L force-field is more than order of magnitude larger.

This observation has its expression in the activation energies too (see Fig. 37). While the BC-V, the MC-P and the new BC-U potential have too small activation energies as well (with the new BC-U potential having the least deviations from the experimental values), their differences are much smaller than that of the LJC-L potential.

[22]Both, non-bridging and bridging oxygens can have to high bond lengths despite the correct average value, because more bridging oxygens exist due to the over-coordination.

$x\mathbf{Li_2O}$ - $\mathbf{P_2O_5}$

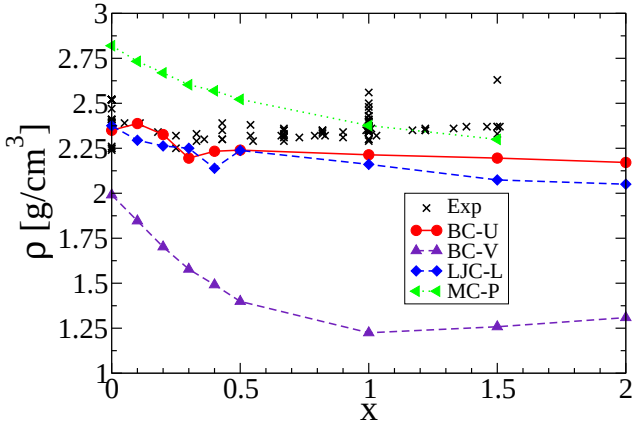


Figure 31: Density $\rho(x)$ at ambient pressurefor. Experimental values were taken from [61]- [99].

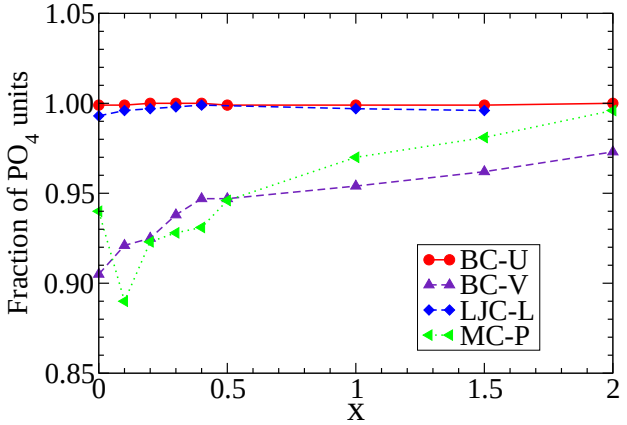


Figure 32: Fraction of PO_4 units. In real phosphate glasses all phosphorous atoms are coordinated with four oxygens [116].

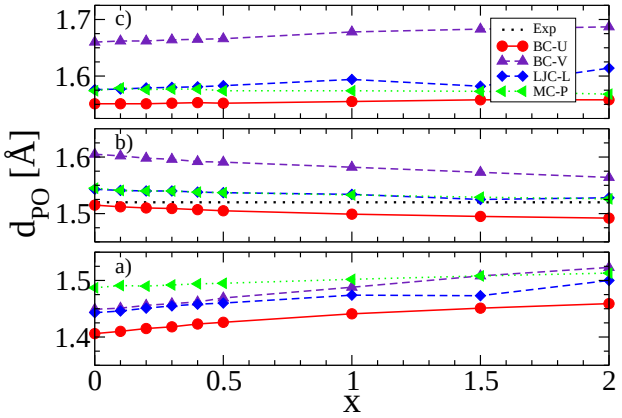


Figure 33: Average P-O bond lengths for bridging oxygens (a), the average (b), and non-bridging oxygens. Experiments show mean values of 1.52 Å [116, 126].

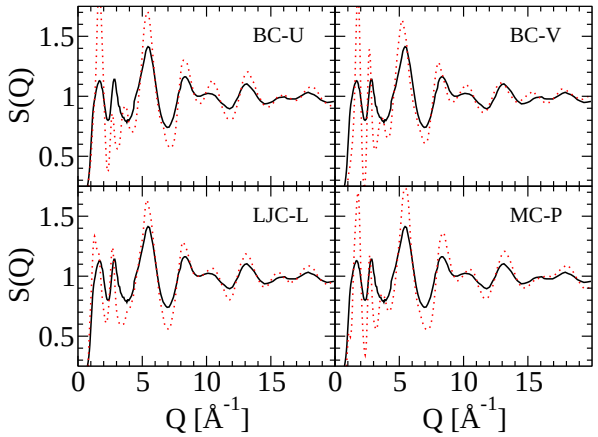


Figure 34: Neutron structure factor of $1.0Li_2O$-P_2O_5. Dotted red lines were determined from the MD simulations. The experimental solid black line was taken from [127].

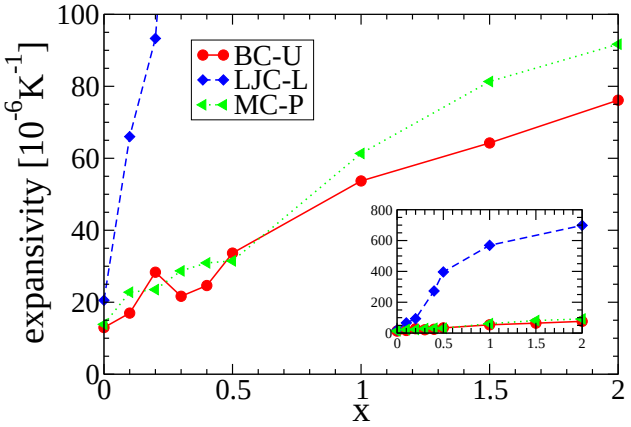


Figure 35: Thermal volume expansivity at ambient pressure. The inset shows a larger scale to accommodate the curve given by the LJC-L potential.

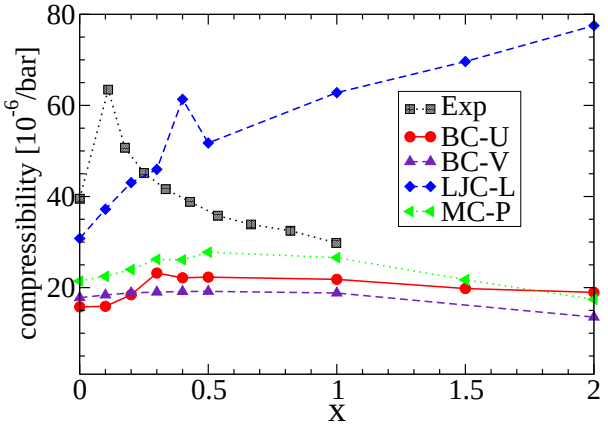


Figure 36: Volume compressibility (bulk modulus). Experimental values are taken from [128, 129].

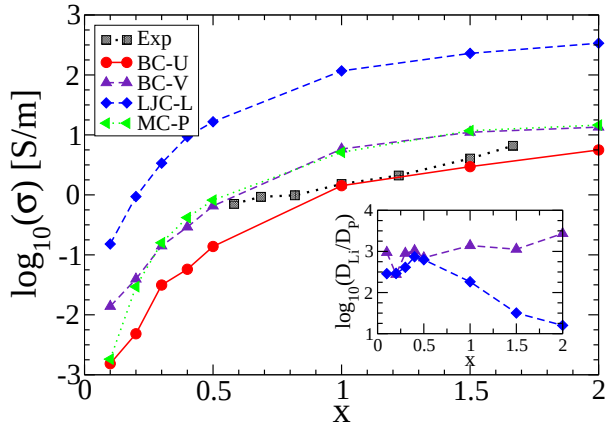


Figure 37: Conductivity σ at 700 K. The experimental values are extrapolated from Fig. 4 of [125]. The inset shows the quotient of lithium and phosphorous diffusion constants. For the LJC-L potential, 500 K data was plotted.

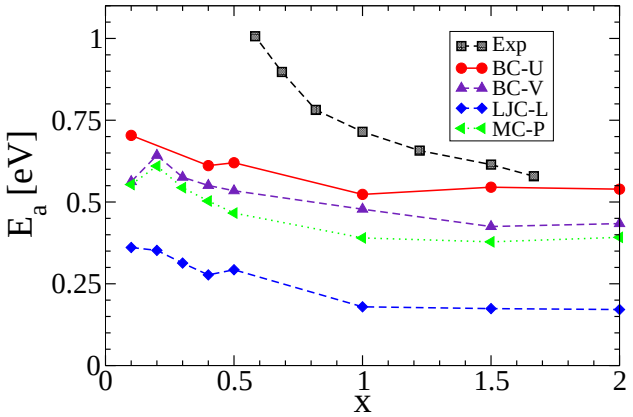


Figure 38: Activation energy. The experimental values are taken from [125].

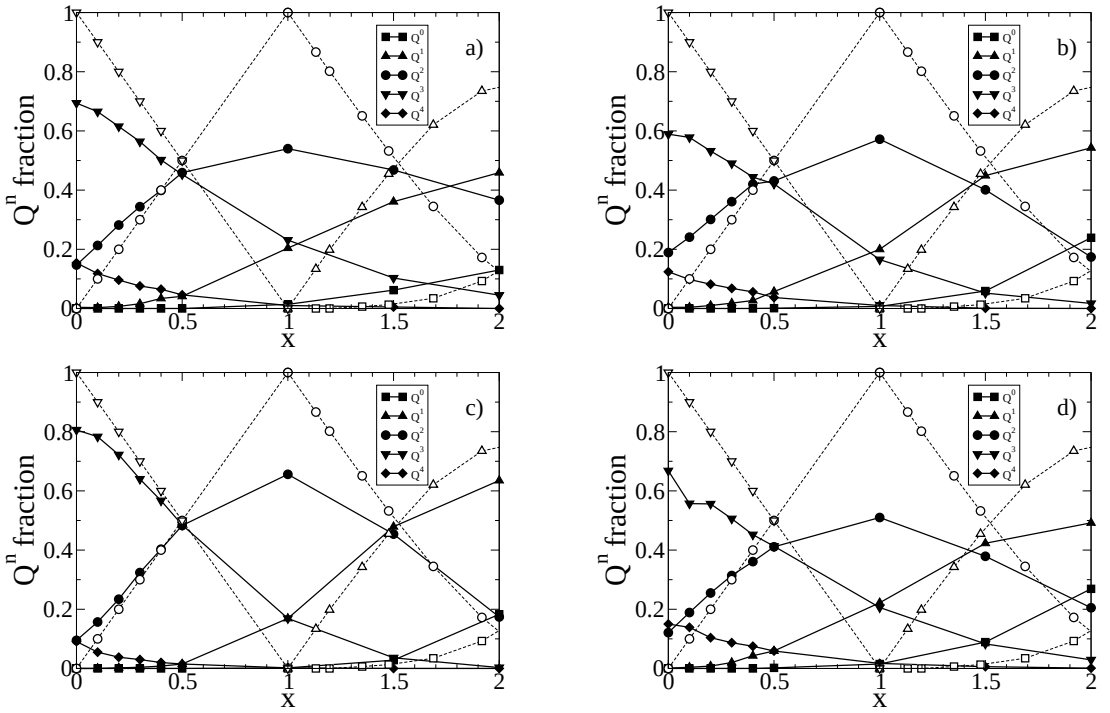


Figure 39: Q^n species distribution of $x$$Li_2O$ - P_2O_5 MD-systems using the potentials BC-U (a), BC-V (b), LJC-L (c), and MC-P (d). Dashed lines with open symbols represent expected values based on experimental data as taken from [116].

3.7 Results for mixed network former glasses

After showing that the BC-U potential generates accurate models of the binary glasses, it will be used to simulate mixed network former glasses. The purpose of this section is twofold. First it shall establish that no critical defects occur. It must for example be checked, whether the oxygen affinity of boron, silicon and phosphorous is not too different. If it was, one might see over-coordination for one of the species and under-coordination for another. Also it is expected that the NFUs do not change drastically upon mixing, something which would show up in the average X-O bond lengths.

The second goal of the chapter is a general characterization of the mixed systems. In particular the properties already used for the evaluation of the binary glasses will be determined. While no systematic experimental data is available now, this might serve as a reference for future investigations, as well as an incentive for experimental studies along the suggested composition variations. An immediate question for the characterization is, whether a mixed network former effect in the activation energies occurs. Since the original motivation for developing the BC-U potential was the investigation of the mixed network former effect, this would be an important achievement and basis for future studies.

In Figs. 40-63 the plots of the properties used for the evaluation of the binary glasses are shown for $0.4Li_2O$ - xSi_2O_4 - $(1-x)B_2O_3$, $0.4Li_2O$ - xB_2O_3 - $(1-x)P_2O_5$, and $0.4Li_2O$ - xP_2O_5 - $(1-x)Si_2O_4$. While most of the curves show a more or less linear change with varying x, there are some features which warrant a closer look.

For one, the average X-O bond lengths in the NFUs do not change over the compositions. This is a strong sign that the NFUs themselves are not altered. With respect to the coordination numbers two points are noteworthy. In the phosphosilicate system some defects occur: a growing number of silicon atoms is not coordinated with four oxygens, when increasing the phosphate content (see Fig. 56). An analysis of the structure reveals, that the missing silicon atoms are mostly coordinated with five oxygens. So the mechanism at play is that non-bridging oxygens of the phosphorous atoms are used to over-coordinate silicon atoms.

In the borophosphate and borosilicate system the fraction of boron atoms which is coordinated with four oxygens increases upon reducing the borate content - a finding which is consistent with experimental data [130]. Note that in [130] a different quantity is plotted. While Figs. 41, 49 and 57 show $[XO_n]/[X]$, the respective plot in [130] shows $[XO_n]$. The fractions can be converted into each other by simply multiplying with x and $(1-x)$ respectively.

But the most important observation is the apparent occurrence of a mixed network former effect in the borophosphate system. Fitting a curve $E(x) = A\exp[-(x-x_0)^2/\sigma]$ yields a minimum at $x = 0.25$ with a depth of about 0.13 eV or 21%. As a test, the same function was used to fit the activation energy data of the borosilicate and the phosphosilicate system, with no minimum occurring. If the observation turns out to be a true effect and not just a random fluctuation (more and longer runs between $x = 0.1$ and $x = 0.4$ are needed in order to establish that), this system would be an important case study for investigating the microscopic causes of the mixed network former effect.

$0.4\mathbf{Li_2O}$ - $x\mathbf{Si_2O_4}$ - $(1-x)\mathbf{B_2O_3}$

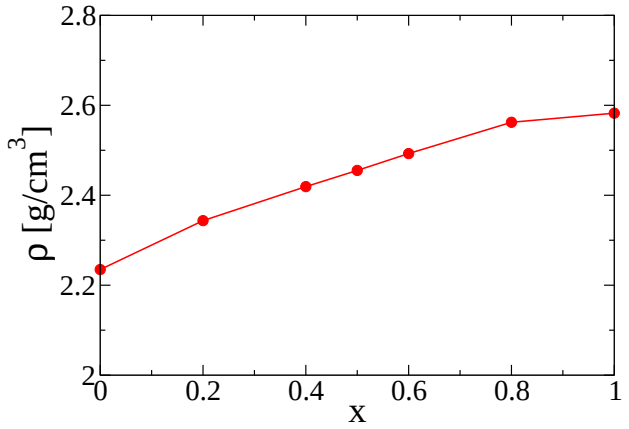


Figure 40: Density $\rho(x)$ at ambient pressure.

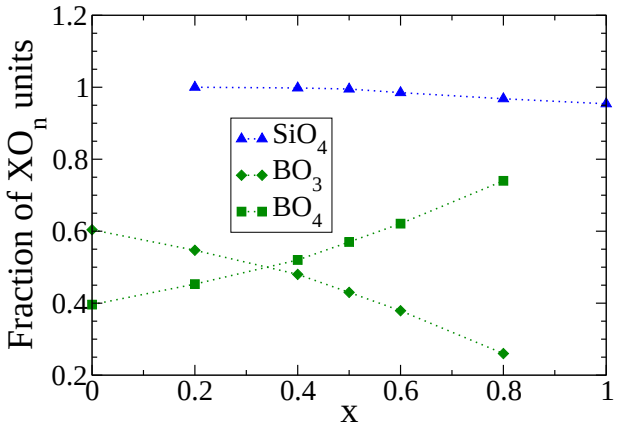


Figure 41: Fraction of Si and B atoms which are coordinated with four or three oxygens (SiO_4, BO_3, and BO_4 units).

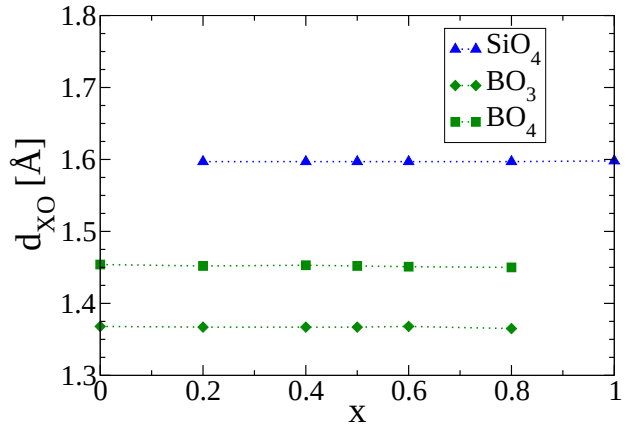


Figure 42: Average X-O bond lengths of SiO_4, BO_3 and BO_4 units.

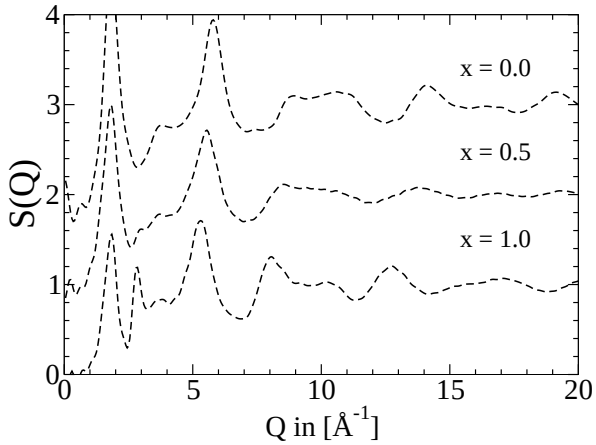


Figure 43: Neutron structure factor for $x = 0.0$, $x = 0.5$, and $x = 1.0$. The curves for $x = 0.0$ and $x = 0.5$ are offset by two and one respectively.

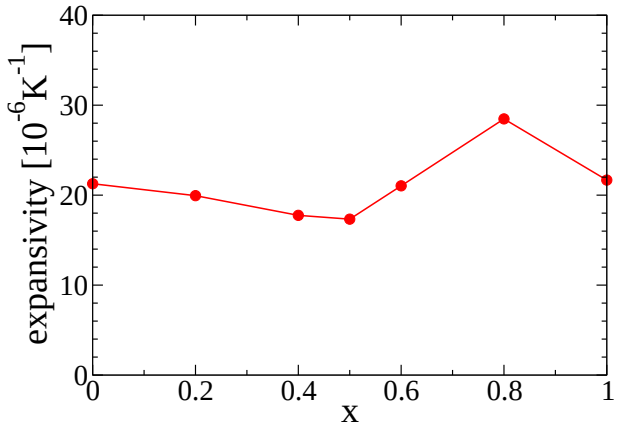


Figure 44: Thermal volume expansivity at ambient pressure.

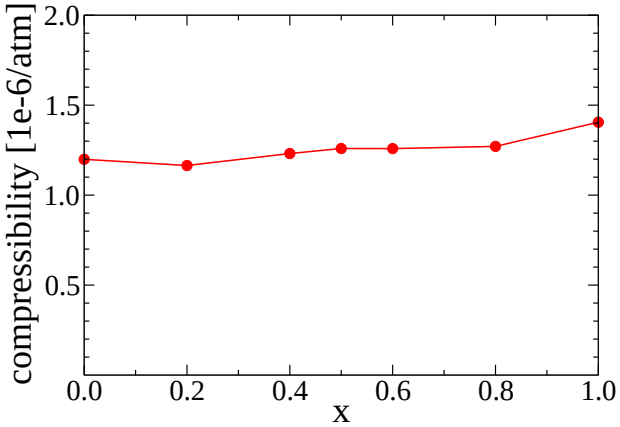


Figure 45: Volume compressibility (bulk modulus).

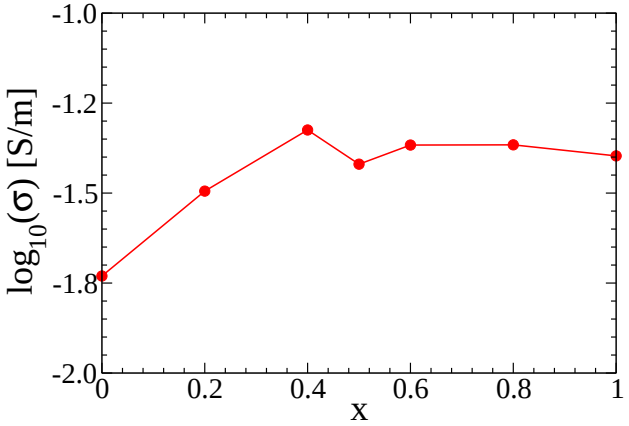


Figure 46: Conductivity σ at 700 K.

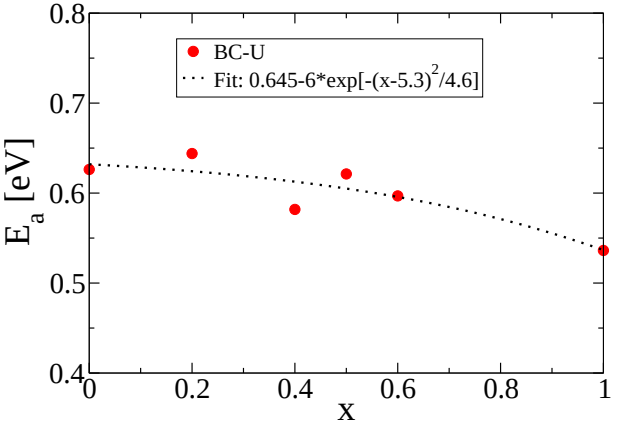


Figure 47: Activation energy.

$0.4\mathbf{Li_2O}$ - $x\mathbf{B_2O_3}$ - $(1-x)\mathbf{P_2O_5}$

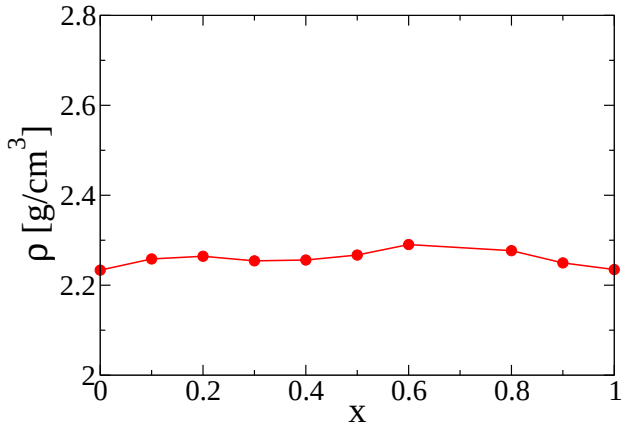


Figure 48: Density $\rho(x)$ at ambient pressure.

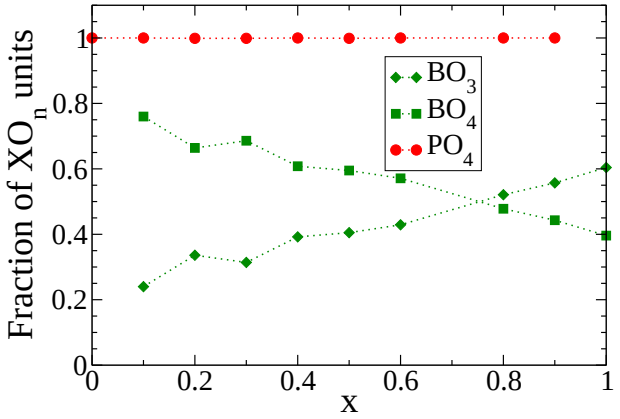


Figure 49: Fraction of P and B atoms which are coordinated with four or three oxygens (PO_4, BO_3, and BO_4 units).

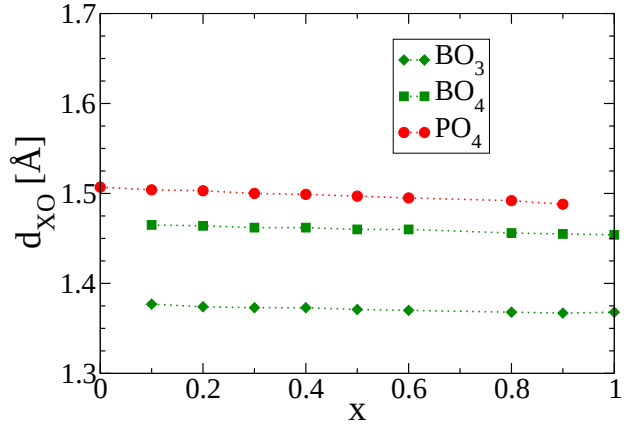


Figure 50: Average X-O bond lengths of PO_4, BO_3 and BO_4 units.

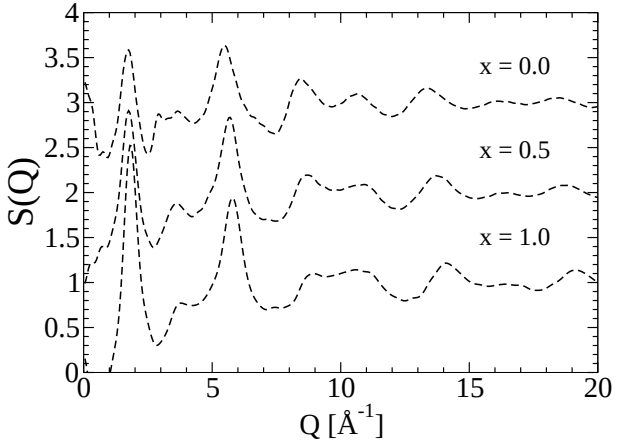


Figure 51: Neutron structure factor for $x = 0.0$, $x = 0.5$, and $x = 1.0$. The curves for $x = 0.0$ and $x = 0.5$ are offset by two and one respectively.

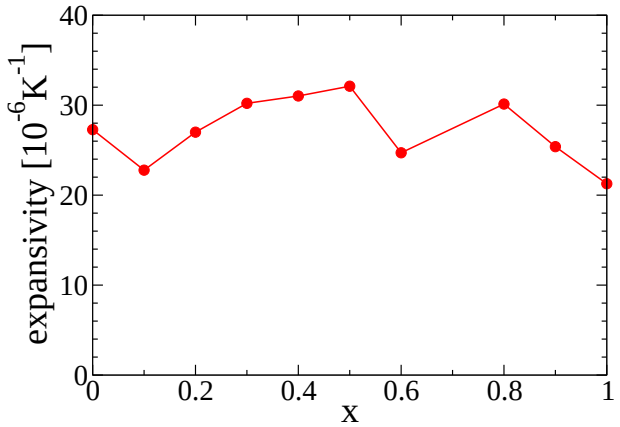


Figure 52: Thermal volume expansivity at ambient pressure.

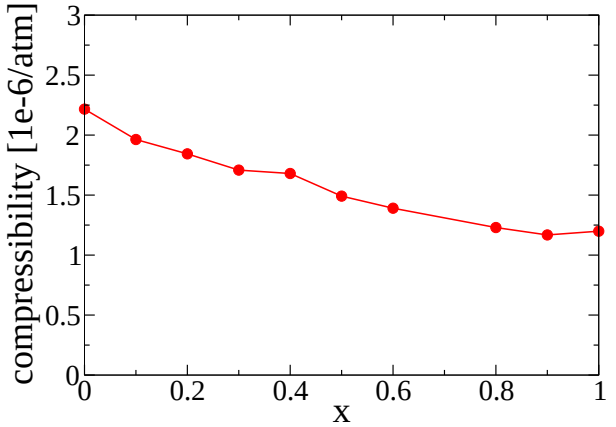


Figure 53: Volume compressibility (bulk modulus).

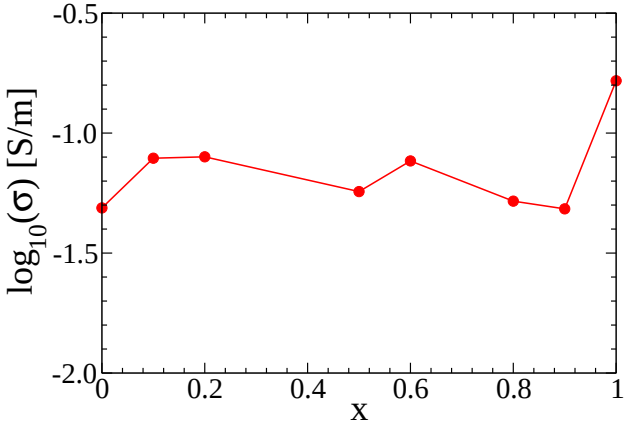


Figure 54: Conductivity σ at 700 K.

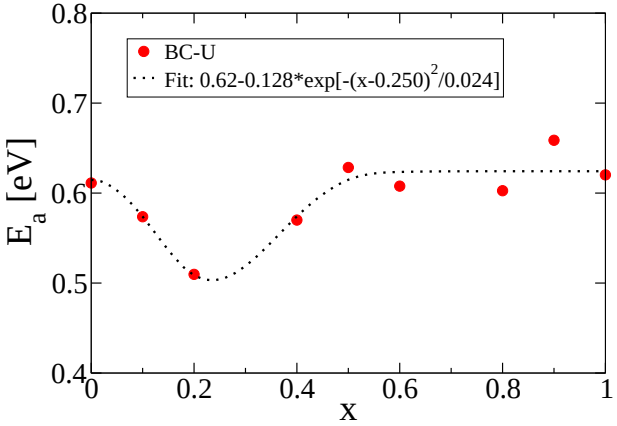


Figure 55: Activation energy.

$0.4\mathbf{Li_2O}$ - $x\mathbf{P_2O_5}$ - $(1-x)\mathbf{Si_2O_4}$

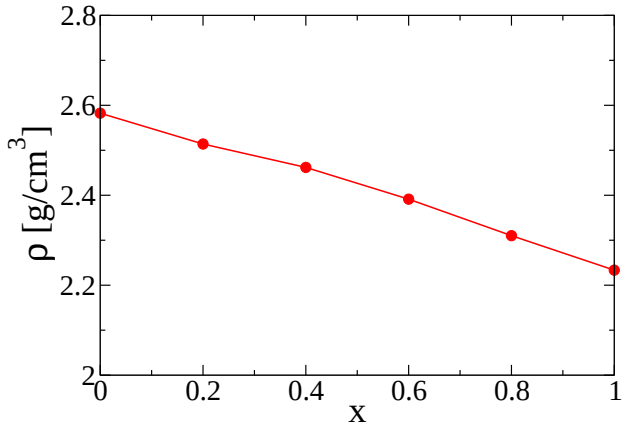


Figure 56: Density $\rho(x)$ at ambient pressure.

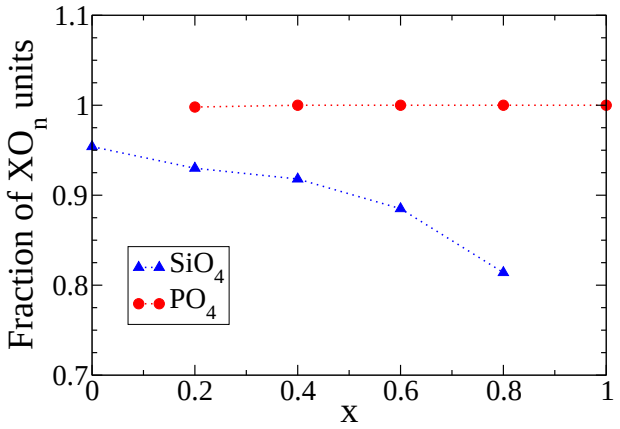


Figure 57: Fraction of P and Si atoms which are coordinated with four oxygens (PO_4 and SiO_4 units).

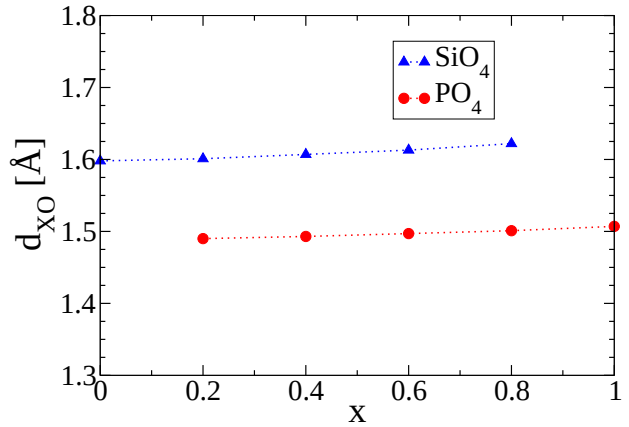


Figure 58: Average X-O bond lengths of PO_4 and SiO_4 units.

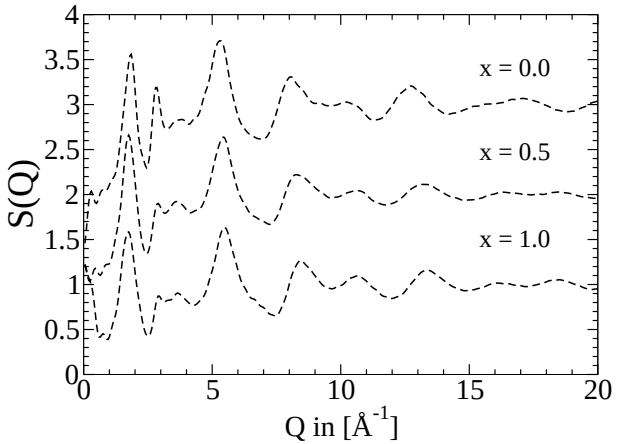


Figure 59: Neutron structure factor for $x = 0.0$, $x = 0.5$, and $x = 1.0$. The curves for $x = 0.0$ and $x = 0.5$ are offset by two and one respectively.

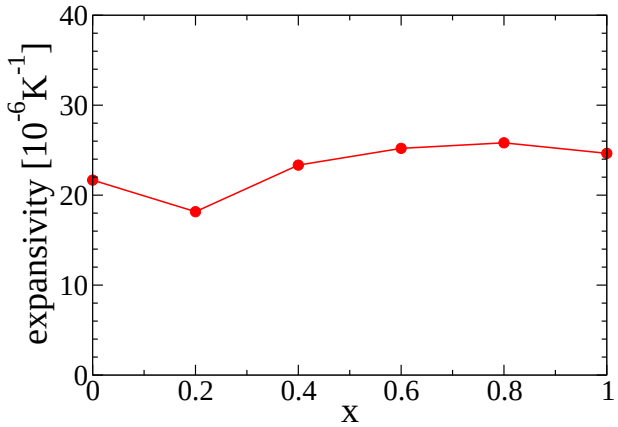


Figure 60: Thermal volume expansivity at ambient pressure.

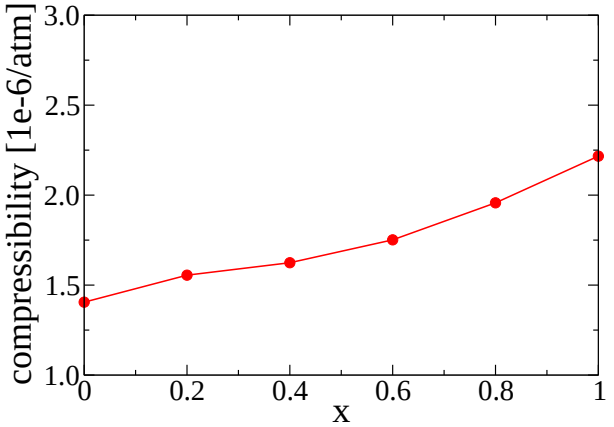


Figure 61: Volume compressibility (bulk modulus).

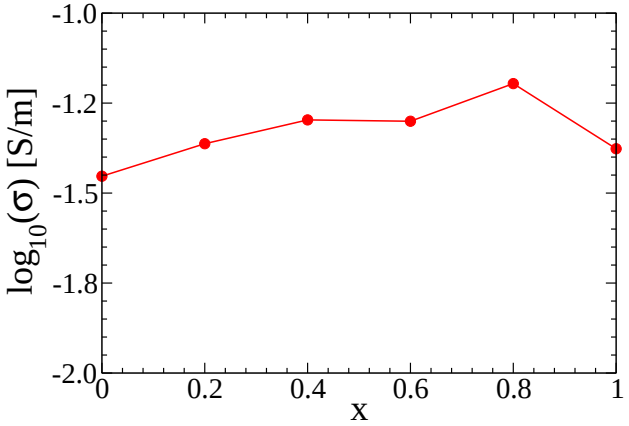


Figure 62: Conductivity σ at 700 K.

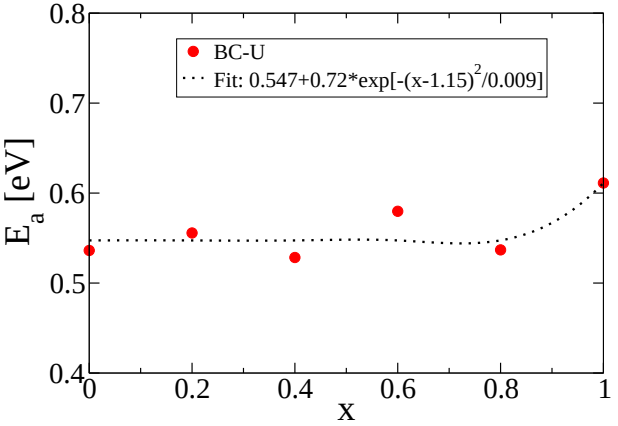


Figure 63: Activation energy.

3.8 Summary of the evaluation

	Silicate		Borate		Phosphate			
Potential	BC-U	MC-P	BC-U	BC-VH	BC-U	BC-V	LJC-L	MC-P
Coord.	2	2	2	2	2	1	2	1
Decoupl.	2	2	2	0	2	2	1	2
ρ	1	1	2	0	1	0	1	1
d_{XO}	1	1	2	1	1	0	1	1
Compr.	1	1	1	1	1	1	0	1
Th. exp.	2	1	2	-	1	-	0	1
$S(Q)$	2	2	1	2	1	1	2	1
σ	2	1	2	1	2	1	0	1
E_a	1	1	1	1	1	1	0	1
Structure	10/12	10/12	11/12	9/12	9/12	4/12	10/12	6/12
Dynamic	9/10	8/10	9/10	2/10	9/10	8/10	3/10	8/10
Total	22/26	20/26	23/26	12/24	20/26	13/24	13/26	16/26

Table 5: Assessment points awarded for agreement results from MD simulations with experimental data according to the evaluation scheme introduced in Sec. 3.5. Comparison plots of the different properties are shown in Figs. 15-38.

Table 3.8 gives the final scores of the force field evaluation. One conclusion, drawn from comparing the final numbers with all the evidence collected over the cause of the evaluation, is that the final three sums (structural properties, features of the ion dynamics, and the total score) give for each force field an adequate assessment of its applicability for different investigations.

For example, judging from Figs. 31 – 38 the LJC-L potential of Liang *et al.* [59] successfully reproduces the structural features of lithium phosphate. Of all investigated phosphate potentials, it has the second lowest defect rate with respect to the coordination number of phosphorous atoms. It shows the best agreement of bond lengths with experimental values. It has the best agreement with experimental neutron structure factors. And it produces the Q-species distribution which is closest to the experimental values. On the other hand, the LJC-L potential has serious shortcomings with respect to the ion-dynamics. Considering that the determined diffusion constants at 500 K are three to four orders of magnitude larger than the experimental values, it is highly likely that the fundamental conduction mechanisms of this MD system are different from the mechanisms in the real phosphate. Summarising our investigations, the LJC-L potential can be used to investigate structural properties, but it should be avoided when studying transport phenomena.

The same conclusion can be drawn from the three final scores for the potential. The LJC-L potential has the second highest structural score but also the second lowest dynamics score.

Comparing the total points achieved by the investigated potentials, it is evident that

the BC-U potential consistently has the highest score. Indeed, the only other force field reaching 75% of the possible points is the MC-P force field of Pedone *et al.* [60] when used to simulate lithium silicate.

The outcome is more differentiated when comparing the achieved points in the structural and dynamics subcategory. The LJC-L potential, the BC-VH potential and the MC-P potential (for silicates) are equally or better suited for structural investigations as the new BC-U potential. Similarly, the BC-V potential and the MC-P potential (for both silicates and phosphates) reproduce the ion dynamics as well as the BC-U potential[23]. Nevertheless, the new BC-U potential developed in this work shows no particular weaknesses for any of the alkali glasses and is at least as well suited as the established force fields for all of them.

As a side note, this outcome also confirms that the approach chosen to develop the force field is a valid one. Despite its lack of comprehensiveness and — one might argue — elegance, the evaluation proves that potentials obtained by more sophisticated means are not superior.

The BC-U potential is fundamentally suited to simulate the mixed network former glasses, as well. One of the main expectations, that the basic NFUs will not change, is mostly fulfilled. A clear defect occurs only in the phosphosilicate system, where up to 20% of all silicon atoms are coordinate with five, not four, oxygen atoms

Further investigations are necessary in order to verify that these mixed systems are good representations of the specific glass compositions. For that, more systematic experimental data is needed. It would, for example, be worthwhile (but beyond the scope of the current work) to investigate the Q-species distribution and the connectivity between the different Q-species.

One finding that might be surprising at first sight, is that the most problematic systems are the ones containing silicon. In both the lithium silicate as well as the phosphosilicate systems a substantial number of silicon atoms have five oxygens in its first coordination shell. Consequently, a too-large density for both systems is seen. Therefore, it should be considered to determine new Si-O interaction parameters, as was done for B-O and P-O.

It is also promising that (apparently) the mixed glass former effect of the activation energies occurs in the borophosphate system, but not in the borosilicate and phosphosilicate systems. This behavior might enable the verification of proposed microscopic mechanisms of the mixed glass former effect, if these mechanisms are found to exist in the borophosphate but not in the other MD systems. Still, some caution is warranted here. The calculated values for conductivity and activation energy are not reliable enough to exclude the possibility that the minimum seen in the activation energy of the borophosphate system is only a fluctuation. More and longer MD runs are needed to verify that result. In particular, one should run multiple independent simulations

[23]Note that Vogel *et al.* used the BC-V potential primarily for investigations of ion-dynamics, which the evaluation suggests is its strength [124, 131]. Moreover their investigations were focused on relatively high alkali content systems. In these systems, the structural shortcomings of the BC-V potential are less pronounced as at low alkali concentrations (see the coordination numbers in Fig. 32).

and average over the results to get more conclusive evidence. While running multiple independent simulations was not possible in the current work[24], it is not out of scope for a future investigation. In fact the average simulation time for the borophosphate system is smaller than the average simulation time needed for the systems investigated here, since the lithium concentration is not particularly small, and hence the MSD runs do not need to be extremely long.

4 Conclusions and outlook

This work presents a new MD code for graphics cards called LAMMPS$_{CUDA}$. Using the new paradigm of heterogenous computing, it allows for an efficient exploitation of graphics cards to accelerate a wide range of MD simulations. Extensive benchmarks show that LAMMPS$_{CUDA}$ is equally well suited for single workstations, as for large clusters. It achieves typical per node speedups of 5 to 10 times. With a particular focus on scalability, LAMMPS$_{CUDA}$ makes efficient use of several hundred GPUs for a single simulation, as was proven by tests on the large scale GPU cluster Lincoln of the National Center for Supercomputing Applications at the University of Illinois.

Because such systems typically deliver five times more computational power than conventional supercomputers of the same cost, more and more of them are installed [10], and new programs such as LAMMPS$_{CUDA}$ are needed to use them.

It is worth noting that not only the necessary hardware is cheaper with respect to a price-performance ratio, but it is also much more cost efficient during operation. The simulations discussed in the second part of this work ran for about half a year using 20 GPUs. The associated electricity costs can be estimated to 3,600 € compared with 44,000 € which it would have cost to do the same simulations on conventional hardware[25]. These cost savings also mean that using GPUs is much more environmentally friendly than doing the same simulations on conventional hardware.

As a consequence, LAMMPS$_{CUDA}$ allows to do investigations which were previously not within reach given the resources available. One of these investigations is the development and evaluation of a new MD potential for ion-conducting network glasses presented in Sec. 3. The main purpose of the new potential, called BC-U, is to allow for the simulation of mixed network former glasses such as borosilicates, phosphosilicates and borophosphates. The potential is based on a force field developed by Tsuneyuki *et al.* for SiO_2 which was later extended by Habasaki *et al.* in order to enable the simulation of alkali silicates [50, 52]. Extensive simulations, modeling lithium silicate, lithium

[24] Counting the various potentials, 11 different systems were investigated with six to ten compostions each.

[25] It was estimated that the ten "Perseus" dual GPU workstations available in the physics department of the TU-Ilmenau were used for six months with an average load of 60% and a power draw of 0.6 kW. Furthermore, based on the benchmark in Sec. 2.4.1 it was assumed that a single GPU is as fast as seven conventional dual-CPU compute nodes with each node drawing 0.35 kW. It was also assumed that an additional factor of 1.5 must be applied to the conventional nodes, since those would have been hosted in a compute center with active cooling. The latter typically adds a factor 1.5-2.0 to the total energy consumption. The price for a kWh was assumed to be 0.23 €.

borate and lithium phosphate with varying lithium content, were carried out in order to validate the force field. Calculated and experimental values for various properties show very good agreement across all compositions. Indeed, the new potential scores consistently more points on a newly-introduced evaluation scale, than various potentials which were used for a comparison. It is therefore suggested that the BC-U potential is - absent specific reasons to do otherwise - the preferable choice for doing any MD simulations in one of the investigated systems. Incidentally, with a grand total of more than ten billion simulated timesteps, the simulations performed with the benchmark force fields represent their most comprehensive composition-dependend study as well.

The investigations presented here also show, that the new BC-U potential can indeed model the mixed network former systems. As expected, the network former units are largely unchanged upon mixing. Only in the phosphosilicate system some defects occur, with up to 20% of the silicon atoms not being coordinated with four oxygens.

Another promising finding is the minimum in the activation energy observed for the borophosphate system, which is not seen in either the bososilicate or the phosphosilicate system. While more simulation runs are needed to exclude the possibility that this observation is coincidental, it is assuring that this finding is consistent with experimental data. The latter suggests that the borophosphate system is a glass in which the mixed network former effect appears consistently [130].

The results point to several future projects. One potential study is the verification of proposed microscopical mechanisms which cause the mixed network former effect. In [132] - a work to which this author contributed - two microscopic causes were suggested. In situation I, the energy barriers in heterogenous environments (both network former species can be found in the vicinity) are lower than in homogenous ones, while in situation I, the energy traps in heterogenous environments are assumed to be shallower. With MD simulations, these models can directly be tested by determining ion site energies and barrier heights between adjacent ion sites. Extensive studies in that direction have already been performed in binary glasses, mostly in alkali phosphates and silicates [32, 131, 35, 31]. Using the BC-U potential, which produces both mixed network former glasses with and without a mixed network former effect enables a direct confirmation or falsification of the proposed mechanisms.

A second study which is now possible is an investigation of the ion site distribution in dependence of the alkali content. While it was found that the number of ion sites is only slightly larger than the number of ions itself [32, 31], these studies were done with large ion concentrations. Because of the long runtimes necessary to identify the sites, as well as the reduced conductivity at low alkali concentrations, such investigations were not yet undertaken. Using LAMMPS$_{CUDA}$, this is nevertheless feasible. Determining the ion site concentration with respect to the alkali content would also allow a modeling of the effect with more coarse-grained methods, such as Monte Carlo algorithms. Using the MD results as input, it would be informative to see if a change in the ion channel connectivity alone is enough to explain the strong reduction in conductivity with decreasing alkali content, and whether a quantitative agreement with the MD simulation can be achieved. This study can also be combined with an investigation of finite size effects. Assuming a

decreasing connectivity of the ion channel network, one can expect such effects to play an increasingly important role at small alkali concentrations.

A third project which could be enabled by using LAMMPS$_{CUDA}$ is an exploration of cooling rate effects on the properties of MD systems. In particular, an investigation of the Q-species distribution with varying cooling rate, would clarify the extent to which structural deviations from experimental observations are caused by differences in the process history, rather than shortcomings of the interaction models. With LAMMPS$_{CUDA}$ cooling rates in the range of 10^{10} K/s to 10^{14} K/s (10 K/ns - 100 K/ps) should be possible.

A Acknowledgments

First and foremost I want to thank my adviser Prof. Philipp Maass who supported me through all those years pursuing my diploma and my PhD. He taught me the art of writing papers and – I hope others agree to that – helped shaping me into someone resembling a scientist. And though I strayed from the path of becoming a classical theoretical physicist, Prof. Maass had seemingly endless faith into my ability to make something useful out of my ideas. I hope that he sees this thesis as a proof that his faith was warranted. I am also grateful that he introduced me into the more pleasant aspects of international collaborations. While he did not succeed in making me appreciate a good glass of wine, I certainly appreciate the try - and the dinners were splendid nevertheless.

I also want to thank the Studienstiftung des deutschen Volkes for bankroling the last three years. Knowing that I do not have to worry about money made it significantly easier to follow some ideas which were not sure to produce results. Additionally the Studienstiftung is a role model in child friendliness. Getting an extra year (even though I did not fully need it) and more money, gave me the possibility to be a much more integral part of the first year of my son - an experience I treasure, and for which I am very grateful.

Then there are the people who directly contributed to this thesis. Without Martin Körner I would probably still be running simulations to get my results together. Offloading a good part of the particular tiresome task to manage the jobs to Martin, certainly helped a lot in finally finishing this project.

Lars Winterfeld was a great support in the early stages of the development of LAMMPS$_{CUDA}$. Not only did he contribute significantly to the original framework of the program, he also forced me to write better code, and was a valuable discussion partner to refine my own ideas.

I would like to thank Michael Schuch for all the work we did together. I hope that he feels that I contributed as much to further his progress, as he did to mine.

Another person who helped making this thesis possible is Prof. Erich Runge. Without his enthusiasm for the GPU programming I introduced at the TU Ilmenau - and the subsequent willingness to support that enthusiasm with a lot of money - it would have been impossible to pursue the ambitious plan for developing and verifying the BC-U force field for mixed network former glasses.

I also want to acknowledge the help of Steve Martin, who provided an extensive list with data and references for densities of lithium phosphates.

By inviting me into the GPU-LAMMPS group effort, Paul Crozier opened the door to a number of contacts who have been immensely helpful for improving the multi-GPU capabilities of LAMMPS$_{CUDA}$. In particular Alex Kohlmayer, who helped me get access to Lincoln, and Greg Scantlen, who let me play with his company's GPU-cluster, furthered the cause of my endeavor - and Greg: you should probably not give any PHD student who happens to come along a full list of all your root, user and who-knows-what-else accounts.

Hartmut Grille provided valuable help for improving the final manuscript. In particular, he remembered me that not everyone has worked for three to five years on my field of studies and that what is considered "basic" knowledge might depend a lot on a persons prior history.

I would also like to express my gratitude to all my other colleagues of the two theoretical physics groups of Ilmenau, who provided a supporting and stimulating environment for all that time. I never regretted staying here.

Amy Young and Michael Trott helped me in the final days of writing this thesis.

And last but not least I want to thank the funniest person in the world for all the joy she brought to my life.

B Reduction code examples

B.1 CUDA

The code can be put into one file (e.g. "cuda_reduction.cu") and compiled with
`nvcc -arch=sm_13 -o cuda_reduction cuda_reduction.cu`.

```
#include <stdio.h>
#include <stdlib.h>
#define N 4096

__global__ void cuda_reduce(int* a,int* b)
{
  a+=blockIdx.x*blockDim.x;
  for(int i=2;i<=blockDim.x;i*=2)
  {
    if(threadIdx.x<blockDim.x/i)
    a[threadIdx.x]+=a[threadIdx.x+blockDim.x/i];
    __syncthreads();
  }
  if(threadIdx.x==0) atomicAdd(b,a[0]);
}

int main (int argc, char *argv[])
{
 int a[N];
 for(int i=0; i<N; i++) a[i]=1;

 int sum = 0;

 int* a_dev;
 int* b_dev;

 int nblocks=N/256;

 cudaSetDevice(4);
 cudaMalloc(&a_dev,sizeof(int)*N);
 cudaMalloc(&b_dev,sizeof(int));
 cudaMemset(b_dev,0,sizeof(int));
 cudaMemcpy(a_dev,a,sizeof(int)*N,cudaMemcpyHostToDevice);

 cuda_reduce<<<nblocks,256>>>(a_dev,b_dev);

 cudaMemcpy(&sum,b_dev,sizeof(int),cudaMemcpyDeviceToHost);

 printf("\nTotal Sum %i\n",sum);
}
```

B.2 OpenMP

The code can be put into one file (e.g. “omp_reduction.cpp”) and compiled with
`g++ -fopenmp -o omp_reduction omp_reduction.cpp`.

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N 4096

int main (int argc, char *argv[])
{
 int np = 2;
 if(argc>1) np = atoi(argv[1]);
 omp_set_num_threads(np);

 int a[N], my_sum, sum=0;

 for(int i=0; i<N; i++) a[i]=1;

 #pragma omp parallel private(my_sum)
 {
   my_sum = 0;

   #pragma omp for schedule(static,1)
   for(int i = 0; i < N; i++)
        my_sum = my_sum + a[i];

   printf("MySum: %i\n",my_sum);

   #pragma omp critical
   sum = sum + my_sum;
 }
 printf("\nTotal Sum %i\n",sum);
}
```

B.3 OpenMP + CUDA

The code can be put into one file (e.g. "omp_cuda_reduction.cu") and compiled with
`nvcc -arch=sm_13 -Xcompiler -fopenmp -o omp_cuda_reduction omp_cuda_reduction.cu`.

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N 4096

__global__ void cuda_reduce(int* a,int* b)
{
  a+=blockIdx.x*blockDim.x;
  for(int i=2;i<=blockDim.x;i*=2)
  {
    if(threadIdx.x<blockDim.x/i)
    a[threadIdx.x]+=a[threadIdx.x+blockDim.x/i];
    __syncthreads();
  }
  if(threadIdx.x==0) atomicAdd(b,a[0]);
}

int main (int argc, char *argv[])
{
 int np = 2;
 if(argc>1) np = atoi(argv[1]);
 omp_set_num_threads(np);
 int a[N], my_sum, sum=0;

 for(int i=0; i<N; i++) a[i]=1;
 #pragma omp parallel private(my_sum)
 {
   my_sum = 0;
   int nblocks=N/256/np;
   int chunk = N/np;

   int id = omp_get_thread_num();

   int* a_dev,* b_dev;

   cudaSetDevice(id);
   cudaMalloc(&a_dev,sizeof(int)*chunk);
   cudaMalloc(&b_dev,sizeof(int));
   cudaMemcpy(a_dev,a+id*chunk,chunk*sizeof(int),cudaMemcpyHostToDevice);
   cuda_reduce<<<nblocks,256>>>(a_dev,b_dev);
   cudaMemcpy(&my_sum,b_dev,sizeof(int),cudaMemcpyDeviceToHost);
   printf("MySum: %i\n",my_sum);

   #pragma omp critical
   sum = sum + my_sum;
 }
 printf("\nTotal Sum %i\n",sum);
}
```

B.4 MPI

The code can be put into one file (e.g. "mpi_reduction.cpp") and compiled with
`mpic++ -o mpi_reduction mpi_reduction.cpp`.
This should work on most systems with MPI installed and configured. Often instead of `mpic++` one might need to use `mpicpp` or `mpicxx`. Please consult your system admin for specifics about compiling MPI programs on your machine.

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#define N 4096

int main (int argc, char *argv[])
{
 long sum = 0;
 int my_sum,id;

 int num_threads;

 MPI_Init(&argc,&argv);                  /* starts MPI */
 MPI_Comm_rank(MPI_COMM_WORLD, &id);  /* get current process id */
 MPI_Comm_size(MPI_COMM_WORLD, &num_threads); /*get number of process */
 MPI_Status Stat;

 int nblocks = N/256/num_threads;
 int chunk = N/num_threads;

 int* a = new int[chunk];
 for(int i=0; i<chunk; i++) a[i]=1;

 for(int i=0; i<chunk; i++) my_sum += a[i];

 printf("my_sum: %i %i\n",id,my_sum);
 if(id == 0)
 {
   sum = my_sum;
   for(int i=1;i<num_threads;i++)
   {
     MPI_Recv(&my_sum, 1, MPI_INT, i, 1, MPI_COMM_WORLD,&Stat);
     sum += my_sum;
   }
   printf("\n %li\n",sum);
 }
 else
   MPI_Send(&my_sum, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);

 MPI_Finalize();
}
```

B.5 MPI + CUDA

The code should be put into two files ("mpi_cuda_reduction.cu" and "mpi_cuda_reduction_host.cpp"). The following part is the CUDA code and can be compiled with

`nvcc -c -arch=sm_13 mpi_cuda_reduction.cu`.

```
#include <cstdlib>
extern "C" int cuda_reduce_host(int id,int chunk,int nblocks,int* a);

__global__ void cuda_reduce(int* a,int* b)
{
  a+=blockIdx.x*blockDim.x;
  for(int i=2;i<=blockDim.x;i*=2)
  {
    if(threadIdx.x<blockDim.x/i)
    a[threadIdx.x]+=a[threadIdx.x+blockDim.x/i];
    __syncthreads();
  }
  if(threadIdx.x==0) atomicAdd(b,a[0]);
}

int cuda_reduce_host(int id,int chunk,int nblocks,int* a)
{
 int* a_dev;
 int* b_dev;
 int my_sum;
 cudaSetDevice(id);
 cudaMalloc(&a_dev,sizeof(int)*chunk);
 cudaMalloc(&b_dev,sizeof(int));
 cudaMemset(b_dev,0,sizeof(int));
 cudaMemcpy(a_dev,a,chunk*sizeof(int),cudaMemcpyHostToDevice);

 cuda_reduce<<<nblocks,256>>>(a_dev,b_dev);

 cudaMemcpy(&my_sum,b_dev,sizeof(int),cudaMemcpyDeviceToHost);

 return my_sum;
}
```

The second part contains the MPI commands and is compiled with
`mpic++ -c mpi_cuda_reduction_host.cpp`.

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#define N 4096
extern "C" int cuda_reduce_host(int id,int chunk,int nblocks,int* a);

int main (int argc, char *argv[])
{
 long sum = 0;
 int my_sum,id;

 int num_threads;

 MPI_Init(&argc,&argv);                  /* starts MPI */
 MPI_Comm_rank(MPI_COMM_WORLD, &id);  /* get current process id */
 MPI_Comm_size(MPI_COMM_WORLD, &num_threads); /*get number of process */
 MPI_Status Stat;

 int nblocks = N/256/num_threads;
 int chunk = N/num_threads;

 int* a = new int[chunk];
 for(int i=0; i<chunk; i++) a[i]=1;

 my_sum = cuda_reduce_host(id,chunk,nblocks,a);

 printf("my_sum: %i %i\n",id,my_sum);
 if(id == 0)
 {
   sum = my_sum;
   for(int i=1;i<num_threads;i++)
   {
     MPI_Recv(&my_sum, 1, MPI_INT, i, 1, MPI_COMM_WORLD,&Stat);
     sum += my_sum;
   }
   printf("\n %li\n",sum);
 }
 else
   MPI_Send(&my_sum, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);

 MPI_Finalize();
}
```

The final program can be linked with
`mpic++ -o mpi_cuda_reduction mpi_cuda_reduction_host.o mpi_cuda_reduction.o -lcuda -lcudart`.
This assumes that paths to cuda are already included in the environmental variable "LD_LIBRARY_PATH".
To run the code with two MPI processes use:
`mpirun -np 2 mpi_cuda_reduction`.

B.6 MPI + OpenMP + CUDA

The CUDA kernel and its object file `mpi_cuda_reduction.o` from the previous MPI + CUDA example can be used again. The host code following now can be compiled with:

```
mpic++ -c -openmp mpi_openmp_cuda_reduction_host.cpp
```

```
#include <mpi.h>
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N 4096
extern "C" int cuda_reduce_host(int id,int chunk,int nblocks,int* a);

int main (int argc, char *argv[])
{
 long sum = 0;
 int my_sum = 0;
 int id;

 int num_omp_threads = 2;
 if(argc>1) num_omp_threads = atoi(argv[1]);
 omp_set_num_threads(num_omp_threads);

 int num_mpi_threads;

 MPI_Init(&argc,&argv);                /* starts MPI */
 MPI_Comm_rank(MPI_COMM_WORLD, &id);  /* get current process id */
 MPI_Comm_size(MPI_COMM_WORLD, &num_mpi_threads); /*get number of process */
 MPI_Status Stat;

 int nblocks = N/256/num_mpi_threads/num_omp_threads;
 int chunk_mpi = N/num_mpi_threads;
 int chunk_omp = chunk_mpi/num_omp_threads;

 int* a = new int[chunk_mpi];
 for(int i=0; i<chunk_mpi; i++) a[i]=1;
#pragma omp parallel
 {
   int omp_id = omp_get_thread_num();
   int my_sum_omp = cuda_reduce_host(id,chunk_omp,nblocks,a+omp_id*chunk_omp);
   printf("my_sum_omp: %i %i %i\n",id,omp_id,my_sum_omp);
  #pragma omp critical
   my_sum = my_sum + my_sum_omp;
 }
 printf("my_sum_mpi: %i %i\n",id,my_sum);

 if(id == 0)
 {
   sum = my_sum;
   for(int i=1;i<num_mpi_threads;i++)
   {
     MPI_Recv(&my_sum, 1, MPI_INT, i, 1, MPI_COMM_WORLD,&Stat);
     sum += my_sum;
   }
```

```
    printf("\n %li\n",sum);
  }
  else
    MPI_Send(&my_sum, 1, MPI_INT, 0, 1, MPLCOMM_WORLD);

  MPI_Finalize();
}
```

The final program can be linked with

```
mpic++ -openmp -o mpi_openmp_cuda_reduction
mpi_openmp_cuda_reduction_host.o mpi_cuda_reduction.o -lcuda -lcudart.
```

This assumes that paths to cuda are already included in the environmental variable "LD_LIBRARY_PATH".
To run the code with two MPI processes and two OpenMP threads each use:

```
mpirun -np 2 mpi_openmp_cuda_reduction 2.
```

C Benchmark systems

- **LJ**. Potential: Lennard-Jones (lj/cut), Cutoff: 2.5 σ_0, Density: 0.84 σ_0^{-3}, Temperature: 1.6.
- **Silicate (cutoff)**. Potential: Buckingham + Coulomb (buck/coul/cut), Cutoff: 15.0 Å, Density: 0.09 $Å^{-3}$, Temperature: 600 K.
- **Silicate**. Potential: Buckingham + Coulomb (buck/coul/long), Atoms: 11,664, Cutoff: 10.0 Å, Density: 0.09 $Å^{-3}$, Long range coulomb solver: PPPM (Precision: 2.4e-6), Temperature: 600 K.
- **EAM**. Potential: Embedded atom method (EAM) [133, 134], Atoms: 256,000, Cutoff: 4.95 Å, Density: 0.0847 $Å^{-3}$, Temperature: 800 K.
- **Coarse Grained**. Potential: Coarse grained systems (cg-cmm) [135, 136], Atoms: 160,560, Cutoff: 15.0 Å, Temperature: 300 K.
- **Rhodopsin**. Potential: CHARMM force field + Coulomb (lj/charmm/coul/long) [137], Atoms: 32,000, Cutoff: 10.0 Å, Long range coulomb solver: PPPM (Precision: 1e-7), Temperature: 300 K.
- **Granular**. Potential: Granular force field (gran/hooke) [138, 139, 140], Atoms: 1,152,000, Density: 1.07 σ_0^{-3}, Temperature: 19.

D Hardware systems

- Workstation Server A (WS_A)
  Intel Q9550 @ 2.8GHz
  8GB DDR2 RAM @ 800 MHz
  Mainboard: EVGA 780i 3xPCIe2.0 16x
  2 x NVIDIA GTX 280
  CentOS 5.4
- Workstation Server B (WS_B)
  Intel i7 950 @ 3.0GHz
  24GB DDR3 RAM @ 1066 MHz
  Mainboard: Asus P6X58D-E 3xPCIe2.0 16x
  2 x NVIDIA GTX 470
  CentOS 5.5
- GPU Cluster (CL)
  2 x Intel X5550 @ 2.66GHz
  48GB DDR3 RAM @ 1066 MHz
  Mainboard: Supermicro X8DTG-QF R 1.0a 4xPCIe2.0 16x
  4 x NVIDIA Tesla C1060
  Scientific Linux 5.4
- NCSA Lincoln (Lincoln)
  192 Nodes
  2 x Intel X5300 @ 2.33GHz
  16GB DDR2 RAM
  2 x NVIDIA Tesla C1060 on one PCIe2.0 8x (two nodes share one S1070)
  SDR Infiniband
  Red Hat Enterprise 4.8

E Calculation of the structurefactor[26]

The partial and total radial distribution functions $g_{ij}(r)$ and $G(r)$ as well as the total scattering structure factor $S(Q)$ were calculated according to the Partial Distribution Function formalism (see Ref. [143] for a discussion of different possible definitions of scattering functions). The partial radial distribution functions are given as

$$g_{ij}(r) = \frac{n_{ij}(r)}{4\rho_j \pi r^2 \mathrm{d}r} \quad , \tag{11}$$

where $n_{ij}(r)$ is the average number of particles of type j between distances $r - \mathrm{d}r/2$ and $r + \mathrm{d}r/2$ from a particle of type i, and ρ_j is the mean number density of particles of type j.

The total radial distribution function $G(r)$ is calculated by

$$G(r) = 4\pi r \rho_0 \left[\sum_{i,j=1}^{m} [w_{ij} g_{ij}(r)] - 1 \right] \quad , \tag{12}$$

where ρ_0 is the total number density of all atoms in the system, m is the number of particle types, and w_{ij} are weighting factors:

$$w_{ij} = \left(\sum_{k=1}^{m} c_k \bar{b}_k \right)^{-2} c_i c_j \bar{b}_i \bar{b}_j \quad . \tag{13}$$

Here $c_i = \rho_i / \rho_0$ are the molar fractions of particles of type i, and $\bar{b}_i$ is their average bound coherent scattering length. In order to calculate X-ray diffraction functions one has to replace the $\bar{b}_i$ with the atomic form factors f_i.

Finally, the total structure factor $S(Q)$ is calculated from $G(r)$ by

$$Q[S(Q) - 1] = \int_0^\infty G(r) \sin(Qr) \mathrm{d}r \quad . \tag{14}$$

The neutron scattering lengths can be obtained from [144].

[26] This appendix is reproduced from reference [38]

References

[1] http://science.energy.gov/~/media/ascr/pdf/benefits/Hpc_boeing_072809_a.pdf

[2] http://www.icr.ac.uk/press/recent_featured_articles/HPC/index.shtml

[3] http://www.dwd.de/

[4] Q. Yang, H. N. Koutsopoulos, M. E. Ben-Akiva, J. Trans. Res. Record **1710/2000**, 122-130 (2007).

[5] T. Sandholm. *Statistical methods for computational markets*. Doctoral Thesis ISRN SU-KTH/DSV/R-08/6-SE. Royal Institute of Technology, Stockholm, 2008.

[6] S. Elkholy and K. Meguro, 13th World Conference on Earthquake Engineering, Vancouver, Canada Aug. 1-6, Paper No. 930 (2004).

[7] Historical CPU clock rates were taken from http://en.wikipedia.org/wiki/List_of_Intel_microprocessors.

[8] J. Bolz, I. Farmer, E. Grinspun, and P. Schroder, ACM Transact. on Graph. **22**, 917 (2003).

[9] J. Georgii, R. Westermann, Sim. Model. Prac. Theory **13**, 693 (2005).

[10] TOP500 Supercomputing Sites, http://www.top500.org/lists/2010/11

[11] S. J. Plimpton, J. Comp. Phys. **117**, 1 (1995), http://lammps.sandia.gov/.

[12] M. Tatsumisago, A. Hayashi, J. Non Cryst. Solids **354**, 1411 (2008).

[13] Jing-Shi Lin and Pouyan Shen, J. Non Cryst. Solids **204**, 135 (1996);

[14] J. S. Cheng, D. H. Xiong, H. Li, and H. C. Wang, J. Alloys Comp. **507**, 531 (2010).

[15] O. H. Andersson, G. Z. Liu, K. H. Karlsson, L. Niemi, J. Miettinen, and J. Juhanoja, J. Mat. Sci.-Mat. Med. **1**, 219 (1990).

[16] Lars Winterfeld, Accelerating the molecular dynamics program LAMMPS using graphics cards' processors and the Nvidia Cuda technology http://db-thueringen.de/servlets/DocumentServlet?id=16406

[17] C. R. Trott, L. Winterfeld, and P. S. Crozier, arXiv:1009.4330v2 [cond-mat.mtrl-sci], *submitted to CPC* (2011).

[18] J.A. Anderson, C.D. Lorenz, and A. Travesset, J. Comp. Phys. **227**, 5342-5359 (2008).

[19] M. Harvey, G. Giupponi, and G. De Fabritiis, J. Chem. Theory and Comput. **5**, 1632 (2009).

[20] P. H. Colberg and F. Höfling, Comp. Phys. Comm. **182**, 1120 (2011).

[21] J.C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R.D. Skeel, L. Kale, and K. Schulten. J. Comp. Chem. **26**, 1781-1802 (2005). http://www.ks.uiuc.edu/Research/namd/

[22] D.A. Case, T.E. Cheatham, III, T. Darden, H. Gohlke, R. Luo, K.M. Merz, Jr., A. Onufriev, C. Simmerling, B. Wang and R. Woods. J. Computat. Chem. **26**, 1668-1688 (2005).

[23] W.M. Brown, P. Wang, S.J. Plimpton, and A.N. Tharrington, Comp. Phys. Comm. **182**, 898-911 (2011).

[24] Programming Guide for CUDA Toolkit 3.1.1 http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/ NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf

[25] L. Verlet, Phys. Rev. **159** 98 (1967).

[26] E.L. Pollock and J. Glosli, Comp. Phys. Comm., **95**, 93 (1996).

[27] Hockney and Eastwood, Computer Simulation Using Particles, Adam Hilger, NY (1989).

[28] G. Shainer, A. Ayoub, P. Lui, T. Liu, M. Kagan, C. Trott, G. Scantlen, and P. Crozier, Comp. Sci. R. and D. **26**, 267 (2011).

[29] Creative Consultants LLC, www.creativc.com .

[30] J. A. Board, Jr., A. John, C. W. Humphries, C. G. Lambert, W. T. Rankin and A. Y. Toukmaji, Proceedings, Eighth SIAM Conference on Parallel Processing for Scientific Computing (1997).

[31] H. Lammert, M. Kunow, and A. Heuer, Phys. Rev. Lett. **90**, 215901 (2003).

[32] C. Müller, E. Zienicke, S. Adams, J. Habasaki, and P. Maass, Phys. Rev. B **75**, 014203 (2007).

[33] A. N. Cormack, J. Du, T. R. Zeitler, Phys. Chem. Chem. Phys. **4**, 3193-3197 (2002).

[34] H. Lammert, R. Banhatti, and A. Heuer, J. Chem. Phys **131**, 224708 (2009).

[35] A. Heuer, M. Kunow, M. Vogel, and R. Banhatti, Phys. Chem. Chem. Phys. **4**, 3185 (2002).

[36] J. Habasakia, I. Okada, and Y. Hiwatari, J. Non Cryst. Solids **208**, 223 (1996).

[37] J. Habasakia, K.L. Ngai, Y. Hiwatari, and C.T. Moynihan, J. Non Cryst. Solids **349**, 223 (2004).

[38] C. R. Mueller, V. Kathriarachchi, M. Schuch, P. Maass, and V. Petkov, Phys. Chem. Chem. Phys. **12**, 10444 (2010).

[39] W. H. Zachariasen, J. Am. Chem. Soc. **54**, 3841 (1932).

[40] G. Hägg, J. Chem. Phys. **3**, 42 (1935).

[41] T.M. Alam, R.K. Brow, J. Non-Cryst. Solids **223**, 1 (1998).

[42] L. Jones, Proc. R. Soc. Lond. A **106**, 463 (1924).

[43] P. M. Morse, Phys. Rev. **34**, 57 (1929).

[44] R. A. Buckingham, Proc. R. Soc. Lond. A **168**, 264 (1938).

[45] B. Vessal, A. C. Wright, and A. C. Hannon, J. Non-Cryst. Solids **196**, 233 (1996).

[46] D. Herzbach, K. Binder,and M. H. Müser, J. Chem. Phys. **123**, 124711 (2005).

[47] L. Woodcock, C. Angell, and P. Cheeseman, J. Chem. Phys **65** [4], 1565 (1976)

[48] T. F. Soules, J. Chem. Phys. **49**, 4032 (1980).

[49] A. H. Verhoef and H. W. den Hartog, J. Non Cryst. Solids **182**, 221 and 235 (1995).

[50] S. Tsuneyuki, M. Tsukada, H. Aoki, and Y. Matsui, Phys. Rev. Let. **61** [7], 869 (1988).

[51] B. W. H. van Beest, G. J. Kramer, and R. A. van Santen, Phys. Rev. Let. **64** [16], 1955 (1988).

[52] J. Habasaki and I. Okada, Mol. Simul. **9**, 319 (1992).

[53] A. Takada, C.R.A. Catlow, and G.D. Price, J. Phys. Cond. Mat. **7**, 8693 (1995).

[54] A. Carre1, J. Horbach, S. Ispas, and W. Kob, EPL **82**, 17001 (2008).

[55] A.A. Hassanali, and S.J. Singer, J. Comp.-Aid. Mat. Des. **14**, 53 (2007).

[56] W. G. Seo and F. Tsukihashi, ISIJ International **44**, 1817 (2004).

[57] C. R. Müller, P. Johansson, M. Karlsson, P. Maass, and A. Matic, Phys. Rev. B **77**, 094116 (2008).

[58] R. Car and M. Parrinello, Phys. Rev. Lett. **55**, 2471 (1985).

[59] J. J. Liang, R. T. Cygan, and T. M. Alam, J. Non Cryst. Solids **263**, 167 (2000).

[60] A. Pedone, G. Malavasi, M. C. Menziani, A. N. Cormack, andU. Segre, J. Phys. Chem. B **110**, 11780 (2006).

[61] A. Abdel-Kader, A. A. Higazy, and M. M. Elkholy, J. Mater. Sci-Mater. El.**2**, 157 (1991).

[62] A. A. Higazy, Mater. Lett. **22**, 289 (1995).

[63] B. V. R. Chowdari and K. Radhakrishnan, J. Non-Cryst. Solids **110**, 101 (1989).

[64] M. Ganguli and K. J. Rao, J. Non-Cryst. Solids. **243**, 251 (1999).

[65] M. Abid, A. Shaim, and M. Et-tabirou, Mater. Res. Bull. **36**, 2453 (2001).

[66] B. R. V. Chowdari, K. L. Tan, and W. T. Chia, J. Non-Cryst. Solids **128**, 18 (1991).

[67] A. A. Higazy and B. Bridge, J. Non-Cryst. Solids. **72**, 81 (1985).

[68] A. P. Shtin, T. A. Khanzhina, and V. K. Slepukhin, Sov. J. Glass Phys. **11**, 298 (1985).

[69] K. Takahashi, N. Mochida, H. Matsui, S. Takeuchi, and Y. Goshi, Yog. Kyo, Shi **84**, 482 (1976).

[70] N. D. Patel, B. Bridge, and D. N. Waters, Phys. Chem. Glasses **24**, 122 (1983).

[71] J.A. van Meel, A. Arnold, D. Frenkel, Portegies, R.G. Belleman. Molec. Sim. **34**, 259 (2008).

[72] N. E. Alekseev, V. P. Gapontsev, and A. K. Gromov , Inorg. Mater+ **11**, 270 (1975).

[73] M. M. Elkholy, J. Mater. Sci-Mater. El. **6**, 404 (1995).

[74] A. A. Pronkin, I. V. Murin, I. A. Sokolov, Y. N. Ustinov, and M. N. Loseva, Glass Phys. Chem+ **23**, 383 (1997).

[75] A. Kishioka, B. Chem. Soc. Jpn. **51**, 2559 (1978).

[76] A. Denoyelle, M. J. Duclot, and J. L. Souquet, Phys. Chem. Glasses **31**, 98 (1990).

[77] L. G. Baikova, Y. K. Fedorov, and V. P. Pukh, Glass Phys. Chem+ **29**, 276 (2003).

[78] K. H. Chang, T. H. Lee, and L. G. Hwa, Chinese J. Phys. **41**, 414 (2003).

[79] U. Hoppe, E. Metwalli, R. K. Brow, and J. Neuefeind, J. Non-Cryst. Solids **57**, 13514 (2002).

[80] T. Tsuchiya and T. Moriya, J. Non-Cryst. Solids **38**, 323 (1980).

[81] R. Sistla, A. R. Dovi, and P. Su, Adv. Eng. Softw. **31**, 707 (2000).

[82] H. M. J. M. Vanass and J. M. Stevels, J. Non-Cryst. Solids **16**, 267 (1974).

[83] C. A. Click, R. K. Brow, and T. M. Alam, J. Non-Cryst. Solids **311**, 294 (2002).

[84] K. Meyer, Phys. Chem. Glasses **39**, 108 (1998).

[85] A. A. Higazy, B. Bridge, and A. Hussein, J. Acm. **86**, 1453 (1989).

[86] J.-M. Delaye and D. Ghaleb, J. Non-Cryst. Solids **195**, 239 (1996).

[87] I. A. Sokolov, A. A. Ilin, N. A. Valova et. al., Glass Phys. Chem. **29**, 300 (2003).

[88] I. A. Sokolov, Y. P. Tarlakov, N. Y. Ustinov, and A. A. Pronkin, Russ. J. Appl. Chem+ **78**, 741 (2005).

[89] M. Ashizuka, T. E. Easler, and R. C. Bradt, J. Am. Ceram. Soc. **66**, 542 (1983).

[90] W. T. Chia, B. V. R. Chowdari, and K. L. Tan, J. Mater. Sci. **28**, 3594 (1993).

[91] B. Bridge, N. D. Patel, and D. N. Waters, Phys. Status Solidi. A **77**, 655 (1983).

[92] K. Muruganandam and K. Seshasayee, Phys. Chem. Glasses **40**, 287 (1999).

[93] S. K. Akhter, Solid State Ionics **51**, 305 (1992).

[94] M. Horiuchi, T. Sei, and T. Tsuchiya, J. Non-Cryst. Solids **177**, 236 (1994).

[95] S. Inaba, S. Oda, and K. Morinaga, J. Non-Cryst. Solids **306**, 42(2002).

[96] S. Inaba, S. Oda, and K. Morinaga, J. Jpn. I. Met. **65**, 680 (2001).

[97] M. Altaf, M. A. Chaudhry, and S. A. Siddiqi, Mater. Chem. Phys. **71**, 28 (2001).

[98] J. J. Hudgens, R. K. Brow, T. R. Tallant, and S. W. Martin, J. Non-Cryst. Solids **223**, 21 (1998).

[99] K. H. Khafagy, M. A. Ewaida, A. A. Higazy et al., J. Mater. Sci. **27**, 1435 (1992).

[100] L. Shartsis, S. Spinner, and W. Capps, J. Am. Cer. Soc. **35**, 155 (1952).

[101] J. Habasaki, I. Okada, and Y. Hiwatari, J. Non-Cryst. Solids **183**, 12 (1995).

[102] R. Banhatti and A. Heuer, Phys. Chem. Chem. Phys. **3**, 5104 (2001).

[103] J. Horbach, W. Kob, and K. Binder, Chem. Geol. **174**, 87 (2001).

[104] H. Lammert, A. Heuer, Phys. Rev. B **70**, 024204 (2004).

[105] A. Heuer, H. Lammert, M. Kunow, Z. Phys. Chem. **218**, 1429(2004).

[106] J. Habasaki and Y. Hiwatari, Phys. Rev. B **69**, 144207 (2004).

[107] J. Habasaki and K. L. Ngai, J. Chem. Phys. **122**, 214725 (2005).

[108] N. Kitamura, K. Fukumi, H. Mizoguchi, M. Makihara, A. Higuchi, N. Ohno, and T. Fukunaga, J. Non Cryst. Solids **274**, 244 (2000).

[109] C. M. Schramm, B. H. W. S. de Jong, V. E.Parziale, J. Am. Chem. Soc. **106**, 4396 (1984).

[110] J. Zhao P.H. Gaskell, M.M. Cluckie, and A.K. Soper, J. Non-Cryst. Solids **232**, 721 (1998).

[111] M. Kodama, and S. Kojima, J. Therm. Anal. Cal. **69**, 961 (2002).

[112] M. Kodama, S. Feller, and M. Affatigato, J. Therm. Analys. and Cal. **57**, 787 (1999).

[113] K. Otto and M. E. Milberg, J. Am. Cer. Soc. **51**, 326 (1968).

[114] T. Matsuo, M. Shibasaki, T. Katsumata, Solid State Ionics **154 155**, 759 (2002).

[115] V. K. Michaelis, P. M. Aguiar, S. Kroeker, J. Non Cryst. Solids **353**, 2582 (2007).

[116] R. K. Brow, J. Non-Cryst. Solids **263&264**, 1 (2000).

[117] C. E. Varsamis, A. Vegiri, and E. I. Kamitsos, Phys. Rev. B **65**, 104203 (2002).

[118] L. Cormier, O. Majérus, D. R. Neuville, and G. Calas, J. Am. Ceram. Soc. **89**, 13 (2006).

[119] O. Majérus, L. Cormier, G. Calas, and B. Beuneu, Phys. Rev. B **67**, 024210 (2003).

[120] J. Swenson, L. Borjesson, and W. S. Howells, Phys. Rev. B **57**, 13514 (1998).

[121] C. E. Weir, and L. Shartsis, J. Am. Cer. Soc. **38**, 299 (1955).

[122] J. E. Shelby , J. Non-Cryst. Solids **66**, 225 (1982).

[123] A. Karthikeyan, P. Vinatier, A. Levasseur, and K.J. Rao, J. Phys. Chem. B **103**, 6185 (1999).

[124] M. Vogel, Phys. Rev. B **68**, 184301 (2003).

[125] S. W. Martin, and C. A. Angell, J. Non-Cryst. Solids **83**, 185 (1986).

[126] K. Muruganandam, M. Seshasayee, and S. Patnaik, Solid State Ionics **89**, 313 (1996).

[127] J. Swenson, A. Matic, A. Brodin, L. Borjesson, and W. S. Howells, Phys. Rev. B **58**, 11331 (1998).

[128] H. A. A. Sidek, S. P. Chow, Z. A. Talib, and S. A. Halim, Sol. St. Sci. and Tech. **11**, 103 (2003).

[129] R. Husin, Md. R. Sahar, A. S. Budi, and Z. Buang, Sol. St. Sci. and Tech. **14**, 121 (2006).

[130] D. Zielniok, C. Cramer, and H. Eckert, Chem. Mat. **19**, 3162 (2007).

[131] M. Vogel, Phys. Rev. B **70**, 094302 (2004).

[132] M. Schuch, C. R. Müller, P. Maass, and S. Martin, Phys. Rev. Lett. **102**, 145902 (2009).

[133] Daw, Baskes, Phys. Rev. Lett., **50**, 1285 (1983).

[134] Daw, Baskes, Phys. Rev. B, **29**, 6443 (1984).

[135] Shinoda, DeVane, Klein, Mol. Sim., **33**, 27 (2007).

[136] Shinoda, DeVane, Klein, Soft Matter, **4**, 2453-2462 (2008).

[137] MacKerell, Bashford, Bellott, Dunbrack, Evanseck, Field, Fischer, Gao, Guo, Ha, *et al.*, J. Phys. Chem.,**102**, 3586 (1998).

[138] Brilliantov, Spahn, Hertzsch, Poschel, Phys. Rev. E, **53**, 5382-5392 (1996).

[139] Silbert, Ertas, Grest, Halsey, Levine, Plimpton, Phys. Rev. E, **64**, 051302 (2001).

[140] Zhang and Makse, Phys. Rev. E, **72**, 011301 (2005).

[141] T.F. Soules and A.K. Varshneya, J. Am. Ceram. Soc. **64**, 145 (1981).

[142] A. El-Adawy, Science Echoes **1**, (2007).

[143] D. A. Keen, *J. Appl. Cryst.*, 2001, **34**, 172.

[144] V. F. Sears, *Neutron News*, 1992, **3**, 26, `http://www.ncnr.nist.gov/resources/n-lengths/`.