Accelerator-based Look-up Table for Coarse-grained Molecular Dynamics Computations

Prepared by:

Ananya Gangopadhyay

GNGANA001

Scientific Computing Research Unit
Department of Chemistry
University of Cape Town

Supervised by:

Prof. Kevin J. Naidoo

Scientific Computing Research Unit
Department of Chemistry
University of Cape Town

Dr. Simon Winberg

Scientific Computing Research Unit
Department of Electrical Engineering
University of Cape Town

July 2018

Dissertation presented to the University of Cape Town in fulfilment of the academic requirements for a Master of Science degree in Computational Science.

Key Words: Molecular Dynamics, Parallel Computing, Coarse-grained, GPU, LUT.

Declaration

I declare that this dissertation titles ACCELERATOR-BASED LOOK-UP TABLE FOR COARSE-GRAINED MOLECULAR DYNAMICS COMPUTATIONS, is a presentation of my original research work done at the Scientific Computing Research Unit, Department of Chemistry, University of Cape Town, South Africa. No part of this thesis has been submitted elsewhere for any other degree of qualification. Whenever contributions of others are involved, every effort is made to indicate this clearly, with due reference to the literature, and acknowledgment of collaborative research and discussions.

Name: **Ananya Gangopadhyay**

Signature: Signed by candidate **Date:** 9 July 2018

Abstract

Molecular Dynamics (MD) is a simulation technique widely used by computational chemists and biologists to simulate and observe the physical properties of a system of particles or molecules. The method provides invaluable three-dimensional structural and transport property data for macromolecules that can be used in applications such as the study of protein folding and drug design. The most time-consuming and inefficient routines in MD packages, particularly for large systems, are the ones involving the computation of intermolecular energy and forces for each molecule. Many fully atomistic systems such as CHARMM and NAMD have been refined over the years to improve their efficiency. But, simulating complex long-time events such as protein folding remains out reach for atomistic simulations. The consensus view amongst computational chemists and biologists is that the development of a coarse-grained (CG) MD package will make the long timescales required for protein folding simulations possible. The shortcoming of this method remains an inability to produce accurate dynamics and results that are comparable with atomistic simulations. It is the objective of this dissertation to develop a coarse-grained method that is computationally faster than atomistic simulations, while being dynamically accurate enough to produce structural and transport property data comparable to results from the latter.

Firstly, the accuracy of the Gay-Berne potential in modelling liquid benzene in comparison to fully atomistic simulations was investigated. Following this, the speed of a course-grained condensed phase benzene simulation employing a Gay-Berne potential was compared with that of a fully atomistic simulation. While coarse-graining algorithmically reduces the total number of particles in consideration, the execution time and efficiency scales poorly for large systems. Both fully-atomistic and coarse-grained developers have accelerated packages using high-performance parallel computing platforms such as multi-core CPU clusters, Field Programmable Gate Arrays (FPGAs) and Graphics Processing Units (GPUs). GPUs have especially gained popularity in recent years due to their massively parallel architecture on a single chip, making them a cheaper alternative to a CPU cluster. Their relatively shorter development time also gives them an advantage over FPGAs. NAMD is perhaps the most popular MD package that employs efficient use of a single GPU or a multi-GPU cluster to conduct simulations. The Scientific Computing Research Unit's in-house generalised CG code, the Free Energy Force Induced (FEFI) coarse-grained MD package, was accelerated using a GPU to investigate the achievable speed-up in comparison to the CPU algorithm. To achieve this, a

parallel version of the sequential force routine, i.e. the computation of the energy, force and torque per molecule, was developed and implemented on a GPU.

The GPU-accelerated FEFI package was then used to simulate benzene, which is almost exclusively governed by van der Waal's forces (i.e. dispersion effects), using the parameters for the Gay-Berne potential from a study by Golubkov and Ren in their work *"Generalized coarse-grained model based on point multipole and Gay-Berne potentials"*. The coarse-grained condensed phase structural properties, such as the radial and orientational distribution functions, proved to be inaccurate. Further, the transport properties such as diffusion were significantly more unsatisfactory compared to a CHARMM simulation. From this, a conclusion was reached that the Gay-Berne potential was not able to model the subtle effects of dispersion as observed in liquid benzene. In place of the analytic Gay-Berne potential, a more accurate approach would be to use a multidimensional free energy-based potential. Using the Free Energy from Adaptive Reaction Coordinate Forces (FEARCF) method, a four-dimensional Free Energy Volume (FEV) for two interacting benzene molecules was computed for liquid benzene. The focal point of this dissertation was to use this FEV as the coarse-grained interaction potential in FEFI to conduct CG simulations of condensed phase liquid benzene. The FEV can act as a numerical potential or Look-Up Table (LUT) from which the interaction energy and four partial derivatives required to compute the forces and torques can be obtained via numerical methods at each step of the CG MD simulation. A significant component of this dissertation was the development and implementation of four-dimensional LUT routines to use the FEV for accurate condensed phase coarse-grained simulations.

To compute the energy and partial derivatives between the grid points of the surface, an interpolation algorithm was required. A four-dimensional cubic B-spline interpolation was developed because of the method's superior accuracy and resistance to oscillations compared with other polynomial interpolation methods. When The algorithm's introduction into the FEFI CG MD package for CPUs exhausted the single-core CPU architecture with its large number of interpolations for each MD step. It was therefore impractical for the high throughput interpolation required for MD simulations. The 4D cubic B-spline algorithm and the LUT routine were then developed and implemented on a GPU. Following evaluation, the LUT was integrated into the FEFI MD simulation package. A FEFI CG simulation of liquid benzene was run using the 4D FEV for a benzene molecular pair as the numerical potential. The structural and transport properties outperformed the analytical Gay-Berne CG potential, more closely

approximating the atomistic predicted properties. The work done in this dissertation demonstrates the feasibility of a coarse-grained simulation using a free energy volume as a numerical potential to accurately simulate dispersion effects, a key feature needed for protein folding.

Acknowledgements

Working on this thesis has been a long and arduous journey. But it would not have been possible to reach this stage without the advice and support from my supervisors Professor Kevin Naidoo and Dr. Simon Winberg. I would like to thank them for their kindness and patience in helping me learn new, unfamiliar concepts and how to perform research effectively.

I would like to thank the Square Kilometre Array bursary fund and the NRF for their generous financial support allowing me to conduct my research comfortably.

I would also like extend my gratitude to my colleagues and fellow lab members at the Scientific Computing Research Unit: Louise Bezuidenhout, Lisl George and Lydia Dreyer for aiding me with numerous administrative tasks; Chris Barnett for help with the systems I use for research and advice on my thesis work; Tharindu Senapathi and Tomas Bruce-Chwatt for helpful discussions, advice on my work and for proofreading some of this thesis.

A huge thank you to my parents and grand-parents for their endless support, love and blessings, and to my friends, in particular David Brennan for being a pillar of support throughout this journey.

Finally, I would like to thank God for providing me the strength to make this journey.

Abbreviations

Å	Ångstrom
MD	Molecular Dynamics
CG	Coarse-grained
LJ	Lennard-Jones
vdW	Van der Waals
GB	Gay-Berne
EMP	Electrostatic Multiple Potential
G&R	Golubkov and Ren
CPU	Central Processing Unit
GPU	Graphics Processing Unit
FPGA	Field Programmable Gate Array
ASIC	Application-specific Integrated Circuit
SM	Streaming Multiprocessor
LUT	Look-up Table
FEARCF	The Free Energy From Adaptive Reaction Coordinate Forces
FEFI	Free Energy Force Induced

Table of Contents

List of Figures

List of Tables

1. Accelerating Molecular Dynamics Simulations through Coarse-graining and Parallel Computing

1.1 Introduction

Computer simulation methods are used to gain an understanding of intermolecular interactions and the overall 3-dimensional structure of the molecules in homogeneous and heterogeneous compounds and mixtures. They are a much faster alternative to an experimental approach like X-ray crystallography[1] and can provide new information that is difficult to produce in any other way. The two branches of computer simulations are Monte Carlo (MC) and Molecular Dynamics (MD). MD simulations, which this dissertation focuses on, are popular since they provide more detailed information on the dynamics of the molecules such as transport coefficients[2].

Simulations of complex condensed phase systems[3-4] and macromolecules such as proteins[5] are essential uses of MD simulations. In particular, the knowledge of protein folding is valuable since it gives insight into their 3-dimensional structure. It is a phenomenon that is difficult to determine experimentally even when the composition of the protein is known. As a result, researchers have been looking to MD simulations to observe it.

Packages such as CHARMM[6] and NAMD[7] have been used for such applications. These packages compute the mechanics of a system at atomic level considering intricate details such as bond lengths and strengths, as well as inter-atomic interactions. However, the trade-off for this level of detail is the amount of time these simulations take to execute. Full-atom simulations of such systems scale up very quickly due to the large number of molecules and interactions to be computed. For this reason, molecular-level phenomena such as protein folding cannot be observed since the timescales they take place in cannot be easily simulated.

There has been a tremendous amount of research that has gone into improving and optimizing MD simulations. A widely researched technique to achieve algorithmic speed-up in MD simulations is coarse-graining. The scale of the system is reduced by replacing an entire molecule or parts of it with coarse-grained (CG) models[8] that encapsulate the intricate atomistic detail. The ideal CG intermolecular potential must keep the effects from these

removed details. Coarse-graining reduces the scale of the problem. However, for very large systems, it can still be a time-consuming affair.

Since Moore's law has begun to no longer hold true, parallel computing has become the method of choice not only for large-scale computations but even the day-to-day PC[9]. Currently, many state-of-the-art computing systems are a result of parallel and high-performance computing. Consequently, most MD simulation packages have been parallelized to run on such systems. Each of them employs a variety of techniques, with some being more suited to specific requirements and simulated systems than others.

In this chapter, an overview of molecular dynamics simulations, the need for their parallelization and some examples of parallelized MD programs are discussed. Based on this reviewed literature and analysis, the objectives of this thesis are established.

1.2 Molecular Dynamics Simulations

In this section, a general overview of MD simulations is provided. The section also highlights the purpose of coarse-graining MD simulations, some well-known CG models and a discussion on CG intermolecular potentials.

1.2.1 General Overview

MD simulations follow an iterative algorithm, executing the same subroutine until a fixed number of iterations or steps are completed. A general, simplified routine that MD packages such as CHARMM and NAMD follow is highlighted by the flowchart in Figure 1.1. The steps in the flowchart are further described as follows:

1) **Setup simulation**: During the setup phase, the MD program creates a simulation box using parameters and data files provided by the user via console input or an input file. Input parameters include initial temperature, pressure, box size, time step length, total steps, boundary conditions and which ensemble to use (NVT or NPT). Along with the parameters, the data files provided are:
    - **Initial coordinate files**: These provide the initial positions, and in some cases velocities, of the atoms, molecules, and residues in a simulation. These are usually in the form of coordinate files or Protein Databank Files (PDB). They may also contain velocities especially in the case of a restart file.

- **Structure files**: These provide the details of the structure of the molecules and residues in the simulation including intramolecular bond definitions, bond angles, bond length, bond strengths, atomic mass and atomic charge. In CHARMM, this data is usually contained in Protein Structure Files (PSF) which are built using topology and parameter files that contain details of known residues.

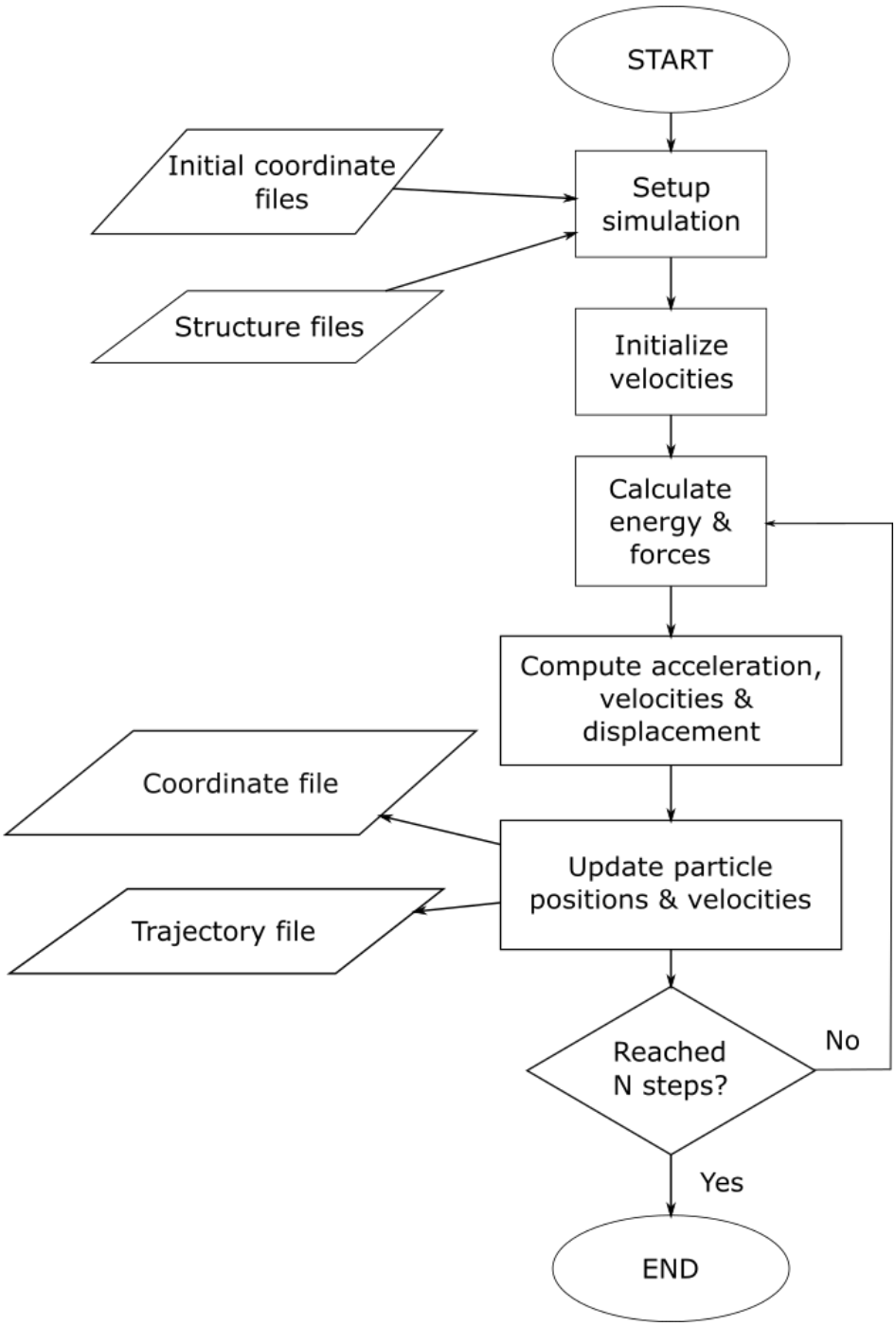


Figure 1.1: Workflow of an MD simulation. Flowchart adapted from[10-11]

2) **Initialize velocities**: Unless provided, for example from restart files, velocities of the particles in the simulation are initialized at random.
3) **Calculate energy and forces:** The force field is an analytical interatomic potential function that is used to compute energies from bonded and non-bonded interactions amongst every atom in the system. The interatomic forces on the atoms are computed from the first derivative of the force field. A detailed discussion of the force field is provided in Subsection 1.2.2.
4) **Compute acceleration, velocity, and displacement:** The computed forces, in turn, are used to compute the motion of the particles. Acceleration is computed using the forces, while the velocity and displacement are calculated using an integrator such as the Velocity Verlet[12] or Leapfrog algorithms[13]. The three computed terms define the trajectory of the particles in the system.
5) **Update particle positions and velocities:** The positions and velocities of each particle in the system are updated. The user can analyse the dynamics in the following files:
    - **Coordinate files:** At every time step, the current position of each particle in the simulation can be written out to a coordinate file. Velocities can be included in these files for use in a restart simulation.
    - **Trajectory files:** Trajectory files contain the displacement and change in velocity of the particles during the simulation. They provide vital information regarding the dynamics in the system and can be analysed using an MD package like CHARMM to gain essential information such as transport coefficients. The dynamics can also be observed using a visualization tool such as VMD[14].

1.2.2 The all-atom MD Force Field

The force field of an all-atom MD simulation is used to determine how a system will behave dynamically. A general force field[15] used in many MD packages is provided in Equation 1.1. Some force fields include additional terms.

$$U = \sum_{bonds} k_r(r - r_0)^2 + \sum_{angles} k_\theta(\theta - \theta_0)^2 + \sum_{torsions} k_\varphi[1 + \cos(n\varphi - \delta)] + \sum_{LJ} 4\varepsilon_{ij}\left[\left(\frac{\sigma_{ij}}{r_{ij}}\right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}}\right)^{6}\right] + \sum_{elect} \frac{q_i q_j}{4\pi\varepsilon_0 r_{ij}} \quad (1.1)$$

The first three terms in the equation provide the intramolecular or bonded potential and forces that exist between bonded atoms in a molecule or residue. The final two terms provide the non-bonded potential that exists between atoms, molecules and residues. The forces are derived from the first derivative of the equation.

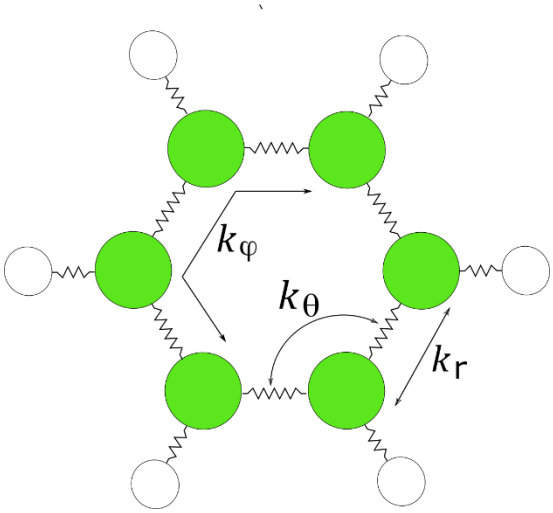


Figure 1.2: An atomistic description of benzene showing the three bonded interactions. Image derived from[16]

i. Bonded interactions

The bonded interactions in a system are amongst atoms that are covalently bonded to one another. These bonds are treated as harmonic springs and so the equations for the energies from the bond length and angles are derived from the harmonic energy equation.

$$U^{bonds} = \sum_{bonds} k_r(r - r_0)^2 \tag{1.2}$$

$$U^{angles} = \sum_{angles} k_\theta(\theta - \theta_0)^2 \tag{1.3}$$

Popular force fields such as AMBER[15, 17] and CHARMM[6, 18] contain the spring constants k_r and k_θ for various types of bonds in their parameter files. The torsion angle energy is slightly different and corresponds to the strength of the central bond within four atoms. It is not a harmonic potential.

$$U^{torsions} = \sum_{torsions} k_\varphi[1 + \cos(n\varphi - \delta)] \tag{1.4}$$

There are far fewer bonded interactions than non-bonded interactions in a system and they are much simpler to compute. However, for complex macromolecules, their computation may add to the execution time extensively.

ii. Non-bonded interactions

The calculation of intermolecular forces forms the most crucial part of a Molecular Dynamics simulations. MD simulations are N-body methods. In a simulation box of N particles, non-bonded interactions have an $O(N^2)$ complexity, which means they can be very time consuming and inefficient. This complexity is reduced in practice via cut-offs and neighbourhood lists that exclude particles that have minimal effect on the current particle. However, the improvement is usually not significant enough. The last two terms in Equation 1.1 are the non-bonded interaction terms.

The Lennard-Jones potential function

Van der Waals (vdW) forces are the weak forces present between molecules. They can exist between any pair of molecules, can be easily disrupted and effectively disappear at long distances. However, they play a major role in supramolecular chemistry and affect the solubility, polarity, and dynamics of substances. They are therefore key forces to calculate in MD simulations. Van der Waals forces are usually not affected by periodic boundary conditions since their cut-off is much shorter than the size of the simulation box.

The Lennard-Jones potential[19] is the most commonly used function to obtain the van der Waals contribution to the non-bonded forces. It is computationally very simple and fast.

$$U_{LJ}(r_{ij}) = 4\epsilon_{ij}\left[\left(\frac{\sigma_{ij}}{r_{ij}}\right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}}\right)^{6}\right] \tag{1.5}$$

The equation, which is in terms of r_{ij} i.e. the distance between the centre-of-masses of a pair of atoms, approximates an atom or a molecule as a perfect sphere.

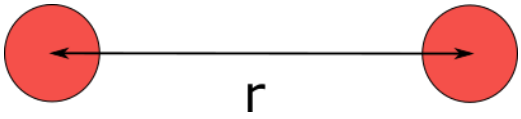


Figure 1.3: Particles approximated as spheres with distance r_{ij} between them

The equation very simply describes the interaction between two spheres as they are pulled apart or pushed together. At the equilibrium distance, σ_{ij}, the potential between a pair of spheres is 0. As r_{ij} increases from that point, the potential rapidly reduces to the lowest possible potential between two spheres; ϵ_{ij}, the potential well. At that point, the force between two spheres is 0. For increasing values of r_{ij}, the potential gradually increases and approaches 0 asymptotically. The force between the spheres in this region is attractive until a certain cut-off value where the force and energy are practically non-existent. This is the weak vdW attractive force originating from the $\left(\frac{\sigma_{ij}}{r_{ij}}\right)^6$ term. At close contact distanced smaller than the equilibrium distance, the molecules experience Pauli repulsion originating from the $\left(\frac{\sigma_{ij}}{r_{ij}}\right)^{12}$ term. The σ_{ij} and ϵ_{ij} values for an interaction are approximated experimentally.

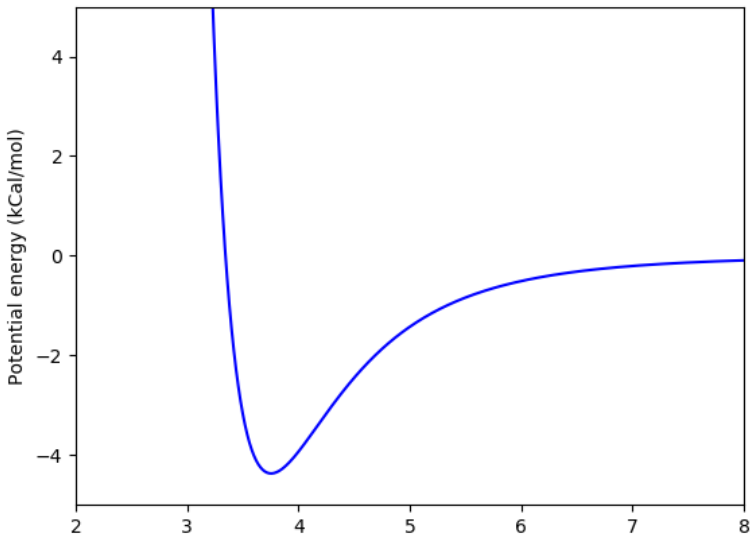


Figure 1.4: Plot of the Lennard-Jones potential for an argon-argon interaction

Due to its simplicity, the equation commonly features in the force fields of most popular MD packages. Since atoms can easily be approximated as spheres, the LJ potential is therefore used extensively in fully atomistic MD simulations.

Electrostatic forces

Electrostatic forces exist between two charged particles. They are present at distances much longer than the vdW forces occur at. Consequently, in periodic systems, periodic boundary conditions need to be considered since a particle and its images can affect one another. The electrostatic energy of a periodic system can be given by[20]:

$$U_{elec} = \frac{1}{2}\sum_{\boldsymbol{n}}\sum_{i}^{N}\sum_{j}^{N}\frac{q_i q_j}{4\pi\varepsilon_0(r_{ij}+\boldsymbol{n}L)} \tag{1.6}$$

Equation 1.6 is a modification of the electrostatics term in the force field (Equation 1.1), adapted for a periodic system. q_i and q_j are the point charges on the interacting particles i and j, while r_{ij} is the scalar distance between their centre-of-masses. $\boldsymbol{n}L$ represents a unit cubic cell and its replicas in the periodic system where L is the length of a unit cubic cell, and $\boldsymbol{n}$ is a vector value that gives the position of the cell. The $\frac{1}{2}$ term accounts for double summations. Within the unit cell, i.e. when $\boldsymbol{n} = \boldsymbol{0}$, the self-interaction term $i = j$ is excluded.

At long-range, the equation cannot be directly computed as it will not converge in real space. Due to its periodicity, the system can be transformed to a reciprocal domain using a Fourier transformation. However, this leads to a divergence at short-range in the reciprocal domain.

To tackle the convergence issue for a long-range electrostatics potential in a charge neutral system, the Ewald summations method[21] separates the equation in two parts: a short-range section which easily converges in real space and a long-range section that converges in the Fourier space. As an N-body method, classical Ewald has the computational complexity of $O(N^2)$ but can be optimized to $O\left(N^{\frac{3}{2}}\right)$. An improvement on this technique is the Particle Mesh Ewald (PME)[22] which is much more commonly used in MD simulations. In PME, the charges on the particles in the system are assigned to mesh points. With the aid of techniques such as Fast Fourier Transforms to speed-up calculations, the overall complexity of the electrostatics computations can be reduced from $O(N^2)$ to $O(N \log N)$.

iii. Computing the motion of the particles

The total force on a single atom i can be described as[2, 23-24]:

$$F_i = f_i^{bonded} + \sum_{j\neq i}^{N} f_{ij} = f_i^{bonded} - \sum_{j\neq i}^{N}\frac{dU_{ij}}{dr_{ij}} \tag{1.7}$$

In Equation 1.7, f_i^{bonded} is the total intramolecular force on atom i due to the bond length, bond angles and torsion angles. f_{ij} is the intermolecular force on i due to j. As per Newton's third law, the force on j due to i is:

$$f_{ji} = -f_{ij} \tag{1.8}$$

The intramolecular and intermolecular forces are further used to compute and update the acceleration, velocities, and positions of the molecules in the simulation. Computing the acceleration is straightforward using Newton's second law. The computations of velocities and positions, however, are more complex since they require integration with respect to time. Two techniques commonly used in MD simulations for this are the Velocity Verlet[12] and the Leapfrog[13] integration methods. They are used to integrate Newton's equations of motion to obtain the velocity and position of each particle in the system. In Velocity Verlet, by looking at the velocity at mid-interval, $\frac{\Delta t}{2}$, the motion of the particle is obtained through the following steps:

$$v_i\left(t + \frac{\Delta t}{2}\right) = v_i(t) + a_i(t)\frac{\Delta t}{2} \tag{1.9}$$

$$r_i(t + \Delta t) = r_i(t) + v_i\left(t + \frac{\Delta t}{2}\right)\Delta t \tag{1.10}$$

$$a_i(t + \Delta t) = \frac{F\left(r_i(t + \Delta t)\right)}{m_i} \tag{1.11}$$

$$v_i(t + \Delta t) = v_i\left(t + \frac{\Delta t}{2}\right) + a_i(t + \Delta t)\frac{\Delta t}{2} \tag{1.12}$$

The leapfrog method is similar however the position and velocities are computed at different times.

1.2.3 Periodic Boundary Conditions

Periodic Boundary Conditions (PBC)[2, 24] are used to simulate large, essentially infinite systems. They are used extensively in MD simulations to simulate condensed phase systems, crystals, and macromolecules in a solvent. A major motivation for their use is surface effects. For condensed phase systems, a large portion of the molecules will be at the surface of a simulation box and these molecules will experience forces that are very different to those that are not at the surface. This is especially an issue for small-scale simulations.

With PBC, an infinite system is created. A unit cell i.e. the simulation box is replicated infinitely around itself. Each of these replications is referred to as an image. When a particle leaves a simulation box, its image enters from the opposite end. Since the system is now infinite, there is no surface thus eliminating the surface effects. The minimum-imaging convention is commonly used to ensure that each particle in a system interacts with the closest image of the remaining particles.

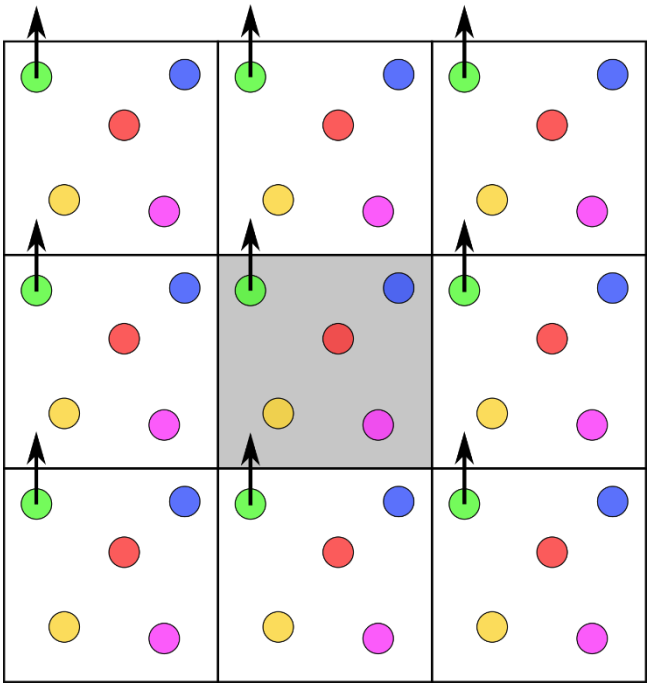


Figure 1.5: A unit simulation cell (in grey) replicated to create a periodic system. When a particle leaves a box, its image enters from the opposite end[24].

The technique does have some disadvantages. It can be considered "unnatural" for the simulation of disordered systems and have undesirable periodic effects. Special attention is also required for the characteristics of the simulation box. For examples, if it is not large enough, a macromolecule can be affected by its own image which will create effects that should not exist.

1.3 Coarse-graining MD simulations

A method of algorithmically speeding up Molecular Dynamics simulations is coarse-graining. The process involves replacing complete molecules or groups of atoms in a residue with simpler, yet similarly shaped CG models or CG "sites". As a result, the molecule and the MD system, in general, can be represented by fewer degrees of freedom as opposed to a fully atomistic model. CG simulations are therefore less complex, requiring fewer resources and thus reducing the overall execution time.

Coarse-graining has long been known as a viable option for simulating complex molecules, but it has only recently gained in popularity. In 2013, Michael Levitt, Ariel Warshel, and Martin Karplus were awarded the Nobel Prize in Chemistry and part of their award-winning research included CG models for complex proteins. This combined with current algorithmic and hardware limitations for atomistic simulations has increased the amount of research that has gone into coarse-graining.

Popular coarse-grained MD programs include ESPResSo,[25] LAAMPS[26], UNRES[27-29], and COGNAC[30]. Many fully-atomistic MD packages like NAMD can also be configured to perform CG simulations using custom CG force fields such as MARTINI[8].

1.3.1 CG Analytic models

Many CG analytic models have been developed to tackle specific issues. Some popular CG analytic models include[31]:

- **Rosetta CEM (Centroid Mode)[31-32]:** In this model, the backbone of the residue is still fully-atomistic, but the side-chains on the residue are represented by a single CG atom with characteristics such as size and charge corresponding to the side-chain.
- **CABS (Cα, Cβ, side-chain)[31, 33]:** The alpha carbon, the beta carbon, the centre of the side-chain and the centre of the peptide bond in a residue are replaced with CG atoms. Along with this, the Cα atoms are positioned on the cubic lattice in such a way that there is an additional pronounced speed-up in the calculations.
- **UNRES (United Residue)[27-29, 31]:** A highly reduced protein model that has only two interaction sites: a united side chain and a united peptide group per residue. The reduction could lead to a massive speed-up of about 1000-4000 times over an all-atom simulation on a single Alpha processor [34-35].
- **MARTINI[8, 31]:** Using a four-to-one mapping, four heavy atoms and linked hydrogens are represented by CG interaction sites which can be polar, non-polar, apolar, and charged. The CG sites are further divided into subtypes to more accurately represent the side-chains and molecules and their original atomic structure. Originally developed for lipids, it was later applied to proteins and is one of the most popular models for simulating membrane environments.

1.3.2 Challenges in coarse-graining

While coarse-graining offers a lot of advantages, with the reduced time and complexity comes a reduction in accuracy. Much of a molecule's essential information is lost, such as intramolecular bonded interactions. The CG model should not have a very low resolution since

that would remove too much of the essential structure of the molecule. A good CG system should be able to re-create the original system from the CG representation for analysis and be comparable to a fully-atomistic simulation.

Each CG interaction site needs to accurately mimic the effects on the atoms, molecules or side chains it is replacing. Since the purpose of coarse-graining is to do away with the atomistic detail, the CG force field is made up of only the non-bonded interaction potential. This analytical potential needs to be accurate enough to recover the missing atomistic information.

1.3.3 Non-bonded interaction potentials for coarse-graining

Since the LJ potential approximates particles are spheres, it is not accurate enough for models with more complex shapes. Most side-chains have complex structures that cannot be easily replaced by a sphere and may be more ellipsoidal or rod-like in shape. Further, van der Waals forces are generally anisotropic: the relative positions of the molecules affect the potential between them and the LJ potential only incorporates the scalar distance. It is, therefore, necessary to choose a potential that encompasses the anisotropy of the molecule, as well as other necessary details. This section discusses a CG model and potential that has provided decent performance for the CG simulation of benzene, as well as an alternative option which while more difficult to achieve, may provide more accurate information in CG simulations.

i. The Gay-Berne and Electrostatic Multipole (GB-EMP) model

The Gay-Berne (GB) potential function is an anisotropic variant of the Lennard-Jones potential originally used in the modelling of liquid crystals[36]. It tackles many of the issues that are present in the LJ potential and is thus more suitable for course-graining of anisotropic and complexly structured models. It has therefore been used as a CG intermolecular potential in models such as UNRES[27-29]. However, like LJ, the GB potential does not consider the effects of electrostatics between GB sites.

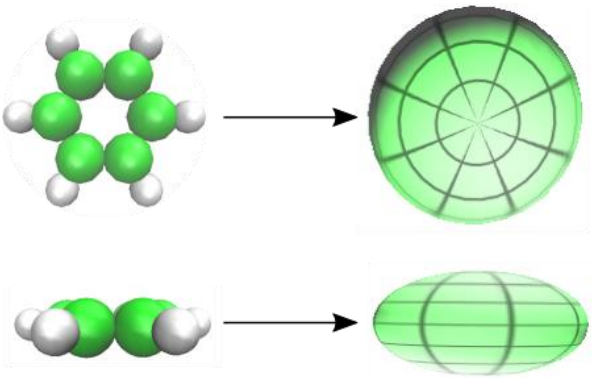


Figure 1.6: Front and side view of a Benzene molecule coarse-grained as an ellipsoidal GB-EMP site modelled in the BioVEC[37] program

A solution to this problem is suggested by Golubkov and Ren in their work[38], where the GB potential was used for its vdW contribution in a model that included Electrostatic Multipole (EMP) interactions. The system was applied to and evaluated for water, methanol, and benzene. As a dimer, benzene is especially suitable for the GB-EMP model since it can be approximated as a disk-like ellipsoid.

Gay-Berne Potential Function

The GB potential function is naturally complex, containing terms that depend on the relative orientations of the molecules[38]:

$$U_{ij}^{GB}\left(\hat{\boldsymbol{u}}_{\boldsymbol{i}}, \hat{\boldsymbol{u}}_{\boldsymbol{j}}, \vec{\boldsymbol{r}}_{\boldsymbol{ij}}\right) = 4\varepsilon_{ij}\left(\hat{\boldsymbol{u}}_{\boldsymbol{i}}, \hat{\boldsymbol{u}}_{\boldsymbol{j}}, \vec{\boldsymbol{r}}_{\boldsymbol{ij}}\right)\left[\left(\frac{\sigma_0}{r_{ij} - \sigma_{ij}\left(\hat{\boldsymbol{u}}_{\boldsymbol{i}}, \hat{\boldsymbol{u}}_{\boldsymbol{j}}, \vec{\boldsymbol{r}}_{\boldsymbol{ij}}\right) + \sigma_0}\right)^{12} - \left(\frac{\sigma_0}{r_{ij} - \sigma_{ij}\left(\hat{\boldsymbol{u}}_{\boldsymbol{i}}, \hat{\boldsymbol{u}}_{\boldsymbol{j}}, \vec{\boldsymbol{r}}_{\boldsymbol{ij}}\right) + \sigma_0}\right)^{6}\right] \quad (1.13)$$

Unlike the scalar LJ potential, the GB potential depends on the three vector parameters:

- $\vec{\boldsymbol{r}}_{\boldsymbol{ij}}$: the vector distance between the centre-of-masses of the two ellipsoids
- $\vec{\boldsymbol{u}}_{\boldsymbol{i}}$: the molecular orientation vector of the first ellipsoid
- $\vec{\boldsymbol{u}}_{\boldsymbol{j}}$: the molecular orientation vector of the second ellipsoid.

For benzene, the molecular vector runs perpendicular to the plane of the molecule. The σ_{ij} and ε_{ij} terms, which were constants in the Lennard-Jones potential function, are now complex, anisotropic functions that depend on the longitudinal and latitudinal length of the ellipsoids and their orientation-dependent well-depth respectively. The Golubkov and Ren version of the

equation also includes a damping term d_w, which is used to fine tune the potential for a given molecule.

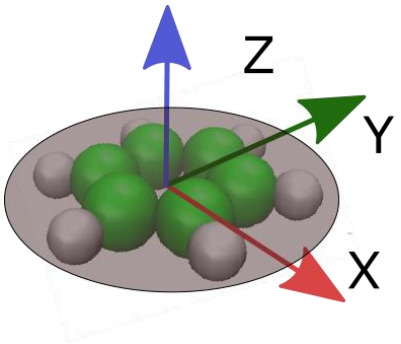


Figure 1.7: Benzene molecule on a Cartesian coordinate set. The molecular orientation vector runs parallel to the z-axis. Image derived from[38]

The equation covers three main orientations for a pair of interacting GB sites: the face-to-face, the end-to-end and the T-configurations. Each configuration has its own energy profiles with any other orientation falling within their range. The Gay-Berne energy plots are noticeably similar to the one-dimensional Lennard-Jones potential plot.

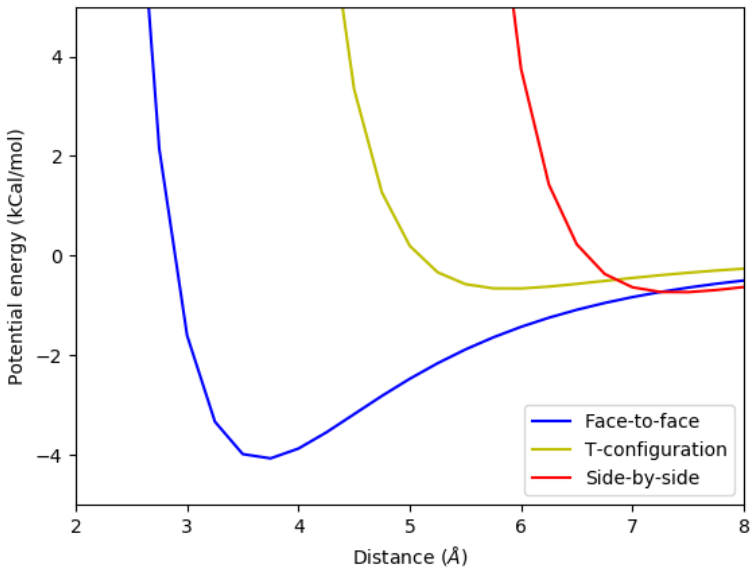


Figure 1.8: Gay-Berne potential energy vs CoM distance for each of the three main orientations - face-to-face (blue); T-configuration (yellow); end-to-end (red). Using Golubkov and Ren parameters[38]

According to their results, the work done by Golubkov and Ren has provided decent results for benzene which suits their model well, while failing to simulate water and methanol to the same degree. Furthermore, Chapter 3 covers how reproducing their work for benzene has revealed notable inaccuracies when it comes to the orientational distributions of the molecules when compared to that of a fully atomistic simulation.

Electric Multipole Potential for coarse-graining

For the Electric Multipole Potential (EMP), the charge distribution around a particle is represented by a multipole expansion[38]:

$$M = \left[q, d_x, d_y, d_z, Q_{xx}, Q_x, \dots, Q_{zy}, Q_{zz}\right] \tag{1.14}$$

The expansion is placed at the centre of mass of the particle. q gives the charge, $\vec{\boldsymbol{d}}$ the dipole moment and $\vec{\boldsymbol{Q}}$ the quadrupole moment on the particle.

Golubkov and Ren use EMP on a CG site to have a generalized and effective way to determine the electrostatics on it. This is used in conjunction with PME to obtain the electrostatic potential on a CG particle, with the Gay-Berne potential providing the van der Waals force. Gay and Berne themselves have argued that electrostatics should be considered for their model, and Golubkov and Ren have shown that the EMP is necessary to get more detail out of their model, especially for polar molecules such as methanol. Despite this, however, they were unable to replicate the atomistic results for methanol and water. Further, computing complex electrostatics adds to the performance time extensively.

ii. Using free energy as the intermolecular potential in CG simulations

The GB-EMP model is complex, however, it is still unable to display the detail necessary for the accurate simulation of simple molecules such as water and methanol. An alternative approach would be to consider coarse-graining as a bridge between macromolecular thermodynamics and atomistic detail[3].

$$\exp\left(\frac{-F}{k_B T}\right) = (const.) \int d\boldsymbol{x} exp\left[\frac{-V(\boldsymbol{x})}{k_B T}\right] \approx (const.') \int d\boldsymbol{x}_{CG} exp\left[\frac{-V(\boldsymbol{x}_{CG})}{k_B T}\right] \tag{1.15}$$

In Equation 1.15, F represents the free energy of the system i.e. its ability to do work. $V(\boldsymbol{x})$ is the potential energy of the system in terms of $\boldsymbol{x}$**,** the positions of all the atoms in the system. The CG definition of the system also fits this equation, albeit with fewer numbers of $\boldsymbol{x}$. The equation illustrates that the ideal "potential" of a CG system is the free energy surface from the all-atom description of the same system. Since mapping from an all-atom description to the CG one removes detail, free energy would contain the entropy effects arising from the degrees of

freedom that have been removed. As a result, a lot of research has gone into trying to use free energy as the intermolecular potential in CG simulations.

1.4 Parallelizing and accelerating MD simulations

The computation of both the van der Waals forces and electrostatics in non-bonded interactions is evidently computationally intensive despite the marginal improvements in computational complexity. Fully atomistic simulations conducted sequentially on a single core machine yield very slow results, and while coarse-graining offers a noteworthy speedup in execution time, further improvements in performance are necessary.

Parallel computing involves designing algorithms and writing programs that can utilise and execute on parallel architecture and memory. The aim is to obtain a considerable of speed-up in execution time and improved efficiency. Parallel computing has become an essential computing and computational tool recently as Moore's law is reaching its limit. As it has become difficult to allocate more transistors on a single chip, especially without compromising power usage, the use of multi-core machines has become inevitable in not only high-performance computing but general day-to-day applications as well.

In the domain of computational science, parallel computing is inevitably an essential aspect. It is difficult to gain important information from scientific computing work without the use of parallel computing systems due to the large amount of data that needs to be processed and the scale of the simulations that need to be executed. With recent major improvements in parallel computing, many gold standard simulation packages like CHARMM and AMBER have been parallelized, and programs such as NAMD have been created to be highly effective and scalable when running in a parallel setup.

1.4.1 Parallel platforms and models

To design the most efficient parallel algorithms, it is necessary for a computational scientist to not only make sure that a given program or algorithm is suitable for parallelism but also to identify the most suitable hardware for the task.

i. Multicore and manycore CPU systems

Since the advent of parallel computing, the most popular platform in use is a multicore system that uses linked "central" processing units (CPUs) together. The attached memory is accessed

by the cores in either a uniform or non-uniform fashion, depending on the programming model employed. Most large-scale systems are configured to run the Single Input Multiple Output (SIMD) paradigm where the same job is assigned to each core in the system. Systems like these usually employ the Message Passing Interface (MPI) paradigm[39] to assign jobs and communicate between cores. OpenMP[40] is also used, with or without MPI, for coarse-grained and fine-grained parallelism. Some manycore CPUs such as Intel's Xeon processor contain multiple processor cores on a single chip. They are used for parallel programming on desktops and clusters using similar principles and programming models as the multicore clusters.

Figure 1.9: The NEC Nehalem CPU Cluster[41†]

CPU-based parallelism is well supported by most languages and is easily accessible, given that most home computers and laptops currently contain at least two cores. However, for large industrial jobs, a large system with at least 100-200 cores is necessary. Not only is access to such a system expensive, it is also limited.

ii. FPGAs and ASICs

Field Programmable Gate Arrays (FPGAs) are reconfigurable hardware that contain numerous programmable logical units that can be interconnected with the aid of a Hardware Descriptive Language (HDL) to mould it for a given task.

† "File:Nec-cluster.jpg" by Hindermath is licensed under CC BY-SA 3.0

Figure 1.10: Spartan FPGA from Xilinx[42†]

Their flexibility and instruction level parallelism make them very popular for designing custom-made hardware such as Application-specific Integrated Circuits (ASICs). However, despite their advantages, they have a steep learning curve and a long development time. They are also not easily accessible and are incredibly expensive.

ASICs can be designed using FPGAs and can offer an even more specific hardware solution to a problem. They are usually very fast and efficient since they are specifically designed for a task and cannot be changed. But this inflexibility can be problematic when it comes to updates. Further, designing and creating ASICs is generally very expensive and it is a resource not available to everyone.

iii. Graphics Processing Units

The Graphics Processing Units (GPU) is a relatively recent addition to the parallel computing options. Popular due to their use in computer gaming, GPUs are highly specialised hardware that can be considered a small, multicore system on a single card. Highly flexible with a relatively gentle learning curve for most programmers, GPUs have become a popular choice for parallel programming.

† "File:Fpga_xilinx_spartan.jpg" by Dake is licensed under CC BY-SA 3.0

Figure 1.11: The Nvidia Tesla K20[43†]

Their popularity has pushed Nvidia and AMD to develop GPUs specifically for programming: General-purpose Graphics Processing Units (GPGPUs). Since a single GPU can perform many "large" parallel jobs, they can be a better choice than an FPGA or a multicore machine. However, there is a limited range of tasks that they are suited to.

1.4.2 Parallelization of N-body methods

In their analysis of the landscape of parallel computing, Asanović et al.[9] list N-body methods as one of their thirteen "dwarves": computational methods that are essential in scientific and engineering research and practice. These dwarves have a high-level of abstraction, however, they cover many important computational models and their potential for parallelization. Molecular Dynamics simulations are a classic example of an N-body method. N-body methods in their sequential form are quite inefficient; their time complexity is normally $O(N^2)$. However, they are highly parallelizable, and their complexity can easily be reduced to $O(N \log N)$ or even $O(N)$ with optimization and efficient parallelization.

1.4.3 Examples of accelerated MD simulations

This section contains a discussion of two examples that each illustrates how the parallelization of MD simulations provides improved performance. Each example represents the advantages and disadvantages of a fixed (ASICs) and flexible solution (multi-CPUs, GPUs and FPGAs) respectively.

† "NVIDIA Tesla K20x GPU" by GBPublic_PR is licensed under CC BY 2.0

i. The custom-built MD simulator: ANTON

While most approaches to parallelizing MD algorithms involve redesigning old algorithms to fit existing parallel hardware, D.E. Shaw Research at New York took it a step further by building a custom-built parallelized supercomputer composed entirely of ASICs. The system comprises multiple processing nodes, set up to form a three-dimensional torus. Each node contains a specialized ASIC that performs the major computations such as determination of non-bonded forces, FFT, interpolation, and integration with massive parallelization. The simulation is spread evenly across these nodes. ANTON2, the most recent iteration of the system, boasts a staggering rate of 85 μs/day for a benchmark system of 23,558 atoms.

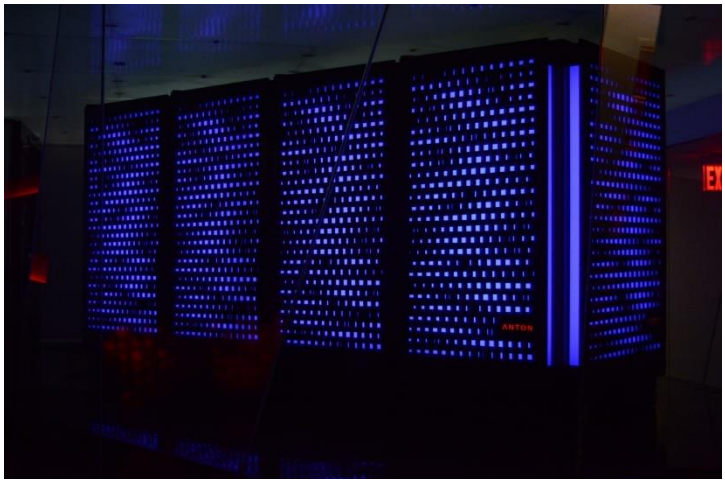


Figure 1.12: The 512-node ANTON2 supercomputer[44†]

In their research to determine the interaction networks in protein folding, Sborgi et al.[45] use NMR spectroscopy to obtain empirical data and ANTON for MD simulations. Using ANTON, the researchers managed to produce 200μs simulations and combined it with empirical data to obtain valuable information regarding protein folding.

ANTON is specific, specialized, efficient and fast. However, access to a supercomputer like ANTON is difficult, unlike software MD packages that can even run on a personal computer. ANTON is therefore generally used for long and complex fully-atomistic simulations that cannot run on commercially available parallel architecture and generalized MD packages.

† "The D.E. Shaw Supercomputer, "Anton"" by Matt Simmonsis licensed under CC BY 2.0

ii. Scalable Molecular Dynamics: NAMD

One of the most popular MD packages available, NAMD was developed by the Theoretical and Computational Biophysics Group in the Beckman Institute for Advanced Science and Technology at the University of Illinois at Urbana-Champaign. NAMD is famous as an MD simulation that is not only fast and ubiquitous but scalable to very large commodity clusters. Implemented in C++, NAMD is entirely parallelized. For CPU-based systems, the dynamic components of NAMD are built using Charm++ (from the same research group) and benefits from its features such as spatial decomposition, adaptive MPI, and load balancing. The GPU-accelerated version of NAMD is its own separate program implemented with CUDA C++ for Nvidia GPUs but follows the same general system and is further discussed in Chapter 2.

NAMD works by decomposing the simulation system into "patches" and "computes" that are executed in parallel on each processor or GPU block. Each "compute" implements a computation, namely bonded forces and non-bonded forces (generally the LJ potential). A more specialized object is created for PME calculations for the electrostatic forces.

NAMD is designed to be able to run on any system that has Ethernet or MPI. This includes both large-scale general-purpose CPU clusters, a quad-core PC or even an Nvidia GPU used for gaming. It is easy and free to obtain and offers a performance of up to 32ns/day for a large commodity cluster. While this is a very modest performance in comparison to the ANTON, this is balanced out by NAMD's ability to run on any platform.

1.5 Concluding remarks

This chapter established the purpose and challenges faced in molecular dynamics simulations, as well as the techniques of speeding them up via coarse-graining and parallelization. This sets the purpose of this project, where we would like to offer a parallel, accelerator-based, coarse-grain molecular dynamics simulation package. The package is influenced by the needs established and the examples given in the literature reviewed above.

1.6 Thesis Overview

This section will define the specific aims of this project based on the reviewed literature, the scope and limitations of the project and an overview of the layout of the thesis.

1.6.1 Purpose of the study

Molecular Dynamics simulations are essential for the simulations of condensed phase systems and macromolecules. The simulation of protein folding is a central challenge as it requires simulation times that are computationally inaccessible on most commodity hardware with the currently available software packages. The weak but important dispersion effects, along with the stronger hydrogen bonds, have been identified as the key driving forces responsible for proteins folding into repeatable three-dimensional shapes.

Coarse-grained (CG) packages invariably rely on analytical potential functions because numerical potential functions rely on look-up tables (LUTs) that are generally implemented in low dimensions. Packages running LUTs, therefore, implemented a simpler potential that depends on inter-particle distances. In this thesis, I will explore the viability of using a complex numerical potential function that is dependent on relative molecular orientation as well as distance using a multidimensional LUT.

1.6.2 Objectives of the study

The first objective of this study is to design and investigate the efficacy of a coarse-grained and parallel software solution for the determination of non-bonded interactions. A fundamental assumption in CG simulations is that the potential functions should be inspired by the free energy of the system to be investigated. The second objective of this thesis is to develop and measure the efficacy of a numerical free energy inspired potential energy function, specifically in its ability to model dispersion interactions in liquid benzene. The solution is influenced by the literature discussed and available implementations but is an alternative approach that is not commonly available. The criterion used to assign success is that the software must be fast, efficient and comparable in performance and accuracy to the available all atomistic MD packages.

1.6.3 Scope and Limitations

The two goals of the thesis are ambitious. Therefore, the primary objective of this thesis is the proof of concept: to show the proposed solution can work. While there are several software and methodological optimizations possible that will improve on the results presented here, this aspect of the larger project remains out of the scope of this thesis. However, the possible future developments are discussed in a concluding chapter at the end of this work.

1.6.4 Plan of development

The thesis is divided into 7 chapters, including the current one. The purpose of each chapter is described below:

- **Chapter 1:** Describes the purpose and challenges of molecular dynamics simulations and how they can be optimized with the aid of parallel computing. The purpose and aim of this project are established with the aid of the reviewed literature and currently available solutions.
- **Chapter 2:** Discusses the chosen parallel platform of this project: the GPU. Why it was chosen, its architecture and some examples of its use in the acceleration of MD simulations are described here.
- **Chapter 3:** Discusses the Free Energy Force Induced (FEFI) coarse-grained Molecular Dynamics package and its CGGB module that uses the Gay-Berne potential model described in Chapter 1. The package is the in-house coarse-grained molecular dynamics method of the Scientific Computing Research Unit (SCRU). The original version is entirely sequential, and a GPU-accelerated version of the Gay-Berne calculations is implemented and evaluated. The CGGB module is used to simulate liquid benzene and its successes and drawbacks are highlighted. A need for a more accurate solution is established.
- **Chapter 4:** The technique used in this project that makes it stand out as opposed to most commonly available solutions is the Look-up table (LUT). The aim is to use a LUT for the computation of non-bonded interactions so that a numerical Free Energy Volume (FEV) can be used as the intermolecular potential. This can be achieved with the aid of an interpolation algorithm. Some examples of LUTs, interpolation algorithms and their advantages and disadvantages are highlighted here.
- **Chapter 5:** The implementations and evaluation of the 4D cubic spline interpolation on the CPU and GPU are discussed. The accuracy and efficiency of the LUT are established ahead of integration into the FEFI MD package.
- **Chapter 6:** The evaluation of the LUT module integrated into the FEFI MD package is described in this chapter. The accuracy of the integrated LUT module is compared with the CGGB analytic module. The overall efficiency and execution time of the simulation are analysed. FEFI MD and its LUT module are then used to simulate liquid benzene using a FEARCF[46-48] FEV numerical potential to model benzene pairwise intermolecular interactions. Structural and transport properties are compared between the LUT-based

FEFI simulation and an all atomistic benzene liquid simulation to gauge the accuracy of the CG simulation employing a free energy based numerical pairwise potential function.

- **Chapter 7:** Provides recommendations for future work and options for features that lay beyond the scope of the dissertation.

1.7 References

1. Ladd, M. F. C.; Palmer, R. A.; Palmer, R. A., *Structure determination by X-ray crystallography*. Springer: 1985.
2. Allen, M. P., Introduction to molecular dynamics simulation. *Computational Soft Matter: From Synthetic Polymers to Proteins* **2004,** *23*, 1-28.
3. Voth, G. A., *Coarse-graining of condensed phase and biomolecular systems*. CRC press: 2008.
4. Berne, B. J.; Cicootti, G.; Coker, D. F., *Classical and quantum dynamics in condensed phase simulations*. 1998; p 880.
5. Lane, T. J.; Shukla, D.; Beauchamp, K. A.; Pande, V. S., To milliseconds and beyond: challenges in the simulation of protein folding. *Current Opinion in Structural Biology* **2013,** *23* (1), 58-65.
6. Brooks, B. R.; Bruccoleri, R. E.; Olafson, B. D.; States, D. J.; Swaminathan, S. a.; Karplus, M., CHARMM: a program for macromolecular energy, minimization, and dynamics calculations. *Journal of Computational Chemistry* **1983,** *4* (2), 187-217.
7. Nelson, M. T.; Humphrey, W.; Gursoy, A.; Dalke, A.; Kalé, L. V.; Skeel, R. D.; Schulten, K., NAMD: a parallel, object-oriented molecular dynamics program. *The International Journal of Supercomputer Applications and High Performance Computing* **1996,** *10* (4), 251-268.
8. Marrink, S. J.; Risselada, H. J.; Yefimov, S.; Tieleman, D. P.; De Vries, A. H., The MARTINI force field: coarse grained model for biomolecular simulations. *The Journal of Physical Chemistry B* **2007,** *111* (27), 7812-7824.
9. Asanovic, K.; Bodik, R.; Catanzaro, B. C.; Gebis, J. J.; Husbands, P.; Keutzer, K.; Patterson, D. A.; Plishker, W. L.; Shalf, J.; Williams, S. W. *The landscape of parallel computing research: A view from berkeley*; Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley: 2006.
10. Sonavane, Y.; Paajanen, A.; Ketoja, J.; Paavilainen, S.; Maloney, T., Molecular Dynamics Simulation of Functionalized Nanofibrillar Cellulose. In *International Conference on Theoretical Biology and Computational Chemistry*, Dubai, UAE, 2014.
11. Cai, W. L., Ju; Yip, Sidney, Molecular Dynamics. In *Comprehensive Nuclear Materials*, Stoller, R., Ed. Elsevier Science: 2010.
12. Swope, W. C.; Andersen, H. C.; Berens, P. H.; Wilson, K. R., A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: Application to small water clusters. *The Journal of Chemical Physics* **1982,** *76* (1), 637-649.
13. Van Gunsteren, W. F.; Berendsen, H., A leap-frog algorithm for stochastic dynamics. *Molecular Simulation* **1988,** *1* (3), 173-185.
14. Humphrey, W.; Dalke, A.; Schulten, K., VMD: visual molecular dynamics. *Journal of Molecular Graphics* **1996,** *14* (1), 33-38.
15. Wang, J.; Wolf, R. M.; Caldwell, J. W.; Kollman, P. A.; Case, D. A., Development and testing of a general amber force field. *Journal of Computational Chemistry* **2004,** *25* (9), 1157-1174.
16. De Wijn, A. S.; Fasolino, A., Relating chaos to deterministic diffusion of a molecule adsorbed on a surface. *Journal of Physics: Condensed Matter* **2009,** *21* (26), 264002.

17. Pearlman, D. A.; Case, D. A.; Caldwell, J. W.; Ross, W. S.; Cheatham III, T. E.; DeBolt, S.; Ferguson, D.; Seibel, G.; Kollman, P., AMBER, a package of computer programs for applying molecular mechanics, normal mode analysis, molecular dynamics and free energy calculations to simulate the structural and energetic properties of molecules. *Computer Physics Communications* **1995,** *91* (1-3), 1-41.
18. Vanommeslaeghe, K.; Hatcher, E.; Acharya, C.; Kundu, S.; Zhong, S.; Shim, J.; Darian, E.; Guvench, O.; Lopes, P.; Vorobyov, I., CHARMM general force field: A force field for drug-like molecules compatible with the CHARMM all-atom additive biological force fields. *Journal of Computational Chemistry* **2010,** *31* (4), 671-690.
19. Lennard-Jones, J. E., On the determination of molecular fields. II. From the equation of state of gas. *Proceedings of the Royal Society A* **1924,** *106*, 463-477.
20. Lee, H.; Cai, W., Ewald summation for Coulomb interactions in a periodic supercell. In *Lecture Notes, Stanford University*, 2009; Vol. 3, pp 1-12.
21. Ewald, P. P., The calculation of optical and electrostatic grid potential. *Annals of Physics* **1921,** *64* (253).
22. Darden, T.; York, D.; Pedersen, L., Particle mesh Ewald: An N· log (N) method for Ewald sums in large systems. *The Journal of Chemical Physics* **1993,** *98* (12), 10089-10092.
23. Allen, M. P.; Germano, G., Expressions for forces and torques in molecular simulations using rigid bodies. *Molecular Physics* **2006,** *104* (20-21), 3225-3235.
24. Allen, M. P.; Tildesley, D. J., *Computer simulation of liquids*. Oxford university press: 2017.
25. Arnold, A.; Lenz, O.; Kesselheim, S.; Weeber, R.; Fahrenberger, F.; Roehm, D.; Košovan, P.; Holm, C., Espresso 3.1: Molecular dynamics software for coarse-grained models. In *Meshfree Methods for Partial Differential Equations VI*, Springer: 2013; pp 1-23.
26. Plimpton, S.; Crozier, P.; Thompson, A. *LAMMPS-large-scale atomic/molecular massively parallel simulator*; 2007; p 43.
27. Liwo, A.; Pillardy, J.; Czaplewski, C.; Lee, J.; Ripoll, D. R.; Groth, M.; Rodziewicz-Motowidlo, S.; Kamierkiewicz, R.; Wawak, R. J.; Oldziej, S. In *UNRES: a united-residue force field for energy-based prediction of protein structure—orgin and significance of multibody terms*, Proceedings of the Fourth Annual International Conference on Computational Molecular Biology, ACM: 2000; pp 193-200.
28. Voth, G. A., Simulation of Protein Structure and Dynamics with the Coarse-Grained UNRES Force Field. In *Coarse-graining of Condensed Phase and Biomolecular Systems*, Voth, G. A., Ed. CRC Press: 2009; pp 108-119.
29. Makowski, M.; Liwo, A.; Maksimiak, K.; Makowska, J.; Scheraga, H. A., Simple physics-based analytical formulas for the potentials of mean force for the interaction of amino acid side chains in water. 2. Tests with simple spherical systems. *The Journal of Physical Chemistry B* **2007,** *111* (11), 2917-2924.
30. Aoyagi, T.; Sawa, F.; Shoji, T.; Fukunaga, H.; Takimoto, J.-i.; Doi, M., A general-purpose coarse-grained molecular dynamics program. *Computer Physics Communications* **2002,** *145* (2), 267-279.
31. Kmiecik, S.; Gront, D.; Kolinski, M.; Wieteska, L.; Dawid, A. E.; Kolinski, A., Coarse-grained protein models and their applications. *Chemical Reviews* **2016,** *116* (14), 7898-7936.
32. Park, H.; DiMaio, F.; Baker, D., CASP 11 refinement experiments with ROSETTA. *Proteins: Structure, Function, and Bioinformatics* **2016,** *84*, 314-322.
33. Koliński, A., Protein modeling and structure prediction with a reduced representation. *Acta Biochimica Polonica* **2004,** *51*.
34. Khalili, M.; Liwo, A.; Jagielska, A.; Scheraga, H. A., Molecular dynamics with the united-residue model of polypeptide chains. II. Langevin and Berendsen-bath dynamics and tests on model α-helical systems. *The Journal of Physical Chemistry B* **2005,** *109* (28), 13798-13810.
35. Liwo, A. UNRES - a package to carry out coarse-grained simulations of protein structure and dynamics. http://www.unres.pl/main (accessed 1-07-2018).

36. Gay, J.; Berne, B., Modification of the overlap potential to mimic a linear site–site potential. *The Journal of Chemical Physics* **1981,** *74* (6), 3316-3319.
37. Abrahamsson, E.; Plotkin, S. S., BioVEC: a program for biomolecule visualization with ellipsoidal coarse-graining. *Journal of Molecular Graphics and Modelling* **2009,** *28* (2), 140-145.
38. Golubkov, P. A.; Ren, P., Generalized coarse-grained model based on point multipole and Gay-Berne potentials. *The Journal of Chemical Physics* **2006,** *125* (6), 064103.
39. Dongarra, J. J.; Otto, S. W.; Snir, M.; Walker, D., An introduction to the MPI standard. *Communications of the ACM* **1995**, 18.
40. Dagum, L.; Menon, R., OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering* **1998,** *5* (1), 46-55.
41. Hindermath, File:Nec-cluster.jpg. Nec-cluster.jpg, Ed. 2011; pp English: The new Nehalem cluster
Deutsch: Der neue Nehalem-Cluster.
42. Dake, File: Fpga xilinx spartan.jpg. spartan.jpg, F. x., Ed. 2005; pp
Xilinx FPGA (Spartan XC3S400). The FPGA has 400000 gates, runs at 50 Mhz. See for specs. The picture is a close-up of a module used in a modular robot (http://birg.epfl.ch/page53469.html).
43. GBPublic_PR, nvidia-tesla-k20-k20x-gpu-accelerators. nvidia-tesla-k20-k20x-gpu-accelerators, Ed. 2012; p NVIDIA Tesla K20x GPU.
44. Simmons, M., The D.E. Shaw Supercomputer, "Anton". https://www.flickr.com/photos/bandman614/5348951193, Ed. www.flickr.com, 2011.
45. Sborgi, L.; Verma, A.; Piana, S.; Lindorff-Larsen, K.; Cerminara, M.; Santiveri, C. M.; Shaw, D. E.; De Alba, E.; Muñoz, V., Interaction networks in protein folding via atomic-resolution experiments and long-time-scale molecular dynamics simulations. *Journal of the American Chemical Society* **2015,** *137* (20), 6506-6516.
46. Naidoo, K. J., FEARCF a multidimensional free energy method for investigating conformational landscapes and chemical reaction mechanisms. *Science China Chemistry* **2011,** *54* (12), 1962-1973.
47. Naidoo, K. J., Multidimensional free energy volumes offer unique insights into reaction mechanisms, molecular conformation and association. *Physical Chemistry Chemical Physics* **2012,** *14* (25), 9026-9036.
48. Gamieldien, M. R.; Strümpfer, J.; Naidoo, K. J., Hydration-Determined Orientational Preferences in Aromatic Association from Benzene Dimer Free Energy Volumes. *The Journal of Physical Chemistry B* **2011,** *116* (1), 324-331.

2. Using Graphics Processing Units for MD simulations

2.1 Introduction

In Chapter 1, the purpose and issues of Molecular Dynamics (MD) simulations and their need for acceleration were highlighted. In this chapter, the Graphics Processing Unit (GPU) will be established as the most suitable platform for the purposes of this project by comparing it with the currently available parallel coprocessors.

Following a brief discussion of their history in processing graphics and general-purpose computing, the architecture of Nvidia's GPUs and the CUDA API[1-2] will be discussed. The tools available to developers to optimize their applications will be highlighted and popular examples of GPU-accelerated MD simulations will be looked at. Finally, the GPU's purpose in the context of this project will be established to be further discussed in Chapter 3.

2.2 Choosing the GPU as the accelerator

In selecting the GPU for this project, careful consideration was given to the platforms previously highlighted in Chapter 1. To achieve the aims defined for this work, the platforms were compared in terms of five requirements.

2.2.1 Programmability

The four parallel accelerators: ASICs, FPGAs, multi-CPU clusters and GPUs, can be categorized in terms of programmability. Multi-CPU clusters and GPUs are easily programmable and flexible. The core software executed on these platforms can be altered or completely replaced. FPGAs can also be reconfigured with the aid of HDLs and other frameworks such as OpenCL[3] and MATLAB[4]. ASICs, however, are inflexible circuits with only one pre-defined, specialized purpose. Flexibility allows updates to be rolled out much quicker, while in the case of ASICs a brand-new piece of hardware needs to be designed and produced every time. This gives GPUs and the other platforms an advantage, albeit at the cost of the efficiency obtained through specialization. This disadvantage can, however, be minimized through optimization and memory access management.

2.2.2 Performance and power consumption

The Nvidia Tesla V100[5-6] is one of the newest GPUs available for use in scientific computing. With up to 32GB of RAM, 900 GB/s memory bandwidth and a TDP of 250W at maximum performance, it offers high performance for low power usage. Nvidia claims that a single V100 GPU board can perform as well as up to 100 CPUs[5]. The Intel Xeon Processor E5-2690v4[7] is a CPU card specialized for cluster usage. A single processing unit has 14 cores, can support up to 1.54TB of DDR4 RAM, a memory bandwidth of 76.8GB/s and a TDP of 135W. While both platforms can provide high performance, in terms of power consumption, the single V100 GPU board outperforms 100 cores of the Xeon E5-2690v4.

The latest high-performance FPGAs come with high-end hardware components. The Virtex UltraScale+ FPGAs[5] from Xilinx, for example, boast up to 3.6M logic cells and 8GB of onboard RAM with 460 GB/s memory bandwidth amongst other specialized elements. Power usage by well-optimized FPGAs can be much lower than that by GPUs and CPU clusters[8]. However, the performance of the FPGA is highly dependent on how well it is configured and optimized. This applies to ASICs as well.

2.2.3 Development time

GPU architecture is quite dissimilar to CPU architecture since it was originally created solely for rendering graphics. Their software and hardware architecture can be difficult to grasp for new GPU programmers at first. Optimizing GPU-based programs can also be more challenging than programming for CPU clusters. However, GPU programming does use languages familiar to CPU programmers. OpenCL and CUDA both support C/C++, while CUDA also supports FORTRAN. This, along with the amount of support, tutorials and examples available can enable a programmer to learn how to program a GPU in a short amount of time. In some cases, programming GPUs can prove to be easier than working with MPI and OpenMP and provide sufficient speed-up with little development.

FPGAs are much harder to work with for beginners unfamiliar with their architecture and HDLs. Programming FPGAs requires the programmer to think at the basic hardware and instruction level. Parallelism in FPGAs is implemented at this level, and programmers unfamiliar with this technique can take some time to understand it. MATLAB and Simulink offer an easier FPGA programming experience, but it would still take longer to program an FPGA than a GPU or multi-

CPU cluster. Since ASICs are specialized hardware units, their development is complex and time-consuming and requires multiple programmers and engineers.

2.2.4 Cost-effectiveness and availability

The greatest advantage GPUs have over other platforms is their cost. A single high-end GPU usually has a high price. A 16GB Nvidia Tesla V1000, for example, can cost about $9000[9]. A single Intel Xeon E5-2690v4 processor can cost about $2,200[10]. However, the unit only has 14 cores, and about 100 cores are necessary to replicate the GPU's performance. The price of nearly 100 cores i.e. 7 processing units is about $15,575. Furthermore, the GPU board contains onboard RAM, can be installed in a workstation instead of a cluster and deliver similar performance. However, there are further costs in setting up a CPU cluster such as additional RAM and communication cables that extensively add to the total price. GPUs, therefore, have a distinct advantage when it comes to cost.

High-end FPGAs[11] can cost about $3495. Like GPUs, FPGA boards come with most peripherals such as onboard RAM and some communication ports and cables. However, the number of FPGAs required depends on the task and can cost much more than even a small CPU cluster. Since they are custom-built, it is difficult to determine the cost of an ASIC. However, as with FPGAs, their cost also depends on their task amongst other factors such as development cost and the cost of sourcing components.

It is also easier to have access to a GPU since they are generally bundled within personal computers for graphics and gaming. Nearly all Nvidia GPUs currently available are CUDA capable. Access to CPU clusters, FPGAs and ASICs is much more difficult for most programmers and developers.

2.2.5 Algorithm suitability

In their work comparing the performance of certain benchmarks on GPUs and FPGAs, Jones et al.[12] tested two benchmarks that are appropriate for this project. The tests were performed on an Nvidia GTX285 GPU and the Convey HC-1 FPGA-based system. This comparison was originally done in my previous work[13]. The expected performance of a multi-CPU cluster and an ASIC is also highlighted.

i. Batch processing

Batch processing is an important aspect of the algorithm used in this project. At each time step, the co-processor will receive the molecular information needed for the computation of the forces and torques and return the results to the main routine after the batch is processed.

Using a Mersenne Twister pseudo-random number generator, batches of random 32-bit numbers were generated. The Mersenne twister code used is custom designed for each platform. The GPU offers a speed-up of 89.3 over a single-threaded CPU, while the HC-1 performs 88.9 times better (on average). The FPGA's relatively poorer performance can be attributed to the data latency from accessing off-chip RAM since a different experiment confirmed that an FPGA can perform 8 times better than the GPU if the RAM was on-chip. On its part, the GPU has a batch processing architecture and is, therefore, more sensitive to batch size than the HC-1 that has a pipeline architecture.

A multi-CPU cluster can also function a using the MPI paradigm, even though it would need many cores to replicate a single GPU's performance. Inter-core communication within a CPU cluster can have a much higher latency than that amongst the cores of a single GPU. A high-end communication paradigm like InfiniBand can, however, considerably reduce this latency. OpenMP can also be used for more fine-grained parallelism but may add further delays due to synchronization.

A custom-built ASIC can be designed to efficiently perform the task and can outperform all the other platforms. However, it will require a longer development time than any other platform.

ii. N-Body Simulation

The other important benchmark to test is the computation of N-body calculations. As established in Chapter 1, MD simulations are N-Body methods with an $O(N^2)$ computation complexity. The platform should be able to run N-body simulations as efficiently as possible.

To test the performance of this benchmark, a two-dimensional second-order simulation of the gravitational forces between N particles of different masses was performed on each platform. The GPU required explicit synchronization by constantly stopping and restarting the deployed kernel, which negatively impacted its performance. As a result, it only offered a speed-up of

43.2 over the sequential CPU. However, it still outperformed the HC-1 which only averaged a speed-up of 1.9.

Expectedly, an FPGA with customized hardware and software can perform better than a GPU. However, this customization cannot be considered a real advantage since that would increase the development time. This also applies to an ASIC designed specifically for this problem.

The synchronization issues that this algorithm faces on GPUs can be managed by optimizing the algorithm to reduce them. A multi-CPU cluster with enough cores can be expected to have similar performance to the GPU-based implementation, depending on how well the CPU algorithm is designed. Fine-grained parallelism with OpenMP may add to or hinder the performance.

2.2.6 Conclusion

While hardware co-processors like FPGAs or ASICs can provide excellent performance, they must be explicitly designed for the task, which may include customizing the components found on your average FPGA board. ANTON, which was discussed in Chapter 1, is an example of that. Customization adds to the development time, which for most tasks is unnecessary. Especially at a stage where an algorithm is being experimented with, it is much more suitable to use a software solution on commodity hardware like GPUs and multicore CPU-clusters.

Comparing the platforms shows that GPUs suitably match all the requirements. They are easily available, can offer high performance for a relatively low cost and use much less power. Their development could be longer depending on the algorithm. However, the development time on CPU-clusters can be similarly long if not longer to ensure sufficient optimization for decent performance. Further, working with the MPI paradigm efficiently can prove to be more difficult than working with CUDA.

The GPU is, therefore, the accelerator chosen for the CG MD simulations in this project. Since the aim of this thesis is the proof of concept of how the algorithm can be accelerated and not its optimization, the GPU offers an easier programming experience than the rest of the platforms with a good potential speed-up over a single-CPU sequential implementation.

2.3 GPU history overview

Before its use in acceleration, the primary objective of a GPU was to render graphics for gaming. However, over the years due to its sophisticated architecture, its general-purpose use has gained popularity. This section offers a brief overview of the GPU's history as a graphics card and later as a general-purpose parallel processor.

2.3.1 Rendering graphics

At the end of the era of hard-coded and CPU-driven graphics rendering, the introduction of the Graphics User Interface (GUI) on PCs and the demand for more realistic gameplay pushed the development for more efficient, high-quality and on-demand video rendering. A series of video cards arrived in the market, each bringing a new feature of its own.

While the term Graphics Processing Unit (GPU) had existed for some time, it was not until 1999 that it entered common usage. The arrival of the Nvidia GeForce series in 1999 changed the landscape of graphics processing. The Nvidia GeForce 256 was marketed as "the world's first GPU"[14] and added upgrades and functionality that made it a much more popular option than its competitors. Eventually, Nvidia's GeForce series[15] and AMD's (formerly ATI) Radeon[16-17] series dominated the market and pushed out other competitors. Their rivalry spawned massive and unprecedented improvements that led to their use in areas other than their intended purpose.

2.3.2 Changing the purpose: GPGPU programming

While designing the hardware for the Xbox 360, ATI technologies introduced a formal "unified shader"[18]. Previously, GPUs had separate shaders for vertices and pixels, but as development continued and aspects such as costs became less substantial, the unified shader became more popular. ATI added this functionality to its TeraScale line[19] and later to its Radeon HD 2000 model, while Nvidia used it in its new Tesla microarchitecture and the GeForce 8 series.

The unified shader was a new feature that further pushed the graphics and gaming industry. But it had a serendipitous consequence: high-performance parallel computing. Processing high-end graphics requires a lot of processing power, and the GPUs contained this in the form of many small parallel cores. With the unified shader architecture, the shaders became programmable and capable of executing flexible instructions. The addition of a floating-point processing capability essentially made GPUs consumer-accessible stream processors.

Programmers began taking advantage of the parallelism of GPUs and started using them for non-graphics purposes, giving rise to general-purpose computing on GPUs (GPGPU)[20].

OpenGL[21] and DirectX[22] are the two most popular graphics programming frameworks for GPUs. The languages became more flexible as the architecture did, and eventually, users familiar with them and the GPU's parallel structure started manipulating them to convert non-graphical data to "graphics" and writing code such as matrix multiplication[23]. This attracted the attention of the scientific computing community since GPUs offered a cost-effective alternative to multi-CPU clusters. Parallel computing had become an essential aspect of scientific computing, especially in the case of MD simulations.

Since the graphical frameworks were not intended for non-graphical data manipulation and proved to be cumbersome, more general-purpose languages for the GPUs began to show up, such as Brook[24] and RapidMind[25].

i. Nvidia GPUs and CUDA

Recognising the popularity of GPUs for non-graphical use, Nvidia introduced the Compute Unified Device Architecture or CUDA[2]; an API that allows a user to directly access the features of the GPU for general-purpose programming without explicit knowledge of graphics-processing. While the original version of CUDA was based on C/C++, Nvidia's collaboration with the Portland Group also provided CUDA FORTRAN[26] which is more popular in the scientific computing community.

Nvidia also began producing a line of GPUs specifically for scientific computations and cluster usage: the Nvidia Tesla series[27] (not to be confused with the Tesla microarchitecture). Teslas are currently used in some of the world's most powerful supercomputers[28-29].

ii. OpenCL and AMD GPUs

Following the introduction of CUDA, which was specific to Nvidia GPUs and proprietary, the Khronos Group, a non-profit consortium that develops OpenGL, introduced OpenCL[3]. Like OpenGL, it is an open-source cross-platform programming framework based on C/C++ for parallel computing use. The group includes computing hardware giants like Nvidia, Intel and AMD and thus OpenCL is well-supported by most mainstream CPUs, GPUs and FPGAs.

The introduction of OpenCL allowed Nvidia's main rival AMD to compete with them in GPGPU. While their primary purpose is CGI and CAD development, AMD's discontinued FireStream series and the newer FirePro series[30] are stream-processors and therefore also intended for GPGPU and supercomputing.

2.4 CUDA architecture

Even though OpenCL can be considered more ubiquitous than CUDA, the lack of proprietary support for it in other languages such as FORTRAN, which is more popular in the scientific computing community, makes it the less attractive option. Furthermore, Nvidia is actively improving CUDA to remain a major player in the GPGPU industry resulting in great support. The Portland Group has a similar motivation when it comes to pushing FORTRAN as a parallel computing language.

As with CUDA, the Nvidia GPU line is also constantly being developed for GPGPU use; almost all currently available Nvidia GPUs are CUDA-capable even if their primary function is processing graphics for video game use on desktops and laptops. Unlike AMD and Intel who also produce CPUs, Nvidia focusses entirely on their GPUs resulting in them being very well supported.

In general, Nvidia GPUs and CUDA are extremely popular with many libraries, source-code and tutorials available, and easy to get started within both C/C++ and FORTRAN. They are thus the GPU brand of choice for this project. In this section, the general hardware and software architecture of the average CUDA-capable GPU is discussed.

2.4.1 Hardware architecture

Starting from the Tesla microarchitecture[31] for the GeForce 8 series, Nvidia started a process of generationally upgrading their microarchitecture. The most recently available microarchitectures are Pascal[32] and Volta[6, 33]. The general hardware architecture that most of the microarchitectures follow will be discussed in this subsection[34].

As stream processing units, GPUs consist of multiple small cores grouped together into streaming multiprocessors (SM). The SMs share the L2 cache that can be used for intercommunication between the processors and caching memory reads from the DRAM, the largest and slowest storage location that acts as the global memory. The DRAM itself has

multiple interfaces connected to the cache and SMs. The host interface provides the connection to the CPU's memory for instruction and data transfer.

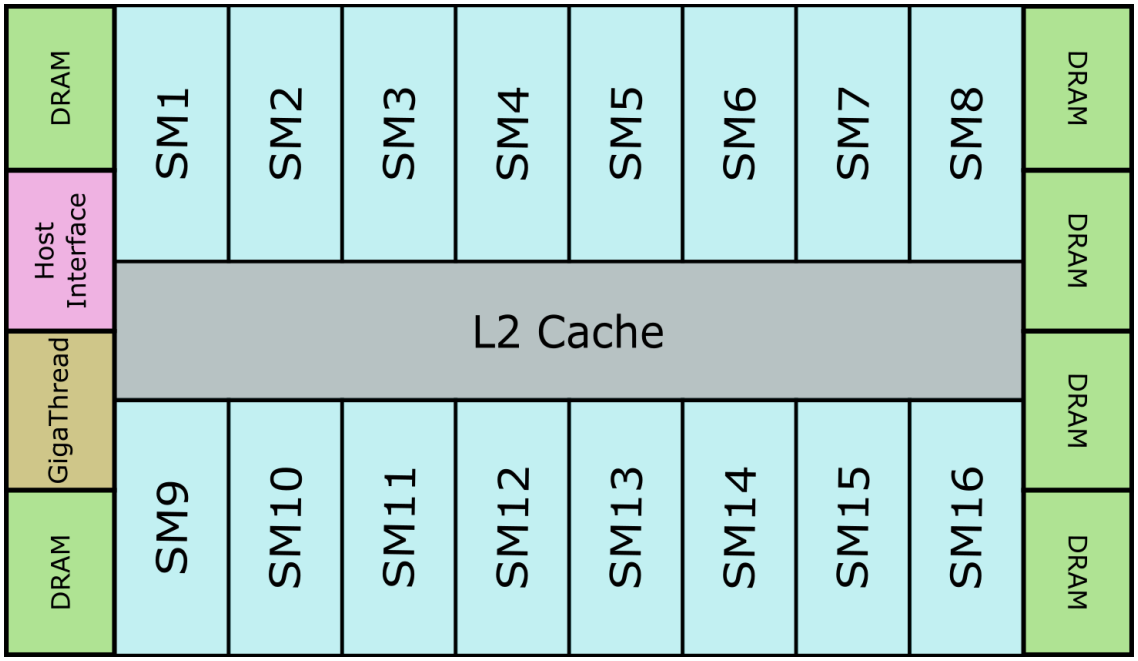


Figure 2.1: The hardware architecture of a CUDA capable GPU showing the SMs, DRAM, GigaThread scheduler and host interface. This figure was derived from the schematic of the Fermi microarchitecture from the Fermi whitepaper[35]

i. Streaming Multiprocessors (SM)

The streaming multiprocessor is an aggregate of smaller processors and units each with a specific purpose. These units are described as follows:

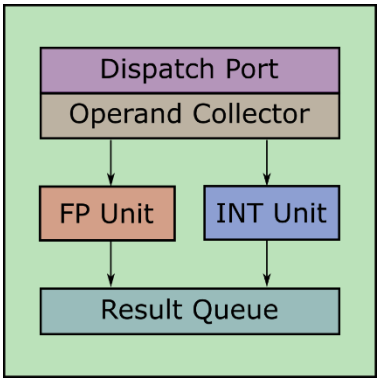


Figure 2.2: A CUDA Core[35]

- **The CUDA cores and DP units:** The CUDA or processing core is the unit that performs the basic operations on the GPU. The processing core consists of a floating-point unit that follows the IEEE 754-2008 floating-point standard and the integer unit (or ALU) to perform operations on the data it receives. It also contains queues to hold the input and the output. While the FPU originally also handled double-precision calculations, in

newer architectures like Kepler, this is handled by a separate double-precision or DP unit, with the FPU in the CUDA core remaining as the single-precision unit. As described in Figure 2.3, there are more CUDA cores present in most architectures than DP units.

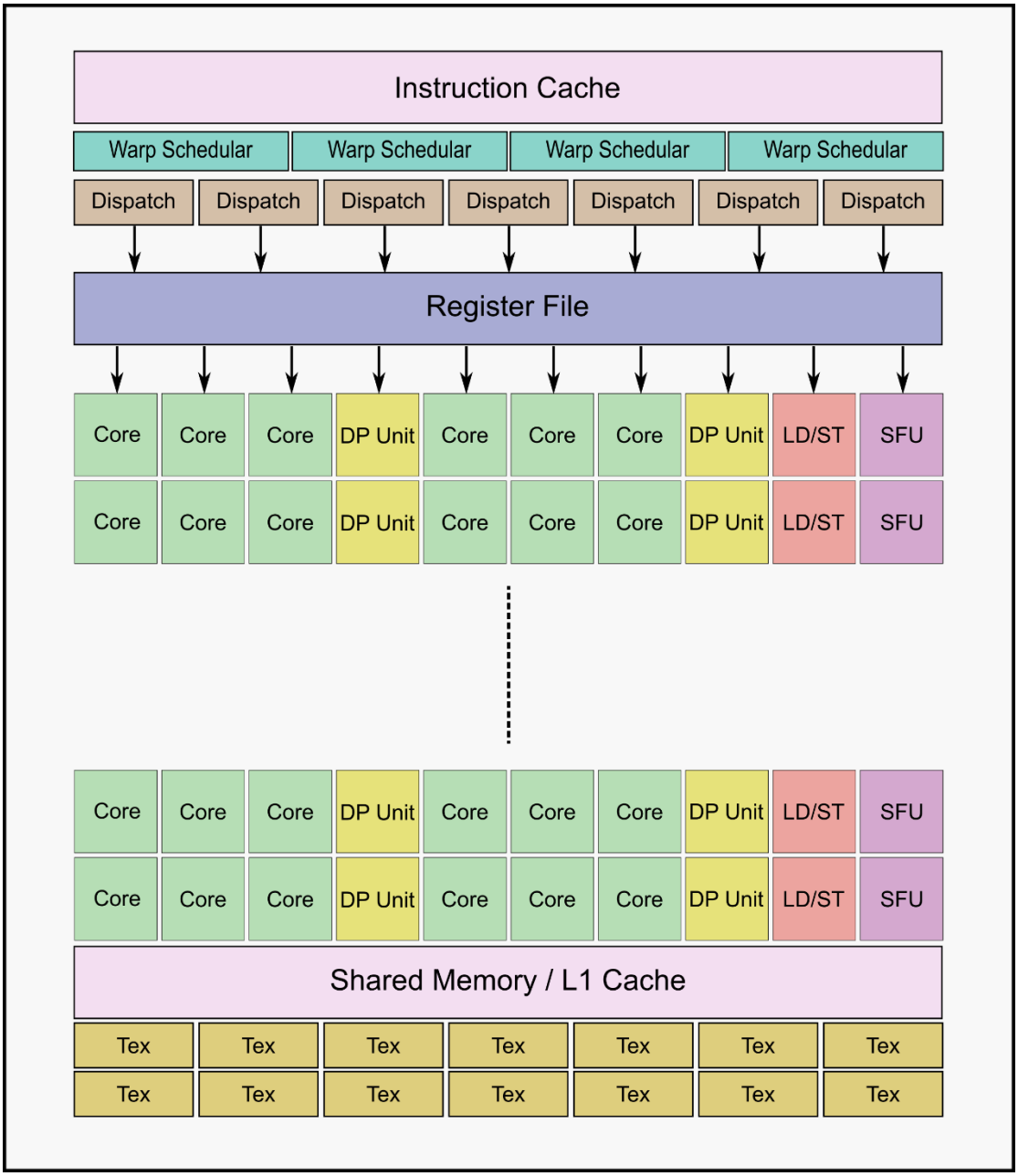


Figure 2.3: The schematic of a streaming multiprocessor. Derived from the schematics in the Fermi and Volta whitepapers[33, 35]

- **LD/ST:** The load/store units transfer instructions and data from the various memory units to the processor cores and units.
- **SFU:** The single-precision special function units are used for more complex operations such as trigonometric functions.

- **Shared memory and L1 cache:** Shared memory is the storage location for inter-core and inter-thread data. It is smaller and quicker to access than the off-chip RAM. The memory location is usually shared with the L1 cache that is used to reduce data latency from global memory reads and writes. The user can configure the size allocated to each.
- **Texture cache: A** specialized cache for speeding-up texture reads.
- **Register file:** The register file is a set of registers for local memory usage by threads. While efficient and fast, their limited capacity needs to be managed in programs to avoid overflow to local memory allocated the RAM.
- **Warp scheduler and dispatch unit:** Manages the simultaneous release of a set of instructions or threads. The size of a single warp is 32 threads.

The SMs and their components generally increase in number with each successive CUDA microarchitecture. In some cases, like the Volta microarchitecture, the cores are additionally split into two equally-sized execution blocks. The shared memory, register and cache sizes have also usually increased per release, and faster interconnects and controllers are generally introduced to reduce latency within the SM.

ii. DRAM

The DRAM, or specifically the GDDR SDRAM, is the largest memory location on the GPU board. It is located off-chip and so access to it is very slow. The L1 and L2 caches are used to reduce latency by storing frequently accessed reads and writes. In software, the DRAM acts as the global memory and as thread-specific local memory if register leakage is present. It also houses the texture memory which has more specialized interconnects and cache for faster access and linear interpolation. Recent GPUs can have onboard RAM that is up to 32GB in size.

iii. GigaThread scheduler

The GigaThread scheduler unit is a hardware scheduler that manages the threads allocated per SM. It acts as a specific point of control for the GPU threads so that the remaining processors can focus all their resources for the task at hand.

2.4.2 Software architecture

The Nvidia CUDA software architecture is very similar to the hardware architecture. Many software terms are analogous to the hardware components of the GPU. The available software features depending on the compute capability (CC) of the GPU. CC is the software “equivalent” of the device’s hardware microarchitecture and defines which CUDA features are available to

the user. It is highly dependent on the GPU model and microarchitecture. Characteristics defined by CC include the maximum number of threads per block and dimension, the maximum number of registers per thread and the amount of usable shared memory.

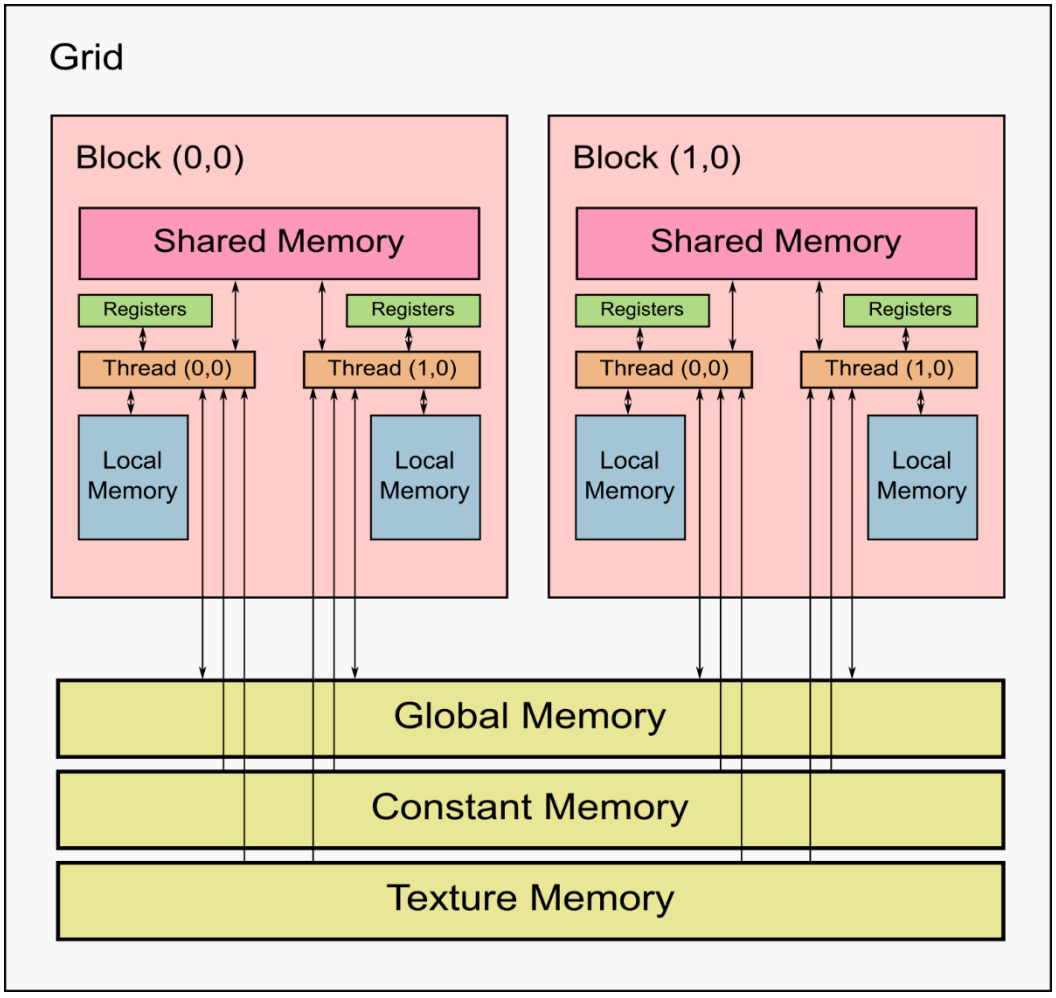


Figure 2.4: CUDA's software architecture[35]

i. Instruction or thread hierarchy

Like the hardware architecture, the software also follows a hierarchical structure. The basic component of the software hierarchy is a thread. Threads are grouped into blocks, which are further grouped into a grid. Each level in this hierarchy has different memory types available to them.

Threads

The most basic software element, a thread runs the list of instructions i.e. the kernel in parallel with other threads. GPUs follow the Single Input Multiple Thread (SIMT) paradigm where each of these threads run the exact same kernel on different data that can be accessed using the thread specific ID. Threads map to a CUDA core in terms of hardware, while making use of

special units outside of the core such as the SFU and the caches. While they ideally run in an embarrassingly parallel setup, CUDA offers semaphores and synchronization options to avoid race conditions and manage reads and writes.

Since a GPU is capable of processing 3D graphics, threads are 3-dimensional; they have IDs in the x, y and z-direction. Regardless of the dimensionality of the threads, the total number of threads per block cannot exceed 1024.

Blocks

A block is a group of threads. Threads within a block can access shared memory that threads in other blocks cannot. This allows for specific block level data manipulation and easy management of race conditions. Like threads, blocks are also three-dimensional and have IDs in each direction.

Since the GPU executes 32 threads simultaneously as a warp, it is generally recommended that block sizes are a multiple of 32. Blocks are analogous to the SMs in terms of hardware. However, a block cannot have more than 1024 threads while an SM's thread occupancy is much higher. Since blocks can be smaller containing at a minimum a single thread, SMs generally house more than one block. This is however limited by the number of registers and shared memory used within the block i.e. the SM's occupancy.

Grids

A grid is at the highest level of abstraction on the GPU. It is a collection of blocks and generally encompasses the entire GPU. A kernel launch includes a grid definition that includes the grid size: the total number of blocks, the block size: the number of threads per block, and the total number of threads. The number of threads and blocks in a grid must be provided per dimension.

ii. Memory hierarchy

As seen in the hardware architecture, the GPU houses multiple memory locations. Some of these are on-chip such as the caches and register files, while others are off-chip like the large RAM. In software terms, this forms a hierarchy of memory. The speed of access to these locations depends on if they are on-chip or off-chip and their scope of availability.

Global memory

The largest and slowest memory location, the global memory is a subsection of the off-chip RAM. Global memory is available to the entire grid of threads and so access needs to be managed to avoid any race conditions. The input and output to kernels i.e. its parameters are generally stored here. Before a kernel launch, any data and locations to be used by the kernel must be allocated on the GPU, with the input data copied to it. Since access to the memory is slow, the programmer should limit the number of reads and writes and take advantage of faster memory locations where possible.

Shared memory

Shared memory is accessible to an entire block of threads. While it is faster to access than global memory, it is also much smaller. Therefore, the block size needs to be managed to make sure that shared memory is used efficiently. Shared memory can be used as a programmer-managed cache to store frequently accessed global memory data. Since it is also available to multiple threads it must be managed with semaphores and synchronization as well. Shared memory shares its location with the L1 cache. The program can configure how much memory is allocated to each.

Registers and local memory

Registers are the fastest memory storage locations allocated to each thread. They also the smallest locations and a kernel can only have a limited number of registers per thread based on the compute capability of the device. If the kernel needs to use more local variables and memory, the registers will "overflow" and "spill" into local memory on the off-chip DRAM. While it is housed near global memory, local memory is still allocated per thread and requires no race condition management. However, like global memory, it is very slow and uses up space on the DRAM which may be otherwise needed for global data. A small amount of spillage may be necessary if the occupancy of the SM is hindering performance.

Constant and texture memory

Constant memory is a special read-only memory location. Housed on the DRAM, it is especially cached and can be read very quickly, but never written to. It is usually allocated before any kernels are launched and deallocated at the end of the program's execution. It is useful for a small amount of constant data like coefficients that need to be accessed frequently by multiple threads but do not need to be altered.

Texture memory is an even more specialized location. A staple on GPUs for image rendering, texture memory is interesting because not only does it allow for faster access via a special cache, it is specially configured for linear filtering and interpolation. Since graphics can be up till 3D in nature, texture memory can perform linear interpolation of linear values (1D), pixels (2D) and voxels (3D). In non-graphical use, texture memory is frequently used for linear interpolation in 1D, 2D and 3D, or for direct data reads for large, constant arrays. This makes it useful for tabulated data making it a kind of look-up table. At the start of a program, the data is allocated to global memory and bound to a texture, or in recent CUDA iterations and microarchitectures, texture objects.

2.5 The Nvidia Tesla K20

The development work for this thesis was done on a single Nvidia Tesla K20 board. The technical specifications obtained from the GPU's datasheets[36-37] are as follows:

Peak single-precision FP performance	3.52 TFLOPS
Peak double-precision FP performance	1.17 TFLOPS
Number of CUDA cores	2496
Processor clock	7-6 MHz
Memory clock	2.6 GHz
Memory size	5GB (GDDR5)
Memory interface	320-bit
Memory bandwidth	208 GB/s
Power consumption	225W

Table 2.1: Technical specifications of the Nvidia Tesla K20

The GPU was designed with the Kepler architecture and CC 3.5. It is intended for use in both workstations and servers. Computational chemistry is one of its many applications, making it suitable for this dissertation. The software designed for this work is in single-precision and so takes advantage of the GPU's superior single-precision performance.

2.6 Measuring GPU performance

To optimize the performance of the developed application and kernels, GPU developers must observe how well the kernels perform in terms of specific metrics. For this purpose, Nvidia offers a set of tools within its CUDA toolkit so that developers can maximize the efficiency of their software[1, 34].

2.6.1 Measurement metrics

While there are many metrics a programmer must observe to ensure their application is running efficiently, these three metrics provide the most detail regarding the performance of the kernels in a GPU-based application.

i. SM occupancy and utilization

The occupancy of an SM is the ratio of the number of warps active on an SM and the maximum number of warps the SM is theoretically capable of executing. A high occupancy is essential to hide latencies between instructions that depend on each other. Generally, a 50% occupancy is enough. Increasing it further will not increase the performance and may in fact further hinder it.

The theoretical occupancy of the kernel is the maximum occupancy the kernel can achieve based on metrics such as shared memory usage, registers per thread, and the number of threads per block. Altering these by computing the occupancy and testing it experimentally can provide ideal performance. It is essential to ensure that any improvements in occupancy are not heavily outweighed by lower performance in other metrics.

ii. Memory and instruction latency

The GPU is affected by instruction and memory latency when the kernel's design causes it to not have enough jobs to operate continuously. Latency in instruction execution arises when threads within a warp have stalled due to user-assigned conditions. As a result, programmers are encouraged to write kernel code with very few conditional statements. Other instruction latencies include delays in instruction fetches, execution dependency and synchronization.

Memory dependencies occur when the resources the GPU needs for execution are not available. Memory throttle can also limit performance by halting execution until multiple stalled memory

transactions are completed. Poor caching for constant and texture memory could also delay the execution.

These latencies can be observed using the Nvidia Profiler and can be improved using suggestions by the profiler and other resources.

iii. Functional unit utilization and instruction counts

Execution of the kernels can also be affected by poor utilization of the functional units in the GPU. A high number of load, store and texture operations can extensively add to the latency. Instruction counts can give an indication of which operations are impacting the kernel execution. They also include a count of inactive threads which shows how much divergence in the kernel is impacting the execution of the warps.

2.6.2 Measurement tools

The following measurement tools can be installed with the CUDA toolkit[1]. Through their efficient use to observe and manage performance metrics, developers can maximize the efficiency of their applications.

i. Nvidia Profiler

The Nvidia profiler is one of the most essential tools in the toolkit. Available as both a command-line operation as well as a visual application, the profiler offers developers all the essential metrics listed in Subsection 2.6.1. The visual profiler is available on all operating systems. It can be used to create a timeline of all the kernels in the application that allow the developer to observe bottlenecks and identify areas where further optimizations can be applied. The profiler itself offers suggestions on which metrics should be improved to increase performance. The developer can also observe the time consumed by API calls and come up with solutions to overlap them to increase efficiency. The source-code can also be observed within the profiler to identify opportunities to improve instruction level parallelism.

ii. CUDA-MEMCHECK and CUDA-GDB

CUDA-MEMCHECK is a powerful command-line tool that can be used to observe memory issues within the software. The tool can pick up issues within the code including memory leaks and illegal access of uninitialized, out-of-bounds or misaligned memory. Potential race conditions resulting from improper access and improper usage of synchronization tools can also be detected. If the GPU encounters any hardware exceptions when running an application, CUDA-

MEMCHECK can provide details on that as well. The tool is very similar to Valgrind[38] used for profiling memory leaks in CPU programs. However, CUDA-MEMCHECK is specialized for CUDA-capable GPU usage.

CUDA-GDB is a command-line debugging tool that is an extension of the GNU Project debugger, GDB. Through this tool, programmers can debug programs running on an actual GPU instead of observing the issues via emulation or simulation.

Each of the two tools can be used on their own or run together in an integrated manner. Together, the two tools can allow programmers to solve issues within their application while running them on a GPU in real-time.

iii. nvidia-smi

For most GPUs with Fermi or higher microarchitectures, nvidia-smi allows the user to obtain details on the GPU such as its model, compute capability and average power usage. It can also be used to monitor and manage any issues in the GPU. Like the Nvidia Visual Profiler, it is a cross-platform tool.

2.7 Examples of GPU-accelerated MD simulations

After GPGPU gained popularity and the introduction of OpenCL and CUDA, many developers of popular parallelized MD packages started experimenting with GPUs. With their highly parallelized structure, GPUs essentially acted as small multicore systems on a single card that can offer the same kind of functionality as a modestly-sized CPU cluster.

Below are some popular examples of MD packages that use GPUs. Some are re-worked from a CPU-cluster based algorithm while others were created specifically for the GPU.

2.7.1 OpenMM

OpenMM[39-40] is an open-source toolkit designed to act as a stand-alone package, a library and even a domain-specific language for MD simulations. The package contains almost any tool to run a simulation and can be imported into a separate simulation package to add parallelism to it.

OpenMM is flexible in terms of platforms and can run on CPUs and both Nvidia and AMD GPUs. It uses OpenCL and CUDA to run efficient, predefined algorithms for bonded and non-bonded potentials such as the standard Lennard-Jones and even the Gay-Berne potential. It can also create on-demand kernels to run user-provided custom potentials. GPU usage for this feature is recommended since there is almost no difference in performance from standard potentials.

As a library, OpenMM is used to provide GPU-acceleration to many existing "gold standard" MD packages such as CHARMM[41] and GROMACS[42].

2.7.2 GROMACS

GROMACS was originally designed to run on multiple CPUs using a combination of MPI and OpenMP[43]. Starting from version 4.5, GROMACS started including GPU acceleration. GROMACS 4.5 was entirely implemented on the GPU using the OpenMM library[44]. The CPU was only used for the purposes of input and output. However, the implementation did not offer a lot native GROMACS features and did not produce considerable speed-up as opposed to parallelized CPU implementations for most simulations.

However, GROMACS 4.6[45] offered a hybridized solution, using native CUDA-based GPU acceleration and well as OpenMP on the CPU. The entire algorithm and system were specifically re-designed for this purpose. This solution allowed for bonded interactions and electrostatics to be computed on the CPU and van der Waals interactions to be offloaded and computed on the GPU simultaneously. The package also supported GROMACS's native features and offers multi-GPU support.

Since GROMACS 5.0[46], non-Nvidia GPUs such as AMD GPUs are also supported via OpenCL. GPU-acceleration has now become an essential part of GROMACS, but the user does have the option to switch it off if no GPU is present. Currently, the GPU only computes the vdW part of the non-bonded interactions on the GPU, while the electrostatics are computed on the CPU. However, this will be changed from the next version.

2.7.3 LAAMPS

LAAMPS[47-48] has configured many of its "pair styles" for GPU execution. Like GROMACS, LAAMPS also employs a hybrid solution taking advantage of the both the GPU and the CPU. The GPUs and CPUs are configured to be able to work in a multi-GPU/multi-CPU hybrid cluster environment.

Atom coordinates are sent to the GPU while the forces are returned to the CPU. Neighbour-lists can be set up and updated on either platform based on the user's choice. For the electrostatics computations, the Fast Fourier Transformation (FFT) requires MPI communication in this package and is thus performed on the CPUs. However, charge assignment and force interpolation are performed on the GPU. Asynchronous calculations of other forces are performed on both platforms.

The speed-up and performance of the GPU-accelerated setup depend on the kind of pair-style used and the model of the platform. In general, for single GPU and CPU comparisons, the GPU implementation clearly outclasses the CPU implementation. However, the speed-up over a cluster setup is more modest. Ultimately, depending on the task, the GPU acceleration can provide a notable boost in performance.

2.7.4 NAMD

Even when it comes to GPU acceleration, NAMD[49-50] is still considered to be one of the best packages available. Not long after the launch of CUDA, NAMD developers started to experiment with the execution of essential MD algorithms on the GPUs such as Direct Coulomb summations, multilevel coulomb summations, the computation of the Lennard-Jones potential and close-ranged electrostatics.

This led to the formulation of efficient GPU algorithms for not only non-bonded interactions but for every aspect of an MD simulation. Many optimization techniques are employed, such as the use of texture memory (or texture objects in recent implementations) for the interpolation of pre-calculated values of energy, force and coefficients stored as a look-up table[51-52]. Shared memory is configured to be used as an extra cache and the CPU controls the GPU's workload. The CUDA version is multi-GPU and implements NAMD's spatial decomposition with each simulation "patch" assigned to a single GPU. The CPU manages the inter-patch communication.

The CUDA-based NAMD package is entirely self-contained and designed specifically for GPU-based execution. The Charm++[53] version also offers some CUDA-accelerated features but not to the extent of the full version.

2.7.5 GALAMOST

Created by the Changchun Institute of Applied Chemistry, the GPU-Accelerated Large-Scale Molecular Simulation Toolkit (GALAMOST)[54] is a new package created to take advantage of powerful Nvidia GPUs. It offers a variety of features, including anisotropic coarse-graining using the Gay-Berne potential function. Implemented using CUDA, GALAMOST executes major MD calculations such as the computation of both bonded and non-bonded forces on the GPU. The neighbour-list is also set up and updated using the GPU. These algorithms are specifically optimized for GPU execution and lead to a reduction in execution time since latency resulting from the data transfer between the host's memory and the device's memory is avoided. The code is configured to run efficiently on a single GPU card with each thread focussing on the interactions and forces on a single atom.

2.8 Concluding Remarks

With their availability, flexibility and potential for providing a considerable performance boost, it is evident why GPUs have been used extensively for MD simulations. For the reasons highlighted in this chapter, they are the platform of choice for the acceleration of the Free Energy Force Induce (FEFI) CG MD package. The GPU-based acceleration of the Gay-Berne potential based force routine of the package, as well as its accuracy in simulating liquid benzene, will be covered in Chapter 3.

2.9 References

1. NVIDIA Corporation CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (accessed 25-06-2018).
2. NVIDIA Corporation, NVIDIA CUDA Compute Unified Device Architecture In *Programming Guide*, 2007; Vol. 2018.
3. Stone, J. E.; Gohara, D.; Shi, G., OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering* **2010,** *12* (3), 66-73.
4. Release, M. *The MathWorks Inc.*, Natick, Massachusetts, United States, 2013.
5. NVIDIA Corporation, NVIDIA TESLA V100 GPU Accelerator - Datasheet. 2018.
6. NVIDIA Corporation *TESLA V100 Performance Guide*; 2018.
7. INTEL® XEON® PROCESSOR E5-2690 V4. Intel Corporation: 2016.
8. Grozea, C.; Bankovic, Z.; Laskov, P. *FPGA vs. Multi-Core CPUs vs. GPUs: Hands-on Experience with a Sorting Application*.
9. Amazon.com, I. Amazon.com: Nvidia Tesla v100 16GB. https://www.amazon.com/PNY-TCSV100MPCIE-PB-Nvidia-Tesla-v100/dp/B076P84525/ref=sr_1_2?ie=UTF8&qid=1531187678&sr=8-2&keywords=nvidia+tesla+v100 (accessed 03-07-2018).

10. Amazon.com, I. Amazon.com: NTEL XEON 14 CORE PROCESSOR E5-2690V4 2.6GHZ 35MB SMART CACHE 9.6 GT/S QPI TDP 135W. https://www.amazon.com/INTEL-PROCESSOR-E5-2690V4-2-6GHZ-SMART/dp/B01DTYPFQC/ref=sr_1_2?s=electronics&ie=UTF8&qid=1531187814&sr=1-2&keywords=Intel+Xeon+E5-2690v4 (accessed 03-06-2018).
11. Xilinx Virtex-7 FPGA VC707 Evaluation Kit. 2018.
12. Jones, D. H.; Powell, A.; Bouganis, C.-S.; Cheung, P. Y. In *GPU versus FPGA for high productivity computing*, Field Programmable Logic and Applications (FPL), 2010 International Conference on, IEEE: 2010; pp 119-124.
13. Gangopadhyay, A. A GPU-based Look-up table for non-bonded interactions in Molecular Dynamics simulations. Bachelor's Thesis, University of Cape Town, Cape Town, 2015.
14. NVIDIA Corporation Graphics Processing Unit (GPU). http://www.nvidia.com/object/gpu.html (accessed 1-07-2018).
15. NVIDIA Corporation GeForce.com: Official Site GTX Graphics Cards, VR, Gaming, Laptops. https://www.nvidia.com/en-us/geforce/.
16. Mantor, M. In *AMD Radeon™ HD 7970 with graphics core next (GCN) architecture*, Hot Chips 24 Symposium (HCS), 2012 IEEE, IEEE: 2012; pp 1-35.
17. Advanced Micro Devices, I. RadeonTM RX Series Graphics Card | AMD. https://www.amd.com/en/RX-series (accessed 1-09-2018).
18. Andrews, J.; Baker, N., Xbox 360 system architecture. *IEEE Micro* **2006,** *26* (2), 25-37.
19. Houston, M., Anatomy of AMD's TeraScale Graphics Engine. 2008.
20. GPGPU.org gpgpu.org. http://gpgpu.org/ (accessed 25-06-2018).
21. Woo, M.; Neider, J.; Davis, T.; Shreiner, D., *OpenGL programming guide: the official guide to learning OpenGL, version 1.2*. Addison-Wesley Longman Publishing Co., Inc.: 1999.
22. Gray, K., *Microsoft DirectX 9 programmable graphics pipeline*. Microsoft Press: 2003.
23. Larsen, E. S.; McAllister, D. In *Fast matrix multiplies using graphics hardware*, Proceedings of the 2001 ACM/IEEE Conference on Supercomputing, ACM: 2001; pp 55-55.
24. Buck, I.; Foley, T.; Horn, D.; Sugerman, J.; Fatahalian, K.; Houston, M.; Hanrahan, P. In *Brook for GPUs: stream computing on graphics hardware*, ACM Transactions on Graphics, ACM: 2004; pp 777-786.
25. McCool, M. D. In *Data-parallel programming on the Cell BE and the GPU using the RapidMind development platform*, the GSPx Multicore Applications Conference, 2006, 2006.
26. Ruetsch, G.; Fatica, M., *CUDA Fortran for scientists and engineers: best practices for efficient CUDA Fortran programming*. Elsevier: 2013.
27. NVIDIA Corporation NVIDIA Tesla Supercomputing | NVIDIA. https://www.nvidia.com/en-us/data-center/tesla/.
28. Ohio Supercomputer Center. 1987.
29. Shimokawabe, T.; Aoki, T.; Ishida, J.; Kawano, K.; Muroi, C., 145 TFlops performance on 3990 GPUs of TSUBAME 2.0 supercomputer for an operational weather prediction. *Procedia Computer Science* **2011,** *4*, 1535-1544.
30. Advanced Micro Devices, I., FirePro™ Professional Workstation Graphics Cards | AMD. **2018**.
31. Lindholm, E.; Nickolls, J.; Oberman, S.; Montrym, J., NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro* **2008,** *28* (2).
32. NVIDIA Corporation *P100 GPU Accelerator*; 2016.
33. NVIDIA Corporation *NVIDIA TESLA V100 GPU ARCHITECTURE*; 2018.
34. NVIDIA Corporation, NVIDIA CUDA. In *Programming guide*, 2010.
35. NVIDIA Corporation, Fermi Whitepaper. Nvidia Corporation: 2012.
36. NVIDIA Corporation *K20-K20X GPU Accelerators Benchmarks*; 2012.
37. NVIDIA Corporation, Tesla® Kepler™ GPU Accelerators. 2012.

38. Nethercote, N.; Seward, J. In *Valgrind: a framework for heavyweight dynamic binary instrumentation*, ACM Sigplan Notices, ACM: 2007; pp 89-100.
39. Eastman, P.; Pande, V., OpenMM: a hardware-independent framework for molecular simulations. *Computing in Science & Engineering* **2010,** *12* (4), 34-39.
40. University, O. t. S. OpenMM Users Manual and Theory Guide. http://docs.openmm.org/latest/userguide/index.html.
41. CHARMM OpenMM - c40b1 - CHARMM. https://www.charmm.org/charmm/documentation/by-version/c40b1/params/doc/openmm/ (accessed 20-06-2018).
42. Goga, N.; Marrink, S.; Cioromela, R.; Moldoveanu, F. In *GPU-SD and DPD parallelization for Gromacs tools for molecular dynamics simulations*, Bioinformatics & Bioengineering (BIBE), 2012 IEEE 12th International Conference on, IEEE: 2012; pp 251-254.
43. Berendsen, H. J.; van der Spoel, D.; van Drunen, R., GROMACS: a message-passing parallel molecular dynamics implementation. *Computer Physics Communications* **1995,** *91* (1-3), 43-56.
44. Pronk, S.; Páll, S.; Schulz, R.; Larsson, P.; Bjelkmar, P.; Apostolov, R.; Shirts, M. R.; Smith, J. C.; Kasson, P. M.; Van Der Spoel, D., GROMACS 4.5: a high-throughput and highly parallel open source molecular simulation toolkit. *Bioinformatics* **2013,** *29* (7), 845-854.
45. Abraham, M. J.; Murtola, T.; Schulz, R.; Páll, S.; Smith, J. C.; Hess, B.; Lindahl, E., GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX* **2015,** *1*, 19-25.
46. Abraham, M.; Van Der Spoel, D.; Lindahl, E.; Hess, B., The GROMACS development team GROMACS user manual version 5.0. 4. *Journal of Molecular Modeling* **2014**.
47. Plimpton, S.; Crozier, P.; Thompson, A. *LAMMPS-large-scale atomic/molecular massively parallel simulator*; 2007; p 43.
48. Sandia Corporation LAMMPS Documentation. http://lammps.sandia.gov/doc/Manual.html.
49. Nelson, M. T.; Humphrey, W.; Gursoy, A.; Dalke, A.; Kalé, L. V.; Skeel, R. D.; Schulten, K., NAMD: a parallel, object-oriented molecular dynamics program. *The International Journal of Supercomputer Applications and High Performance Computing* **1996,** *10* (4), 251-268.
50. Stone, J. E.; Phillips, J. C.; Freddolino, P. L.; Hardy, D. J.; Trabuco, L. G.; Schulten, K., Accelerating molecular modeling applications with graphics processors. *Journal of Computational Chemistry* **2007,** *28* (16), 2618-2640.
51. Rodrigues, C. I.; Hardy, D. J.; Stone, J. E.; Schulten, K.; Hwu, W.-M. W. In *GPU acceleration of cutoff pair potentials for molecular modeling applications*, Proceedings of the 5th Conference on Computing Frontiers, ACM: 2008; pp 273-282.
52. Hardy, D. J., GPU Particle-Particle Algorithms: Non-bonded Force Calculation. 2013.
53. Kale, L. V.; Krishnan, S. In *CHARM++: a portable concurrent object oriented system based on C++*, ACM Sigplan Notices, ACM: 1993; pp 91-108.
54. Zhu, Y. L.; Liu, H.; Li, Z. W.; Qian, H. J.; Milano, G.; Lu, Z. Y., GALAMOST: GPU-accelerated large-scale molecular simulation toolkit. *Journal of Computational Chemistry* **2013,** *34* (25), 2197-2211.

3. GPU-Accelerated Coarse-grained Benzene Simulations using the Gay-Berne Potential

3.1 Introduction

In the preceding chapters, the need for faster, accurate and efficient molecular dynamics simulations was highlighted and the GPU's suitability for their acceleration was established. The objective of this thesis is to find faster ways to simulate benzene molecules using a coarse-grained (CG) setup. The challenge, however, is to minimize the loss of essential atom level information in the process.

The Gay-Berne potential[1] is an anisotropic variant of the Lennard-Jones potential[2] that has been mostly used to model liquid crystals. Attempts to fit the Gay-Berne potential to monomeric molecules that are building blocks of or represent essential elements in macromolecular systems have been made by several researchers. Golubkov and Ren[3], for example, fit the potential to benzene, water and methanol with benzene providing the best results. Sheraga et al. used a variant of the Gay-Berne potential for their UNRES model and package for CG simulations of proteins[4-6].

This chapter will cover an in-depth look at the coarse-grained Gay-Berne (CGGB) module: the energy and force routines implemented within the Free Energy Force Induced (FEFI) generalised CG molecular dynamics package created by SCRU[7]. The CGGB module uses the Gay-Berne analytical function to determine the non-bonded interaction energies and forces. The module had previously been implemented for sequential computations on a CPU. A parallelized GPU version of the module was created to accelerate the CG dynamics. This is discussed in detail in this chapter. Finally, the performance of the Gay-Berne potential with the Golubkov and Ren parameters will be analysed in terms of accuracy with respect to a fully-atomistic CHARMM simulation.

3.2 Simulating benzene in vacuum

The FEFI package is used to simulate liquid benzene. Its energy and force routines that implement the coarse-grained Gay-Berne analytical function is referred to as the CGGB module.

3.2.1 The coarse-grained potential and parameters

In Chapter 1, coarse-grained models and potentials for MD simulations were discussed. The suitability of the potential and model depends on the simulation system. The scope of this project is limited to the simulation of liquid benzene in a vacuum. The Gay-Berne potential function is designed for simulating liquid crystal systems comprising aromatic molecules. It is, therefore, well suited for coarse-grained simulations of liquid benzene.

Golubkov and Ren[3] parametrized their Gay Berne-Electrostatic Multipole Potential (GB-EMP) model for benzene, water and methanol. These parameters were used to perform coarse-grained simulations for each molecular system by implementing their respective CG force fields in TINKER[8]. Out of the three, the Gay-Berne potential is most suited to model benzene due to its ellipsoidal, disk-like shape that is mostly affected by dispersion effects. It is therefore not surprising to note that the Gay-Berne potential contributes around 95% of the interaction potential energy in a simulation of liquid benzene. Because of this, the EMP was excluded in the modelling of liquid benzene in the FEFI package and CGGB module.

3.2.2 Simulation overview

The FEFI MD package[7] generally follows the same structure as the one described for MD simulations in Chapter 1. The package and the CGGB module have limited development features as their implementation is not within the scope of the project. The following is a synopsis of its included features.

1) **Setup simulation**: Like a fully-atomistic simulation, FEFI starts by setting up the simulation using the input and parameters provided by the user. Currently, FEFI can only run NVT simulations i.e. the number of molecules, volume and temperature are fixed while the pressure and energy may vary. The simulation box is set up using the box length provided by the user. The temperature, time step and a total number of steps are also provided by the user. The user also provides a "skip" value which determines the frequency at which the trajectory files are written to and the information is provided on the console. The neighbourhood list for each particle in the system is also created during the setup phase. Besides these values, FEFI further requires the following files.
    - **Initial coordinate files**: These files contain the initial positions of the atoms, molecules and residues in CHARMMS's[9] .cor or .crd format. Structural information is required to convert the fully-atomistic description of the molecule

to its coarse-grained representation and vice versa. The development and parsing of structure or PSF files do not fall under the scope of this thesis. So, the structural information for benzene is currently hard-coded.

- **Parameter files**: When using the CGGB module, the parameters for the Gay-Berne potential function are provided through a parameter file. They are also used to convert the values in the simulation to reduced units.

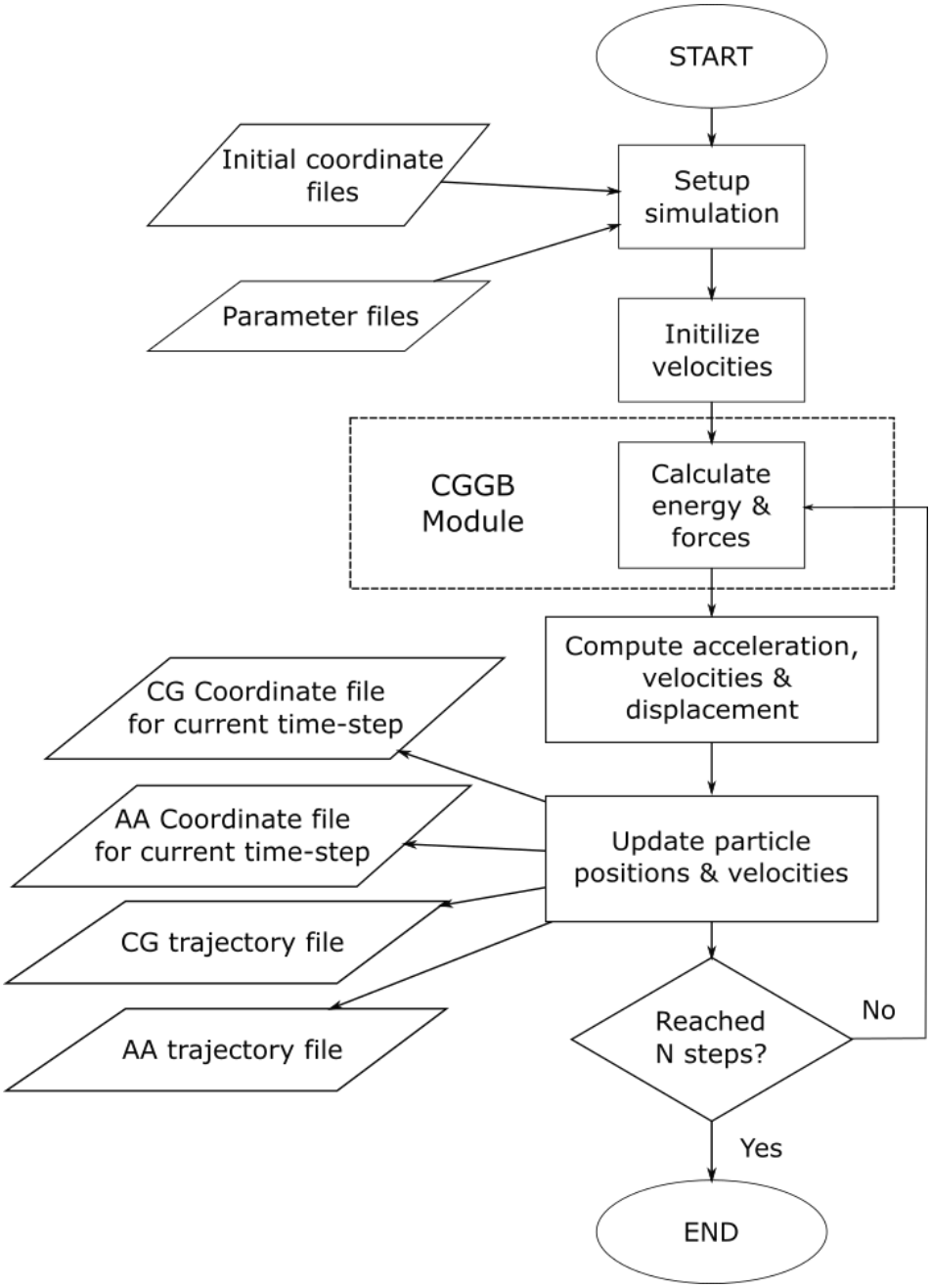


Figure 3.1: The flow of the FEFI MD simulation with the integrated CGGB module

2) **Initialize velocities**: Velocities of the particles in the simulation are initialized at random from a Gaussian distribution depending on the required temperature provided by the user. For a restart simulation, initial velocities are obtained from the restart file.
3) **Calculate potential energy and forces:** The CGGB module performs the energy and force computations using the Gay-Berne analytical function as the force field. It is set up using the Gay-Berne parameter file provided by the user.
4) **Update the dynamics:** Scalar and angular accelerations are computed using the forces and torques obtained from the GB force routine. The scaler velocities and displacements are computed using the Velocity Verlet algorithm[10-11], while angular velocity and rotations are determined using quaternions[7]. To make sure the system stays at the required constant temperature, the Berendsen thermostat[12] is used to scale the velocities. The velocities and temperature are used to determine the kinetic energy of the particles.
5) **Output simulation information:** The user is provided with the information on the potential and kinetic energies via console output. Dynamics can further be analysed using the following files that are created and updated during the simulation:
    - **Coordinate files and restart files:** The final positions of the molecules in the simulation are written out to two files: one coarse-grained version and one fully-atomistic version. The files are in CHARMM's format. A restart file containing the final velocities is also created.
    - **Trajectory files:** CHARMM-format trajectory files are written to after every pre-defined number of steps i.e. the "skip" value provided by the user. Four files are provided: coarse-grained displacement, fully-atomistic displacement, coarse-grained velocity and fully-atomistic velocity. The user also has the option to write a LAAMPS-format trajectory file for displacement to be used in BioVEC[13]: a visual simulation tool that can more accurately represent the shape of the coarse-grained Gay-Berne particle.

3.2.3 The CGGB module and force routine

The computation of the non-bonded interactions is one of the most important sections of FEFI. As mentioned earlier, the Gay-Berne analytical potential is used as the force field for the non-bonded interactions within the CGGB module. The non-bonded energy of the system can be computed using:

$$U_{system} = \sum_{i=1}^{N} \sum_{j=i+1}^{N} U_{ij}^{GB} \tag{3.1}$$

The interactions are computed in this double loop structure. In an actual implementation, however, the number of iterations in the inner loop is restricted by the neighbourhood list size and the cut-off interaction distance.

i. Calculating the energy

The Gay-Berne potential was previously described in Chapter 1 and will be discussed in depth here. Given the distance and molecular orientation vectors describing two interacting particles $(\hat{\boldsymbol{u}}_{\boldsymbol{i}}, \hat{\boldsymbol{u}}_{\boldsymbol{j}}, \vec{\boldsymbol{r}}_{\boldsymbol{ij}})$, the potential can be described by the following equations[1, 3]:

$$U_{ij}^{GB}(\hat{\boldsymbol{u}}_{\boldsymbol{i}}, \hat{\boldsymbol{u}}_{\boldsymbol{j}}, \vec{\boldsymbol{r}}_{\boldsymbol{ij}}) = 4\varepsilon_{ij}(\hat{\boldsymbol{u}}_{\boldsymbol{i}}, \hat{\boldsymbol{u}}_{\boldsymbol{j}}, \vec{\boldsymbol{r}}_{\boldsymbol{ij}})[R^{12} - R^{6}] \tag{3.2}$$

Where:

$$R = \frac{\sigma_0}{r_{ij} - \sigma_{ij}(\hat{\boldsymbol{u}}_{\boldsymbol{i}}, \hat{\boldsymbol{u}}_{\boldsymbol{j}}, \vec{\boldsymbol{r}}_{\boldsymbol{ij}}) + \sigma_0} \tag{3.3}$$

The ε_{ij} and σ_{ij} terms are anisotropic and depend on the orientation of the particle as much as they depend on its physical and chemical properties. σ_{ij} is given by:

$$\sigma_{ij}(\hat{\boldsymbol{u}}_{\boldsymbol{i}}, \hat{\boldsymbol{u}}_{\boldsymbol{j}}, \vec{\boldsymbol{r}}_{\boldsymbol{ij}}) = \sigma_0[1 - H]^{-\frac{1}{2}} \tag{3.4}$$

Where:

$$H = \frac{\chi\alpha^2(\hat{\boldsymbol{u}}_{\boldsymbol{i}} \,.\, \hat{\boldsymbol{r}}_{\boldsymbol{ij}})^2 + \chi\alpha^{-2}(\hat{\boldsymbol{u}}_{\boldsymbol{j}} \,.\, \hat{\boldsymbol{r}}_{\boldsymbol{ij}})^2 - 2\chi^2(\hat{\boldsymbol{u}}_{\boldsymbol{i}} \,.\, \hat{\boldsymbol{r}}_{\boldsymbol{ij}})(\hat{\boldsymbol{u}}_{\boldsymbol{j}} \,.\, \hat{\boldsymbol{r}}_{\boldsymbol{ij}})(\hat{\boldsymbol{u}}_{\boldsymbol{i}} \,.\, \hat{\boldsymbol{u}}_{\boldsymbol{j}})}{1 - \chi^2(\hat{\boldsymbol{u}}_{\boldsymbol{i}} \,.\, \hat{\boldsymbol{u}}_{\boldsymbol{j}})^2} \tag{3.5}$$

The terms σ_0, χ and α are obtained using the dimensions of the molecule: the length, l, and the thickness, d.

$$\sigma_0 = \sqrt{d_i^2 - d_j^2} \tag{3.6}$$

$$\chi = \left[\frac{(l_i^2 - d_i^2)(l_j^2 - d_j^2)}{(l_j^2 + d_i^2)(l_i^2 + d_j^2)}\right]^{-\frac{1}{2}} \tag{3.7}$$

$$\alpha^2 = \left[\frac{(l_i^2 - d_i^2)(l_j^2 + d_i^2)}{(l_j^2 - d_j^2)(l_i^2 + d_j^2)}\right]^{-\frac{1}{2}} \tag{3.8}$$

Similarly, ε_{ij} is given by:

$$\varepsilon_{ij}(\hat{\boldsymbol{u}}_{\boldsymbol{i}}, \hat{\boldsymbol{u}}_{\boldsymbol{j}}, \vec{\boldsymbol{r}}_{\boldsymbol{ij}}) = \varepsilon_0 \varepsilon_1^{\nu}(\hat{\boldsymbol{u}}_{\boldsymbol{i}}, \hat{\boldsymbol{u}}_{\boldsymbol{j}}) \varepsilon_2^{\mu}(\hat{\boldsymbol{u}}_{\boldsymbol{i}}, \hat{\boldsymbol{u}}_{\boldsymbol{j}}, \vec{\boldsymbol{r}}_{\boldsymbol{ij}}) \tag{3.9}$$

Where:

$$\varepsilon_1(\hat{\boldsymbol{u}}_{\boldsymbol{i}}, \hat{\boldsymbol{u}}_{\boldsymbol{j}}) = \left[1 - \chi^2(\hat{\boldsymbol{u}}_{\boldsymbol{i}} . \hat{\boldsymbol{u}}_{\boldsymbol{j}})^2\right]^{-\frac{1}{2}} \tag{3.10}$$

$$\varepsilon_2(\hat{\boldsymbol{u}}_{\boldsymbol{i}}, \hat{\boldsymbol{u}}_{\boldsymbol{j}}, \vec{\boldsymbol{r}}_{\boldsymbol{ij}}) = 1 - H' \tag{3.11}$$

$$H' = \frac{\chi'\alpha'^2(\hat{\boldsymbol{u}}_{\boldsymbol{i}} . \hat{\boldsymbol{r}}_{\boldsymbol{ij}})^2 - \chi'\alpha'^{-2}(\hat{\boldsymbol{u}}_{\boldsymbol{j}} . \hat{\boldsymbol{r}}_{\boldsymbol{ij}})^2 - 2\chi'^2(\hat{\boldsymbol{u}}_{\boldsymbol{i}} . \hat{\boldsymbol{r}}_{\boldsymbol{ij}})(\hat{\boldsymbol{u}}_{\boldsymbol{j}} . \hat{\boldsymbol{r}}_{\boldsymbol{ij}})(\hat{\boldsymbol{u}}_{\boldsymbol{i}} . \hat{\boldsymbol{u}}_{\boldsymbol{j}})}{1 - \chi'^2(\hat{\boldsymbol{u}}_{\boldsymbol{i}} . \hat{\boldsymbol{u}}_{\boldsymbol{j}})^2} \tag{3.12}$$

The $\chi'\alpha'^2$, $\chi'\alpha'^{-2}$ and χ'^2 terms are described by:

$$\chi' = \frac{1 - \left(\frac{\varepsilon_E}{\varepsilon_S}\right)^{\frac{1}{\mu}}}{1 + \left(\frac{\varepsilon_E}{\varepsilon_S}\right)^{\frac{1}{\mu}}} \tag{3.13}$$

$$\alpha'^2 = \left[1 + \left(\frac{\varepsilon_E}{\varepsilon_S}\right)^{\frac{1}{\mu}}\right]^{-1} \tag{3.14}$$

Where:

- ε_0 is the well-depth of the interacting molecules in the cross configuration
- ε_E is the well-depth of the interacting molecules in the end-to-end/face-to-face configuration
- ε_S is the well-depth of the interacting molecules in the side-by-side configuration

μ and ν define the structure and characteristics of the Gay-Berne potential and are set to the canonical values of 2.0 and 1.0 respectively. From the interaction energies, the energy per

molecule is computed by dividing the energy equally between the two interacting particles and accumulating the energy for each particle.

ii. Computing the reaction coordinates

Using vectors as parameters can be complicated. As a simplification, four scalar variables or reaction coordinates can instead be derived from these vectors. The Gay-Berne potential equation can, therefore, be re-written as:

$$U_{ij}^{GB}(r, a, b, g) = 4\varepsilon_{ij}(r, a, b, g)[R^{12} - R^{6}] \tag{3.15}$$

The reaction coordinates are described as follows:

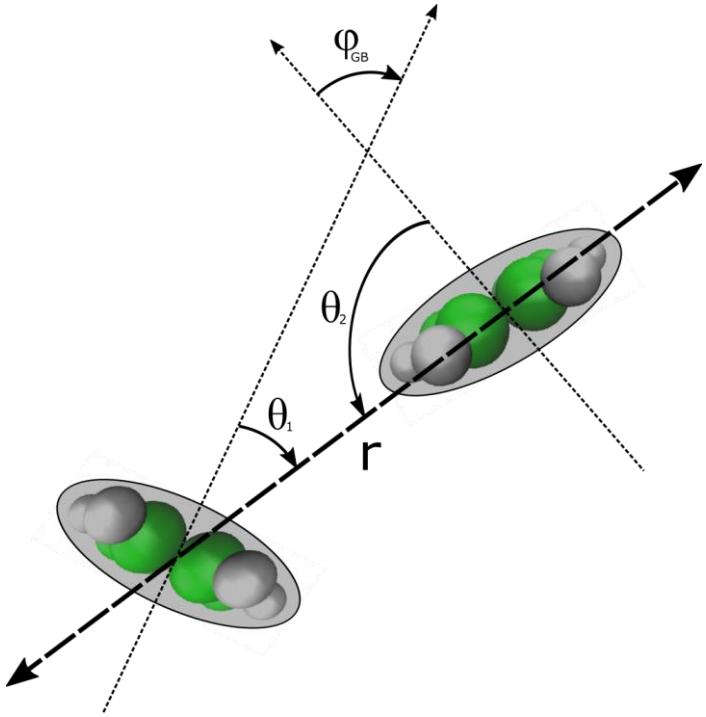


Figure 3.2: The reaction coordinates for coarse-grained benzene

- r: The distance between the centers of the two particles.
- $a = r\cos\theta_1$: The cosine of the molecular vector angle (orientation) of the first particle.
- $b = r\cos\theta_2$: The cosine of the molecular vector angle (orientation) of the second particle.
- $g = \cos\varphi$: The cosine of the angle between the two molecular vectors.

The position vectors of a molecular pair i and j are defined as:

$$\vec{\boldsymbol{p}}_{\boldsymbol{i}} = \begin{pmatrix} x_i \\ y_i \\ z_i \end{pmatrix} \text{ and } \vec{\boldsymbol{p}}_{\boldsymbol{j}} = \begin{pmatrix} x_j \\ y_j \\ z_j \end{pmatrix} \tag{3.16}$$

where each vector comprises three Cartesian coordinates defining the position of the molecule in 3-dimensional space. The distance between i and j is therefore defined as:

$$\vec{\boldsymbol{r}}_{\boldsymbol{ij}} = \vec{\boldsymbol{p}}_{\boldsymbol{i}} - \vec{\boldsymbol{p}}_{\boldsymbol{j}} = \begin{pmatrix} x_i - x_j \\ y_i - y_j \\ z_i - z_j \end{pmatrix} \tag{3.17}$$

$\vec{\boldsymbol{r}}_{\boldsymbol{ij}}$ runs along the interaction axis of the two molecules. Besides their positions, each of the molecules are also defined by their molecular directional vectors: $\hat{\boldsymbol{u}}_{\boldsymbol{i}}$ and $\hat{\boldsymbol{u}}_{\boldsymbol{j}}$. For benzene, the molecular vector is parallel to the C6 axis and is therefore normal to the plane of both the fully-atomistic molecule and CG particle. Using these three vectors, the reaction coordinates are computed:

$$r = \left|\vec{\boldsymbol{r}}_{\boldsymbol{ij}}\right| = \sqrt{\left(\vec{\boldsymbol{r}}_{\boldsymbol{ij}} \,.\, \vec{\boldsymbol{r}}_{\boldsymbol{ij}}\right)} \tag{3.18}$$

r is the scalar distance between the two molecules i.e. the magnitude of the distance vector $\vec{\boldsymbol{r}}_{\boldsymbol{ij}}$. It is also used to normalize $\vec{\boldsymbol{r}}_{\boldsymbol{ij}}$:

$$\hat{\boldsymbol{r}}_{\boldsymbol{ij}} = \frac{1}{\left|\vec{\boldsymbol{r}}_{\boldsymbol{ij}}\right|}\vec{\boldsymbol{r}}_{\boldsymbol{ij}} = \frac{1}{\sqrt{\left(\vec{\boldsymbol{r}}_{\boldsymbol{ij}} \,.\, \vec{\boldsymbol{r}}_{\boldsymbol{ij}}\right)}}\vec{\boldsymbol{r}}_{\boldsymbol{ij}} = \frac{1}{r}\vec{\boldsymbol{r}}_{\boldsymbol{ij}} \tag{3.19}$$

The normalized distance vector, $\hat{\boldsymbol{r}}_{\boldsymbol{ij}}$, along with the molecular directional vectors, $\hat{\boldsymbol{u}}_{\boldsymbol{i}}$ and $\hat{\boldsymbol{u}}_{\boldsymbol{j}}$ are used to compute the interaction angles and their cosines:

$$\theta_1 = \cos^{-1}\left(\hat{\boldsymbol{u}}_{\boldsymbol{i}} \,.\, \hat{\boldsymbol{r}}_{\boldsymbol{ij}}\right) \tag{3.20}$$

θ_1 is defined as the angle between the directional vector $\hat{\boldsymbol{u}}_{\boldsymbol{i}}$, and the interaction axis or distance vector $\vec{\boldsymbol{r}}_{\boldsymbol{ij}}$.

$$\theta_2 = \cos^{-1}\left(\hat{\boldsymbol{u}}_{\boldsymbol{j}} . \hat{\boldsymbol{r}}_{\boldsymbol{ij}}\right) \tag{3.21}$$

θ_2 is defined as the angle between the directional vector $\hat{\boldsymbol{u}}_{\boldsymbol{j}}$ and the interaction axis or distance vector $\vec{\boldsymbol{r}}_{\boldsymbol{ij}}$.

$$\varphi = \cos^{-1}\left(\hat{\boldsymbol{u}}_{\boldsymbol{i}} . \hat{\boldsymbol{u}}_{\boldsymbol{j}}\right) \tag{3.22}$$

φ is defined as the angle between the two directional vectors $\hat{\boldsymbol{u}}_{\boldsymbol{i}}$ and $\hat{\boldsymbol{u}}_{\boldsymbol{j}}$.

It is important to note that the Gay-Berne potential as defined in Equation 3.15 is not actually in terms of the angles described above but rather their cosines i.e. the dot products of the three vectors involved. So, the actual reaction coordinates to be used are:

$$r = \left|\vec{\boldsymbol{r}}_{\boldsymbol{ij}}\right| = \sqrt[2]{\left(\hat{\boldsymbol{r}}_{\boldsymbol{ij}} . \hat{\boldsymbol{r}}_{\boldsymbol{ij}}\right)} \tag{3.23}$$

$$a = \left(\hat{\boldsymbol{u}}_{\boldsymbol{i}} . \vec{\boldsymbol{r}}_{\boldsymbol{ij}}\right) = r\left(\hat{\boldsymbol{u}}_{\boldsymbol{i}} . \hat{\boldsymbol{r}}_{\boldsymbol{ij}}\right) = r\cos\theta_1 \tag{3.24}$$

$$b = \left(\hat{\boldsymbol{u}}_{\boldsymbol{j}} . \vec{\boldsymbol{r}}_{\boldsymbol{ij}}\right) = r\left(\hat{\boldsymbol{u}}_{\boldsymbol{j}} . \hat{\boldsymbol{r}}_{\boldsymbol{ij}}\right) = r\cos\theta_2 \tag{3.25}$$

$$g = \left(\hat{\boldsymbol{u}}_{\boldsymbol{i}} . \hat{\boldsymbol{u}}_{\boldsymbol{j}}\right) = \cos\varphi \tag{3.26}$$

iii. Calculating the forces and torques

With respect to the scalar reaction coordinates, the partial derivatives can be described as[3]:

$$\frac{\partial U_{ij}^{GB}}{\partial r} = \frac{8\varepsilon\mu H'}{\varepsilon_2 r_{ij}}\left[R^{12} - R^6\right] - \frac{4\varepsilon}{\sigma_0}\left(1 + \frac{\sigma^3 H}{r_{ij}\sigma_0^2}\right)\left[12R^{13} - 6R^7\right] \tag{3.27}$$

$$\begin{aligned}\frac{\partial U_{ij}^{GB}}{\partial a} = &-\frac{8\varepsilon\mu}{\varepsilon_2 r_{ij}}\left[R^{12} - R^6\right]\frac{\chi'\alpha'^2\left(\hat{\boldsymbol{u}}_{\boldsymbol{i}} . \hat{\boldsymbol{r}}_{\boldsymbol{ij}}\right) - \chi'^2\left(\hat{\boldsymbol{u}}_{\boldsymbol{j}} . \hat{\boldsymbol{r}}_{\boldsymbol{ij}}\right)\left(\hat{\boldsymbol{u}}_{\boldsymbol{i}} . \hat{\boldsymbol{u}}_{\boldsymbol{j}}\right)}{1 - \chi'^2\left(\hat{\boldsymbol{u}}_{\boldsymbol{i}} . \hat{\boldsymbol{u}}_{\boldsymbol{j}}\right)^2} \\ &+ \frac{8\varepsilon\sigma^3}{r_{ij}\sigma_0^3}\left[6R^{13} - 3R^7\right]\frac{\chi\alpha^2\left(\hat{\boldsymbol{u}}_{\boldsymbol{i}} . \hat{\boldsymbol{r}}_{\boldsymbol{ij}}\right) + \chi^2\left(\hat{\boldsymbol{u}}_{\boldsymbol{j}} . \hat{\boldsymbol{r}}_{\boldsymbol{ij}}\right)\left(\hat{\boldsymbol{u}}_{\boldsymbol{i}} . \hat{\boldsymbol{u}}_{\boldsymbol{j}}\right)}{1 - \chi^2\left(\hat{\boldsymbol{u}}_{\boldsymbol{i}} . \hat{\boldsymbol{u}}_{\boldsymbol{j}}\right)^2}\end{aligned} \tag{3.28}$$

$$\begin{aligned}\frac{\partial U_{ij}^{GB}}{\partial b} = &-\frac{8\varepsilon\mu}{\varepsilon_2 r_{ij}}\left[R^{12} - R^6\right]\frac{\chi'\alpha'^2\left(\hat{\boldsymbol{u}}_{\boldsymbol{i}} . \hat{\boldsymbol{r}}_{\boldsymbol{ij}}\right) - \chi'^2\left(\hat{\boldsymbol{u}}_{\boldsymbol{j}} . \hat{\boldsymbol{r}}_{\boldsymbol{ij}}\right)\left(\hat{\boldsymbol{u}}_{\boldsymbol{i}} . \hat{\boldsymbol{u}}_{\boldsymbol{j}}\right)}{1 - \chi'^2\left(\hat{\boldsymbol{u}}_{\boldsymbol{i}} . \hat{\boldsymbol{u}}_{\boldsymbol{j}}\right)^2} \\ &+ \frac{8\varepsilon\sigma^3}{r_{ij}\sigma_0^3}\left[6R^{13} - 3R^7\right]\frac{\chi\alpha^2\left(\hat{\boldsymbol{u}}_{\boldsymbol{i}} . \hat{\boldsymbol{r}}_{\boldsymbol{ij}}\right) + \chi^2\left(\hat{\boldsymbol{u}}_{\boldsymbol{j}} . \hat{\boldsymbol{r}}_{\boldsymbol{ij}}\right)\left(\hat{\boldsymbol{u}}_{\boldsymbol{i}} . \hat{\boldsymbol{u}}_{\boldsymbol{j}}\right)}{1 - \chi^2\left(\hat{\boldsymbol{u}}_{\boldsymbol{i}} . \hat{\boldsymbol{u}}_{\boldsymbol{j}}\right)^2}\end{aligned} \tag{3.29}$$

$$\frac{\partial U_{ij}^{GB}}{\partial g} = 4\varepsilon v[R^{12} - R^6]\left(\frac{\chi^2(\hat{\boldsymbol{u}}_i.\hat{\boldsymbol{u}}_j)}{1-\chi^2(\hat{\boldsymbol{u}}_i.\hat{\boldsymbol{u}}_j)^2}\right) - \frac{8\varepsilon\mu}{\varepsilon_2}[R^{12} - R^6]\frac{\chi'^2(\hat{\boldsymbol{u}}_i.\hat{\boldsymbol{r}}_{ij})(\hat{\boldsymbol{u}}_j.\hat{\boldsymbol{r}}_{ij}) - H'\chi'^2(\hat{\boldsymbol{u}}_i.\hat{\boldsymbol{u}}_j)}{1-\chi'^2(\hat{\boldsymbol{u}}_i.\hat{\boldsymbol{u}}_j)^2} + \frac{8\varepsilon\sigma^3}{\sigma_0^3}[6R^{13} - 3R^7]\frac{\chi^2(\hat{\boldsymbol{u}}_i.\hat{\boldsymbol{r}}_{ij})(\hat{\boldsymbol{u}}_j.\hat{\boldsymbol{r}}_{ij}) - H\chi^2(\hat{\boldsymbol{u}}_i.\hat{\boldsymbol{u}}_j)}{1-\chi^2(\hat{\boldsymbol{u}}_i.\hat{\boldsymbol{u}}_j)^2} \tag{3.30}$$

Using these derivatives, the forces and torques are computed as follows[3, 14]:

$$\vec{\boldsymbol{F}}_i = -\vec{\boldsymbol{F}}_j = \hat{\boldsymbol{r}}_{ij}\frac{\partial U_{ij}^{GB}}{\partial r} + \hat{\boldsymbol{u}}_i\frac{\partial U_{ij}^{GB}}{\partial a} + \hat{\boldsymbol{u}}_{ij}\frac{\partial U_{ij}^{GB}}{\partial b} \tag{3.31}$$

$$\vec{\boldsymbol{\tau}}_i = (\hat{\boldsymbol{r}}_{ij} \times \hat{\boldsymbol{u}}_i)\frac{\partial U_{ij}^{GB}}{\partial a} + (\hat{\boldsymbol{u}}_i \times \hat{\boldsymbol{u}}_j)\frac{\partial U_{ij}^{GB}}{\partial g} \tag{3.32}$$

$$\vec{\boldsymbol{\tau}}_j = (\hat{\boldsymbol{r}}_{ij} \times \hat{\boldsymbol{u}}_j)\frac{\partial U_{ij}^{GB}}{\partial b} + (\hat{\boldsymbol{u}}_j \times \hat{\boldsymbol{u}}_i)\frac{\partial U_{ij}^{GB}}{\partial g} \tag{3.33}$$

Like the energy, the forces and torques are accumulated for each particle in the system during the computation of the interactions.

3.2.4 CPU-based sequential implementation

The CGGB module has a simple sequential implementation for a single CPU. In the double loop-structure described by Equation 3.1, the routine iterates over every CG particle in the system and processes its interaction with the particles in its neighbourhood list. The routine, which is called once per time step, is outlined by the pseudo-code in Listing 3.1:

```
For every particle (i) in the system
   - Obtain the neighbourhood list
   - For every pair particle (j) in the list
        o Compute the distance
        o Check for cut-offs and apply minimum imaging
        o Compute the remaining reaction coordinates
        o Compute the Gay-Berne potential energy
        o Compute the Gay-Berne partial derivatives
        o Divide the energy equally between the particles
```

```
        ○ Compute the forces and torques from the partial derivatives
        ○ Update total system energy
```

Listing 3.1: Pseudo-code for the computation of the energy and forces on a single-threaded CPU

To test the potential of the GPU in improving the computational performance of FEFI, the CGGB module was ported to a GPU with the aid of CUDA. Due to parallelization, the GPU algorithm is different from the one described for the CPU version in Listing 3.1.

While the CGGB module was ported to the GPU, the remaining routines of FEFI still execute on the CPU sequentially. The schematic in Figure 3.3 illustrates this operation:

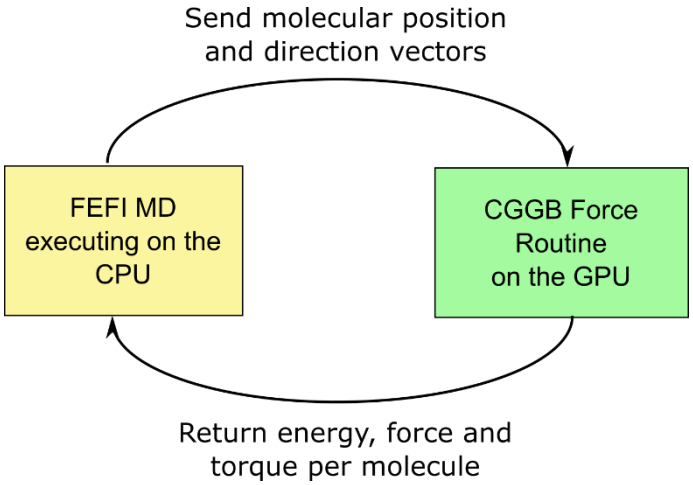


Figure 3.3: FEFI MD executes on the CPU and requests the CGGB module on the GPU for energy, forces and torques at every time step

i. Algorithm overview

Due to the GPU's SIMT architecture, for ideal performance the algorithm must have an embarrassingly parallel structure. This can be achieved by shifting the focus from the "calculation-per-particle" method in the sequential implementation to a "calculation-per-interaction" method instead. The threads in the GPU will each be assigned to a unique interaction in the system. The reaction coordinates are set up as batches of input for the energy and force calculations on the GPU. These batches are created on the GPU itself using the molecular position and directional vectors. Along with these vectors, the CGGB routine also has access to the neighbourhood list per particle.

ii. Computing the reaction coordinates in parallel

Unlike the energy computations, the reaction coordinate computations do require the double loop structure, where the outer loop iterates over the particles, and the inner loop iterates over its neighbour list. To do away with the double-loop structure, the threads and block IDs are used to map to the inner and outer loop respectively.

The size of the thread block, TPB, is chosen to achieve the maximum theoretical occupancy of the GPU's SMs. There is no shared memory usage employed in this kernel. The grid size is derived from the maximum number of interactions per CG particle and the thread block size:

$$\begin{aligned} BPG &= \left\lceil \frac{\max(interactions)}{TPB} \right\rceil \\ &= \left\lceil \frac{\max(pairs\ per\ particle) * total\ number\ of\ particles}{TPB} \right\rceil \end{aligned} \tag{3.34}$$

In the kernel, the global thread index is broken down to obtain two indices. The first index maps to each particle in the system. The second index is used to obtain the paired particle from the neighbourhood list of the current particle. The global thread index is computed using the thread index per block, the block index per grid and the block size. This method is used to ensure that very few threads are idle or stalled per block, and to increase the throughput of the GPU to compute the reaction coordinates of larger systems. The number of pairs per particle is not consistent, however, and some blocks will have more stalled threads than others.

Another source of complexity and stalls is the cut-off distance. While a particle may be present in another's neighbourhood list if the distance between the two is too long the energy and force for that interaction is practically zero. As a result, the actual number of interactions needs to be tracked and stored as it is different from the previously determined number, and to ensure that there are fewer stalled threads in the next kernel.

A counter variable that resides in the device's global memory is used to keep track of the next available index in the reaction coordinate array. The counter starts at zero and each thread reads in the counter value, stores the computed reaction coordinate at the location pointed by it, and updates it by one. Since these computations are in parallel, this will inevitably lead to race conditions when reading from and writing to the same location. To address this, CUDA's

in-built mutex-based *atomicAdd*[15] function was used to read from and update the counter in a single step. The intrinsic function is well optimized and adds very little delay to the overall function. The pseudo-code in Listing 3.2 briefly explains the implementation of the kernel in code:

```
Break the global thread ID to particle ID (i) and list index (l)
If (l) is less than the size of the neighbour list for the particle
   - Obtain index (j) from the neighbour list using (l)
   - Compute the distance for the pair and apply minimum imaging
   - If within the cut-off range
        o Compute the remaining reaction coordinates
        o Obtain the available index value from the counter and update
          the counter (mutex-based access)
        o Use the index to store reaction
```

Listing 3.2: Pseudo-code for computing the reaction coordinate batch with Kernel 1

Indexing considerations with the block and thread IDs are taken to ensure there are no out-of-bounds errors and incorrect fetches and writes. Each item in the neighbourhood list, input vectors and output reaction coordinates is accessed only once by individual threads and so the kernels follow an embarassingly parallel structure. The shared memory was not involved in this mechanism but may act as a cache in future revisions. The actual index of the interaction is also stored along with the reaction coordinates for reverse mapping when computing the total energy, force and torque per particle.

iii. Computing energy and forces

This step is split between two kernels. The first one is the GPU implementation of the Gay-Berne potential and its partial derivatives. As evidenced by Equations 3.2 to 3.30, the algorithm for computing GB potential energy and derivatives is complex. However, the CPU implementation[7] offers a fast adaptation of the algorithm which the GPU implementation also uses.

As mentioned earlier, to conform to the GPU's SIMT structure, the double-loop in the algorithm (Equation 3.1) had to be adapted to an embarrassingly parallel structure. Since Kernel 1 was used to compute the reaction coordinate batch, Kernel 2 can focus on the computation of the GB energy and its derivatives. The final value of the counter variable at the end of Kernel 1's execution provides the total number of unique interactions to be computed. This ensures that most of the threads are involved in the computation and are not stalled. The kernel's flow is

straight-forward, however, the energy and derivative computations are slightly complicated (Equations 3.2 to 3.30) using up many resources.

```
If thread ID (tid) is less than the total number of interactions:
   - Read in reaction coordinates to shared memory
   - Compute the energy and four partial derivatives and store in shared
     memory
   - Copy the energy and derivatives to their arrays in global memory
```

Listing 3.3: Pseudo-code for computing the Gay-Berne energy and partial derivatives using Kernel 2

Shared memory is used as a cache to store the repeatedly used reaction coordinates and results. This reduces the load on the registers. Given the complexity of the GB calculations and the resources used, Kernel 2 does not compute and accumulate the energy and forces per particle. This is instead done in Kernel 3 that follows the same embarrassingly parallel structure as Kernel 2.

```
If thread ID (tid) is less than the total number of interactions:
   - Split (tid) to get the indices of the pair of interacting particles
     (i) and (j)
   - Read in the position and molecular direction vectors
   - Use the vectors to compute the force and torque
   - Using mutex-based addition operations:
        o Add half of the energy to the total for particle (i) and the
          other half to particle (j) at their respective indices in the
          global energy array
        o Add the total energy to the global energy variable
        o Add the computed forces and torques for particle (i) and
          particle (j) to their total at their respective indices in the
          global force and torque arrays
```

Listing 3.4: Pseudo-code for computing and accumulating the energy, forces and torques per particle using Kernel 3

Like Kernel 1, Kernel 3 also uses the *atomicAdd* function to accumulate the energy, forces and torques per particle and the total energy of the system. While this does slow the execution of the threads, it eliminates the need for multiple arrays and additional kernels. Kernel 3 does not use any shared memory.

3.2.5 Performance analysis

The performance of the GPU implementation of the CGGB module is evaluated by checking the accuracy of the results and the execution times as opposed to the CPU version. The GPU occupancy of the kernels and other metrics, previously covered in Chapter 2, are also looked at to highlight potential improvements.

The FEFI MD simulation and the CPU version of CGGB were executed on an Intel® Xeon® E5-2620 clocked at 2.00GHz. The GPU version of the CGGB module was executed on an Nvidia Tesla K40 (specifications provided in Chapter 2).

i. Kernel structure analysis

The analysis of the kernels using the Nvidia Profiler provides the information in Table 3.1:

	K1	K2	K3
Threads per block	96	640	512
Registers per thread	23	45	34
Shared memory per block	0B	22.5KiB	0B
Theoretical occupancy	75%	62.5%	75%
Actual occupancy	54.6%	29.4%	25.9%
FLOP (% of total execution count)	14%	52%	39%
Inactive (% of total execution count)	65%	8%	11%
Load/Store (% of total execution count)	2%	4%	11%

Table 3.1: Kernel structure analysis of the GPU-based CGGB module

The theoretical occupancies of all three kernels is quite high. The number of threads per block for Kernels 2 and 3 were chosen specifically to get the best possible theoretical occupancy. However, the number of threads used in Kernel 1 relies on the maximum number of interactions per particle and is thus variable. Due to its low register usage, the theoretical occupancy of the kernel is always above 50% for more than 64 threads per block and enough to hide the latency of the kernel.

Despite the large number of inactive threads per warp due to the three branching statements, the actual occupancy of the kernel is high at 54.6%. The main source of latency is execution

dependency. This can potentially be rectified in future revisions by increasing instruction level parallelism however it does not seem to weigh down the kernel heavily.

Kernels 2 and 3, however, do not achieve the desired 50% occupancy in practice. The two kernels are much more complex as evidenced by the large number of floating-point operations they execute. Kernel 2 has latencies arising from execution and memory dependency, which can be addressed by managing instruction level parallelism and memory access patterns.
Kernel 3 is affected by memory throttle. The kernel is not only dependent on Kernel 2 for results but has multiple memory-based operations due to its high usage of *atomicAdd* to compute the total energy, force and torque per molecule and the system energy.

Since the focus of this thesis is a proof of concept of the CG algorithms, a high-level optimization of the kernels is not included in this thesis. The objective is to observe the suitability of the algorithm and implementation at a "simpler" level in terms of accuracy and timing. The in-practice execution of the kernels will be analysed in the following subsections.

ii. Analysing the accuracy of the GPU implementation

The Mean Absolute Percentage Error (MAPE) check was used to determine the accuracy of the GPU results when compared to those from the CPU. It is described as:

$$MAPE = \frac{1}{N}\sum_{i}^{N}\frac{|y_c - y_o|}{|y_o|} \tag{3.35}$$

In Equation 3.35, y_o are the original (CPU) values and y_c are the calculated (GPU) values.

The check was performed on the energy, force and torque values obtained after one time-step against various box sizes. Since the forces and torques are vectors, the MAPE is computed for their magnitudes.

No. of molecules	**Box side length(Å)**	**MAPE(%)**		
		Energy	**Force**	**Torque**
100	24.0	0.0821	0.0014	0.0033
250	34.0	0.0040	0.0003	0.0005
500	42.0	0.0322	0.0007	0.0013

1000	53.0	0.0003	0.0039	0.0012
2500	72.0	0.0055	0.0014	0.0011

Table 3.2: Observing the accuracy of GPU implementation when compared to the CPU one

The errors in all the results are minimal, with the box size barely affecting the results. This is expected since GB energy and partial derivatives are computed using the same algorithm on the CPU and GPU.

iii. Comparing execution times

The execution time for one step of the CGGB module provides a specific indication of the speed-up achieved. Table 3.3 covers the time taken by a single execution of the CGGB module on the CPU and GPU for varying box sizes:

No. of molecules	Box side length(Å)	CPU(ms)	GPU without latency(ms)	Speed-up (without latency)	GPU with latency(ms)	Speed-up (with latency)
100	24.0	33.3	0.190	175	13.4	2.49
250	34.0	65.3	0.184	355	13.8	4.73
500	42.0	119.3	0.207	575	17.2	6.94
1000	53.0	228.0	0.315	724	38.1	5.98
2500	72.0	479.1	0.615	779	89.0	5.38

Table 3.3: Time taken for a single execution of the CGGB module for various simulation sizes

The speed-up achieved from the GPU is broken into two sections: the total execution time of only the three kernels on the GPU (i.e. without latency) and the total time of the GPU-based force routine including the memory copy overhead and other latencies (i.e. with latency). While comparing to the kernel execution times without the latency, the GPU's speed-up is phenomenal. However, it is fairer to include the latencies, since they notably add to the total time taken. The speed-up achieved with latencies is, in fact, much less.

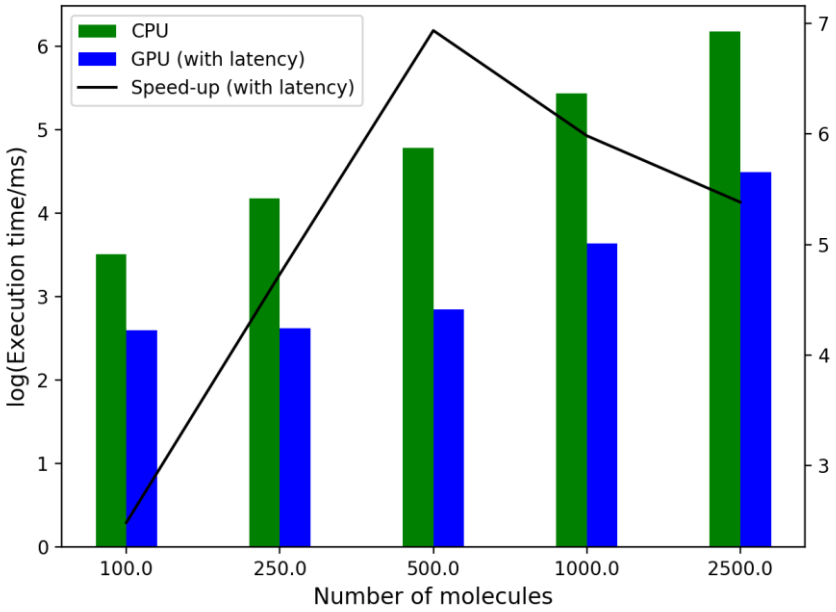


Figure 3.4: Plot of the execution time per time step of the CGGB module for various simulation sizes

Looking closely at the GPU's latency in Table 3.4 shows that memory copy and other latencies take up most of the time on the GPU.

No. of molecules	Box side length(Å)	GPU without latency(ms)	GPU with latency(ms)	Latency(ms)	Latency (%)
100	24.0	0.190	13.4	13.2	97.0
250	34.0	0.184	13.8	13.6	98.5
500	42.0	0.207	17.2	17.0	98.8
1000	53.0	0.315	38.1	27.8	73.0
2500	72.0	0.615	89.0	88.4	99.3

Table 3.4: Latency in the GPU's execution of the CGGB module

The size of the latency is an indication that the memory copy to and from the GPU needs to be managed better. This can be done in a future implementation using GPU streams to coalesce kernel execution and memory copy.

In Table 3.5, the speed-up achieved in long simulations of FEFI with the CGGB module is analysed. The simulations were conducted for a 42.0Å × 42.0Å × 42.0Å cubical box with 500 benzene molecules at a 1fs time step while printing out to console at every 10 steps.

No. of steps	Total time(ps)	CPU(s)	GPU(s)	Speed-up
10000	10	266	148	1.80
25000	25	665	359	1.85
50000	50	1323	750	1.76
100000	100	2659	1566	1.70

Table 3.5: Execution time and speed-up of the CGGB module on GPU and CPU

From the Table 3.5 and Figure 3.1, it is visible that the increase in CPU and GPU execution times is relatively similar at different rates. This is evident from the speed-up which is almost consistent. This is because the main source of the speed-up is the CGGB module which takes the same amount of time per time step, while the remaining execution of the FEFI MD simulation is done sequentially on the CPU. This is an indication that, along with better memory copy management, more of the FEFI code should be on the GPU. Not only will it limit the memory copy latency, but it will also make asynchronous execution using GPU streams more feasible.

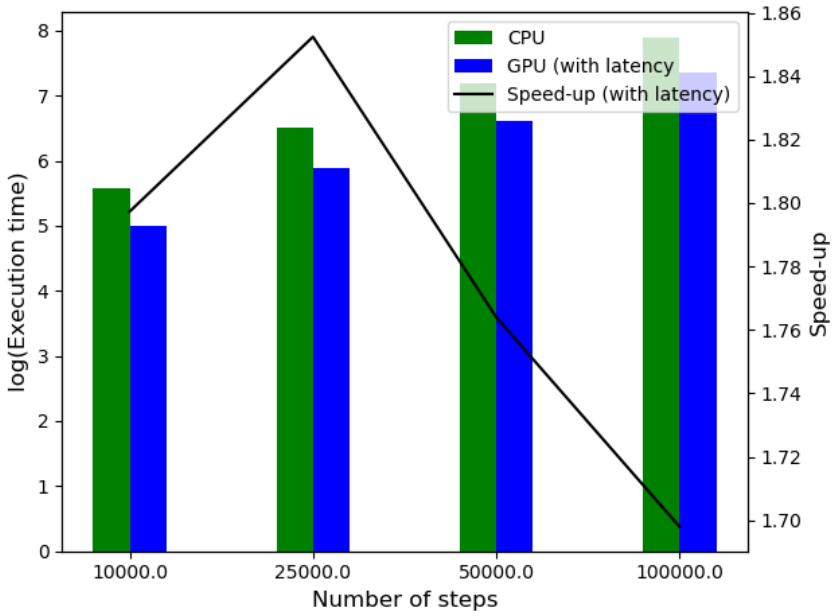


Figure 3.5: Plot of execution times of FEFI with the CGGB module for several time steps

3.3 Simulation results

The FEFI MD simulation using the CGGB module was used to simulate liquid benzene using the Golubkov and Ren parameter set[3]. The performance and accuracy of the simulation were verified when compared to a "golden standard" CHARMM simulation of benzene using various

MD metrics. All simulations were run for 100,000 steps at $1fs$ time steps. The constant temperature was set to 298K.

3.3.1 Probability Distribution Functions

Probability distribution functions provide information on how the molecules are spread throughout the simulation and their structural information in the box.

i. Radial Distributions

The radial distribution[16] or the pair correlation function is the measure of how the density of the particles in a system varies with the distance from a reference particle. In other words, it provides the probability of finding particles at any given position from the reference particle. It can be described as:

$$g(r) = \rho 4\pi r^2 dr \tag{3.36}$$

Where:

- ρ is the number density of the system of particles
- r is the distance from the reference particle
- dr is the extra margin that is included in the measurement

It is generally performed for each molecule in the system and averaged. For benzene, the RDF for the centre-of-mass to centre-of-mass of the molecules and the carbon to carbon for a single carbon on the ring is measured.

The plots in Figure 3.6 and Figure 3.7 were obtained from the position trajectories of the molecules in the system using CHARMM's in-built RDF calculator[17]. The RDF from the CHARMM simulation can be considered the "gold standard" version and is used as the basis of comparison.

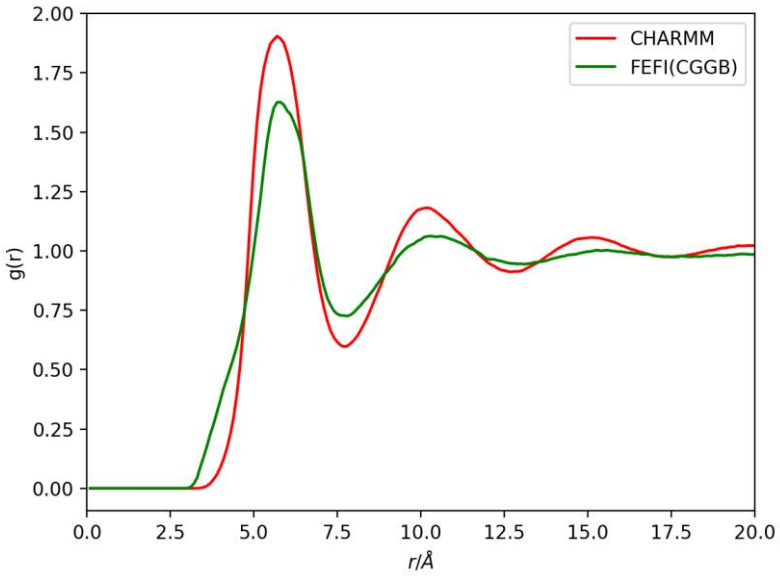


Figure 3.6: The centre-of-mass to centre-of-mass Radial Distribution of Benzene

For the centre-of-mass to centre-of-mass plots, the Golubkov and Ren results follow the trend of the CHARMM results well, albeit with poorer peaks. There is also some structure present at short-contact distances for the Golubkov and Ren results that does not exist for CHARMM.

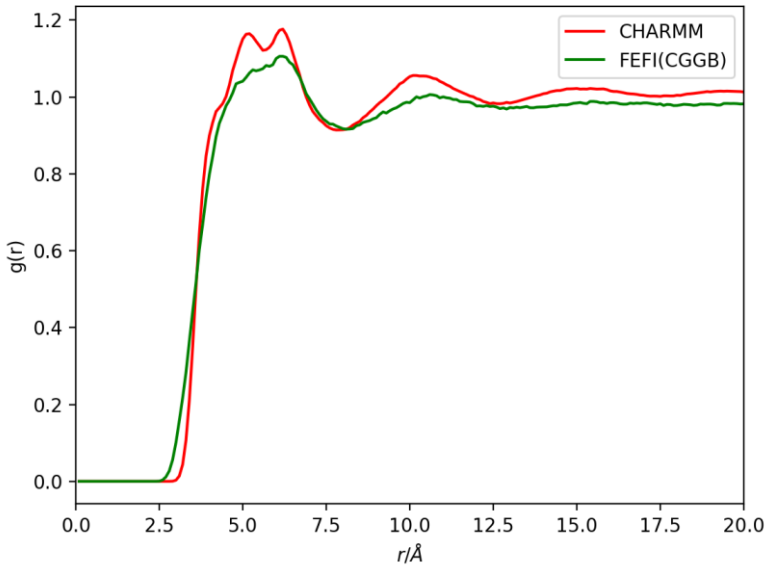


Figure 3.7: The carbon to carbon Radial Distribution of Benzene

For the carbon to carbon plots, the Golubkov and Ren results show good correlation with the CHARMM results. However, the results do not provide the same level of detail and the peak heights are also slightly smaller. This is, however, expected. Despite recovering benzene's structure and its planar orientation, while mapping the CG particle of benzene back to its fully-

atomistic form, the carbons were placed around its centre-or-mass randomly. Since the single tracked carbon was not necessarily in its expected location at each time step, the carbon to carbon RDF does not show the same level of accuracy and detail as the fully-atomistic result.

ii. Radial-Angular Distributions

While the RDFs give a valuable indication of the accuracy of the results, they are one-dimensional and cannot provide details on anisotropy. However, a radial distribution with an angular component can give a much better indication of that.

The Radial-Angular Distribution plots in Figure 3.11 were obtained from the displacement trajectories of the simulations. The displacements were used to obtain the $r\ vs.\ \varphi$ plots that show a clear distribution of the three general positions a pair of benzene molecules take:

- The area before the first peak at very low distances shows the probability of the parallel-displaced or end-to-end/face-to-face position. In this position, the "wider" faces of each benzene molecule face each other i.e. they are parallel:

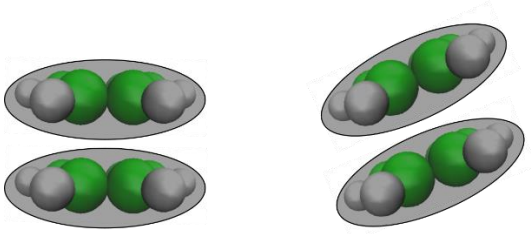


Figure 3.8: The end-to-end/face-to-face configuration

- The first, smaller peak at shorter distances gives the probability of the T-configuration.

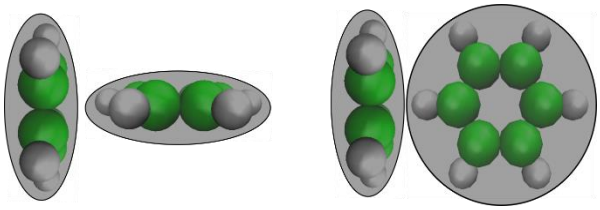


Figure 3.9: The T-configuration

- The second, larger peak at greater distances gives the probability of side-to-side configuration. In this position, the two benzene rings lie flat on the plane with their "wider" faces pointing upwards.

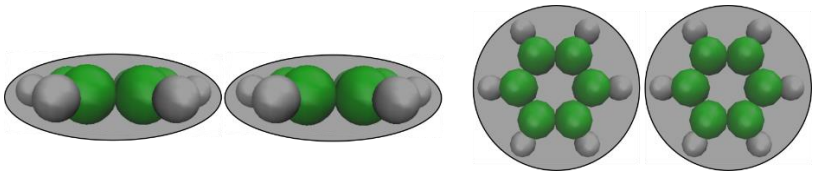


Figure 3.10: The side-to-side configuration

The 2D contour and 3D surface plots for $r\ vs.\ \varphi$ are as follows.

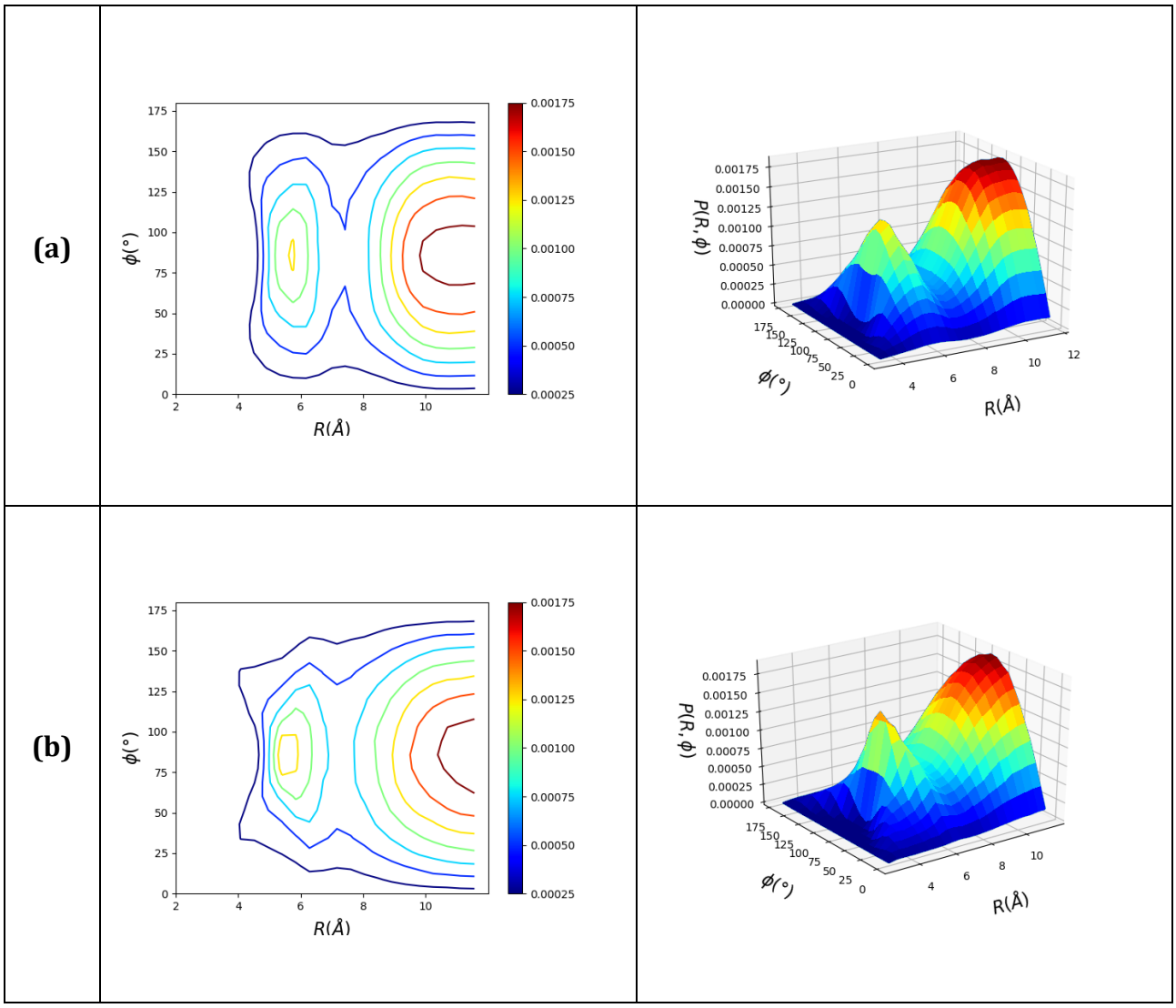


Figure 3.11: 2D contour and 3D surface plots of r vs. φ from a) CHARMM and b) FEFI running the CGGB module with G&R parameters

The CHARMM plots again provide the basis of comparison. The plots are highly detailed and give a good indication of the positions and orientations of the benzene molecules during the simulation.

The Golubkov and Ren plots suggest more favour towards the T-configuration with a higher first peak than the CHARMM plots. The second peak, however, has a similar height. Further, the area and orientations between the peaks seem to be poorly defined when compared to the CHARMM results.

3.3.2 Diffusion coefficient

Transport coefficients, more specifically the diffusion constants are a good indicator of how accurate the motion of the molecules is. Translational diffusion is the linear movement of the molecules in 3D space. The translational diffusion coefficients were computed for each simulation. The published experimental value [18] for the diffusion of liquid benzene at 298K, which the simulations were run at, is also provided for comparison.

	Diffusion coefficient (10^{-9} m^2/s)
CHARMM	2.18
FEFI(CGGB)	3.82
Experimental[18]	2.20

Table 3.6: Comparing the diffusion coefficients

The fully-atomistic CHARMM simulation provides a near accurate diffusion constant when compared to the experimental value. The Golubkov and Ren results however over-estimated both the experimental and CHARMM values of the diffusion coefficients. This corresponds to the center-of-mass to center-of-mass RDF plot in Figure 3.6. The peaks of the Golubkov and Ren plots are shorter and less defined. This suggests that there is less structure present in the simulation with these parameters.

3.3.3 Computational performance

Besides the accuracy of the results, the other important aspect of coarse-graining and a GPU-based implementation is the computational speed-up achieved when compared to a fully-atomistic simulation. For a simulation box with 500 benzene molecules, the execution times for a $1ns$ simulation in CHARMM and FEFI using the CGGB module at $1fs$ time steps are estimated to be:

	Execution time (hrs)
CHARMM	40.74
FEFI(CGGB)	4.35
Speed-up	8.63

Table 3.7: Comparing estimates long timescale simulation executions times

The speed-up obtained from the coarse-grained FEFI simulation and the GPU-accelerated CGGB module is 9.36, which is a remarkable improvement over CHARMM's execution time.

3.4 Concluding remarks

The Gay-Berne analytical potential with the Golubkov and Ren parameters shows promise when it comes to the coarse-grained simulation of benzene. This was expected since benzene is a suitable molecule for the potential. Even so, there are several inaccuracies between the all-atomistic results and the GB results.

An alternative approach to using the Gay-Berne potential would be using a numerical free energy description of the molecule. The Free Energy From Adaptive Reaction Coordinate Forces (FEARCF)[19-20] is a sampling method used to create Free Energy Volumes (FEV)[21] from the interactions in an atomistic molecular dynamics simulation. These FEVs contain the description of two interacting molecules and describe them with respect to various reaction coordinates. This means that that the arrays can be used to obtain the structural information of various systems. For example, a pure substance in a vacuum like benzene can be described with the aid of four reaction coordinates resulting in a four-dimensional free energy array.

Due to the amount of information they include and their dissimilarity with the potential, these arrays cannot be used to parameterize the Gay-Berne potential equation. Instead, the FEV could be treated as a numerical free energy potential for the dynamics instead of extracting parameters from it. This can be achieved by treating the FEARCF FEV as a look-up table (LUT) using an interpolation algorithm to approximate results between grid points. This idea led to the creation of the LUT module for the FEFI MD simulation package. The development and testing of the package will be discussed in Chapters 4, 5 and 6.

3.5 References

1. Gay, J.; Berne, B., Modification of the overlap potential to mimic a linear site–site potential. *The Journal of Chemical Physics* **1981,** *74* (6), 3316-3319.
2. Lennard-Jones, J. E., On the determination of molecular fields. II. From the equation of state of gas. *Proceedings of the Royal Society A* **1924,** *106*, 463-477.
3. Golubkov, P. A.; Ren, P., Generalized coarse-grained model based on point multipole and Gay-Berne potentials. *The Journal of Chemical Physics* **2006,** *125* (6), 064103.
4. Liwo, A.; Pillardy, J.; Czaplewski, C.; Lee, J.; Ripoll, D. R.; Groth, M.; Rodziewicz-Motowidlo, S.; Kamierkiewicz, R.; Wawak, R. J.; Oldziej, S. In *UNRES: a united-residue force field for energy-based prediction of protein structure—orgin and significance of multibody terms*, Proceedings of the Fourth Annual International Conference on Computational Molecular Biology, ACM: 2000; pp 193-200.
5. Voth, G. A., Simulation of Protein Structure and Dynamics with the Coarse-Grained UNRES Force Field. In *Coarse-graining of Condensed Phase and Biomolecular Systems*, Voth, G. A., Ed. CRC Press: 2009; pp 108-119.

6. Makowski, M.; Liwo, A.; Maksimiak, K.; Makowska, J.; Scheraga, H. A., Simple physics-based analytical formulas for the potentials of mean force for the interaction of amino acid side chains in water. 2. Tests with simple spherical systems. *The Journal of Physical Chemistry B* **2007,** *111* (11), 2917-2924.
7. Gamieldien, M. R. Parameterization of the Gay-Berne Coarse-Grain Potential from atomistically detailed anisotropic free energy volumes. Doctoral Thesis, University of Cape Town, 2012.
8. Ponder, J. W.; Richards, F. M., An efficient newton-like method for molecular mechanics energy minimization of large molecules. *Journal of Computational Chemistry* **1987,** *8* (7), 1016-1024.
9. Brooks, B. R.; Bruccoleri, R. E.; Olafson, B. D.; States, D. J.; Swaminathan, S. a.; Karplus, M., CHARMM: a program for macromolecular energy, minimization, and dynamics calculations. *Journal of Computational Chemistry* **1983,** *4* (2), 187-217.
10. Allen, M. P.; Tildesley, D. J., *Computer simulation of liquids*. Oxford university press: 2017.
11. Swope, W. C.; Andersen, H. C.; Berens, P. H.; Wilson, K. R., A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: Application to small water clusters. *The Journal of Chemical Physics* **1982,** *76* (1), 637-649.
12. Berendsen, H. J.; Postma, J. v.; van Gunsteren, W. F.; DiNola, A.; Haak, J., Molecular dynamics with coupling to an external bath. *The Journal of Chemical Physics* **1984,** *81* (8), 3684-3690.
13. Abrahamsson, E.; Plotkin, S. S., BioVEC: a program for biomolecule visualization with ellipsoidal coarse-graining. *Journal of Molecular Graphics and Modelling* **2009,** *28* (2), 140-145.
14. Allen, M. P.; Germano, G., Expressions for forces and torques in molecular simulations using rigid bodies. *Molecular Physics* **2006,** *104* (20-21), 3225-3235.
15. NVIDIA Corporation CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (accessed 25-06-2018).
16. Chandler, D., *Introduction to Modern Statistical Mechanics*. Oxford University Press: 1987; p 288.
17. Nilsson, L., CHARMM Analysis Tools. 2006.
18. McCool, M.; Collings, A.; Woolf, L., Pressure and temperature dependence of the self-diffusion of benzene. *Journal of the Chemical Society, Faraday Transactions 1: Physical Chemistry in Condensed Phases* **1972,** *68*, 1489-1497.
19. Naidoo, K. J., FEARCF a multidimensional free energy method for investigating conformational landscapes and chemical reaction mechanisms. *Science China Chemistry* **2011,** *54* (12), 1962-1973.
20. Naidoo, K. J., Multidimensional free energy volumes offer unique insights into reaction mechanisms, molecular conformation and association. *Physical Chemistry Chemical Physics* **2012,** *14* (25), 9026-9036.
21. Gamieldien, M. R.; Strümpfer, J.; Naidoo, K. J., Hydration-Determined Orientational Preferences in Aromatic Association from Benzene Dimer Free Energy Volumes. *The Journal of Physical Chemistry B* **2011,** *116* (1), 324-331.

4. Look-up tables and interpolation algorithms

4.1 Introduction

Look-up tables (LUTs) have historically been very popular in mathematics and computer science. Any table with one or more indices mapping to unique value constitutes a LUT. This includes trigonometric and logarithmic tables found in many mathematics textbooks[1]. In computer science and engineering, the concept of table look-ups has led to many uses such as memory caching, databases with a search operation and spreadsheets. Most of these use the very basic principle of indexing, but many complex LUTs such as hash tables are used for improving performance.

While simple table lookups have been used to reduce the computation time of many complex operations, this can be taken a step further by completely replacing the computation with a table look-up and an interpolation. Interpolation algorithms can predict the trend of the data points in a table and provide an approximate solution based on that prediction. While the accuracy and precision vary between algorithms, an interpolation will nearly always contain an approximation error. However, the gain received in resources, execution time and efficiency may greatly outweigh the loss in accuracy for most algorithms. Furthermore, for data sets with no functional form, an interpolation algorithm can be used to estimate values between the data points.

Many MD simulation packages have made use of look-up tables and interpolations to improve the execution times and efficiency of the algorithms. In Chapter 2, the use of the GPU's in-built texture memory and linear interpolation for the look-up of Lennard-Jones forces in NAMD[2-3] was highlighted.

In the preceding chapters, the need for using a numerical free energy surface as the intermolecular potential in coarse-grained simulations was established. The free energy volumes used in this project were created via sampling in FEARCF[4-6] and have no functional form. The surfaces can be used as a LUT and the values in-between the grid points can be determined with the aid of an interpolation algorithm. The aim of this chapter is to investigate

a suitable interpolation algorithm for this purpose. Since these surfaces are four-dimensional (4D) in nature, the algorithm must be capable of multidimensional implementations.

This chapter begins with a brief discussion of look-up table examples and their uses. In-built hardware LUTs in FPGAs and GPUs are also highlighted. This is followed by a discussion on interpolation and its various types. Polynomial and cubic spline interpolation algorithms are analysed in detail, and the reasons for choosing the cubic spline interpolation for this project are offered. Finally, the chapter ends with a look at two examples of cubic spline interpolation algorithms that are further investigated in the thesis.

4.2 Examples and uses of look-up tables

The aim of using LUTs in most of their implementations is to reduce execution time and increase efficiency by replacing repetitive computation. The computation of non-trivial functions is generally more time consuming than reading from an array. In many cases, the indexing is not direct but is performed using a function or algorithm to map an index to the corresponding value in an array.

LUTs and similar data structures are used extensively in computer and computational science. Some commonly used types of LUTs are briefly discussed below.

4.2.1 Lists and indexing

Lists are the simplest form of LUTs. In many programs, a list is implemented using an array that acts as a quick access storage location for commonly used values. A set of indices can be mapped one-to-one or many-to-one (for multidimensional data) to a value stored in an array. When the values in the arrays are known, a simple array indexing operation can return the value immediately. When the arrangement is not known, however, search operations need to be employed.

Databases and spreadsheets are more sophisticated forms of lists where the data can be queried using keys that can be integer values or strings.

4.2.2 Caching and Memoization

A cache is an on-chip hardware or software structure that stores the original address and data of frequently used locations on off-chip memory. Off-chip memory like DRAM is large but very slow to access. To reduce computation time, caches store values that the compiler or programmer determines will be used often. On a cache miss, i.e. when a value is not present in the cache, the expensive off-chip memory fetching operation needs to be performed. However, this will be stored in the cache and a future call for the same location will result in a cache hit. Hardware caches are used by processors like CPUs[7] and GPUs[8]. A web browser cache is an example of software caching.

Memoization[9-10] is a technique where data from computationally expensive function calls is stored in a LUT for future re-use to avoid computing the value again. It is related to caching, however, instead of storing commonly used data from a slower storage location like DRAM, the data is computed in runtime and stored on-the-fly when required. When a value is computed, the LUT is accessed to check if the value is present. If it is not, the value is computed and stored in the LUT. Memoization is used in functional programming languages like Haskell[11-12] and top-down parsing[10] to reduce the computational costs in the algorithms of these platforms and increase efficiency.

4.2.3 Hash tables and functions

Hash tables use complex algorithms known as hash functions to compute a unique key to map to a value. The most ideal hash function distributes the key values uniformly and has no collisions, i.e. no key has two values it can map to. In a realistic scenario, collisions may occur, and a hash table needs to be specifically designed to handle such a condition. Optimization techniques are used to handle collisions.

Hash tables are commonly used to implement associative arrays: tables of keys and values, where a unique key, that is separate from the index, is mapped to a value in the table. Examples of associative arrays include dictionaries and maps. They are also used for indexing operations in databases and caches which were mentioned earlier.

4.2.4 Hardware-based LUTs

LUTs are hard-coded on some hardware to reduce computations of commonly used functions on these platforms. Caches, which can be hardware-based, were discussed in Section 4.2.2. Hardware LUTs on FPGAs and GPUs are discussed below.

i. Programmable logic LUTs in FPGAs

Look-up tables are commonly found on FPGAs as part of the Configurable Logic Block (CLB). They are implemented using multiplexers and can be multidimensional i.e. use more than one input to map to a single value. These values can be hard-coded for common functions such as sines and cosines or be reconfigurable with the aid of D-latches. Boolean functions can be accelerated by implementing a truth table as a LUT.

ii. LUTs in GPUs for image processing

LUTs have been used extensively in graphics processing in 1D, 2D and 3D forms. They are used for the acceleration of colour operations and reduced their computationally complexity and time[13]. Three 1D LUTs, one for each colour, are used for simple modifications, such as brightness, gamma and contrast. More complex colour operations require higher dimensionality and more complex interpolation algorithms. 3D LUTs are used to accelerate operations such as shadow, hue and saturation.

LUTs are used in conjunction with interpolation (which will be discussed in the next section) to determine data in between the values stored on the LUT. The most commonly used forms of interpolation used in graphics are the nearest-neighbour and linear interpolation (or trilinear interpolation in the case of 3D operations)[13].

LUTs and linear interpolation are hard-coded in most GPUs as texture look-ups. Both images and non-graphical data can be loaded into texture memory and the texture fetching operations can be configured to perform a “direct” look-up using nearest-neighbour interpolation or linear filtering which applies linear interpolation. Both the LUTs and interpolation methods on GPUs are available up till 3D.

4.3 Interpolation algorithms

A computationally intensive function can be replaced by numerical data using a LUT to reduce the cost of the calculation. The output from a set of input values can be computed and stored in

an array to act as a LUT and the values can be directly looked-up instead of being computed. However, the technique fails for values that fall in between the data points in the table. While these can be computed on-the-fly and stored again for future use (i.e. memoization), it will reduce the computation efficiency of the algorithm.

The alternative solution is the use of interpolation algorithms. Interpolation can be described as a numerical method used for the determination of values between an available set of discrete data points. A computationally intensive function can be replaced with a look-up table and a "simpler" interpolation algorithm. Any gains in computational speed and efficiency may, however, come at the cost of accuracy.

As described in Section 4.2.4(ii), interpolation has been used for image filtering and correction to smooth rougher images using pixel data. However, its use is even more essential when determining values from a set of scattered data with no functional form. In this case, the interpolation algorithm acts as the function to return values between the data points as accurately as possible. In this section, some popular interpolation algorithms and their advantages and disadvantages are discussed.

4.3.1 Nearest-neighbour interpolation

Nearest-neighbour is the most basic type of interpolation[14]. The technique simply returns the value closest to the index being looked-up. For example, if a table contains index values ranging from 1 to 10 at a step size of 1, depending on the algorithm, an index of 5.6 can be mapped to 6.0 and the value of that index will be returned.

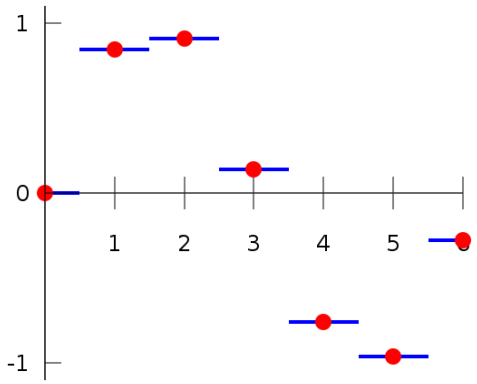


Figure 4.1: A 1D nearest-neighbour interpolation (blue) around the data points (red)[15†]

† "File:Piecewise_constant.svg" by Berland is available in the Public Domain.

Nearest-neighbour is computationally simple and is almost equivalent to a direct read from an array. However, not only is the function piecewise and discontinuous, it is also particularly inaccurate for multidimensional data and unusable if derivatives are required.

4.3.2 Linear interpolation

Linear interpolation is one of the most commonly used interpolation forms for basic data[16]. Unlike nearest-neighbour, linear interpolation computes a new value for any data point that lies between the ones present in an array or LUT.

In 1D, for a given value x_q that falls in the range of x values, the corresponding y_q value is computed using Equation 4.1:

$$y_q = y_i + (y_{i+1} - y_i)\frac{(x_q - x_i)}{(x_{i+1} - x_i)} \tag{4.1}$$

As the name suggests, the function has a linear form. Interpolation takes place in between two data points (x_i, y_i) and (x_{i+1}, y_{i+1}) where the x_q value falls. For multi-dimensional data, linear interpolation is performed for each dimension iteratively to obtain the final value.

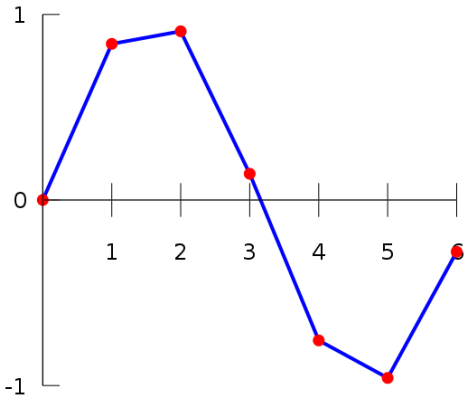


Figure 4.2: Linear interpolation (blue) between data points (red)[17†]

Linear interpolation is continuous and more accurate than the nearest-neighbour interpolation. However, it is also not differentiable. This makes it unsuitable for data where derivatives are required.

† "File:Interpolation example linear.svg" by Berland is available in the Public Domain.

4.3.3 Polynomial interpolation

Higher order polynomial interpolation is used for data that requires more accuracy than that offered by its linear variant. For a given set of n data points, a polynomial of $n-1$ degree can fit through them[18]:

$$y = c_{n-1}x^{n-1} + c_{n-2}x^{n-2} + c_{n-3}x^{n-3} + \cdots + c_1 x + c_0 \tag{4.2}$$

High-order polynomial interpolations are not only continuous but differentiable, making them much more suitable than linear interpolation

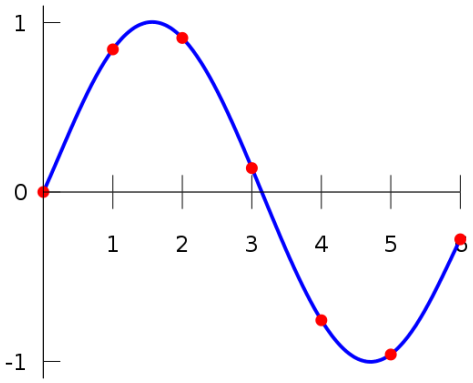


Figure 4.3: A polynomial (blue) passing through the data points (red). Note the higher accuracy in approximation with no discontinuities[19]†

With higher accuracy and differentiability comes higher computational costs. The polynomial may also exhibit oscillatory effects at endpoints, known as the Runge's phenomenon[20].

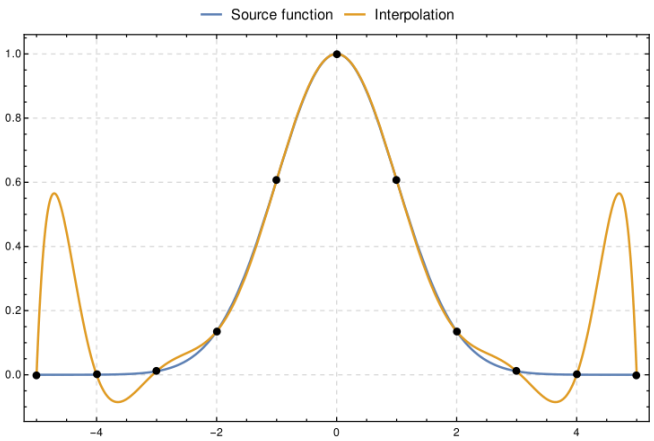


Figure 4.4: Runge's phenomenon with a high-order Lagrange polynomial[21]‡

† "File:Interpolation_example_polynomial.svg" by Berland is available in the Public Domain.
‡ "File:Runge's phenomenon in Lagrange polynomials.svg" by Glosser.ca is licensed under CC BY-SA 4.0

As opposed to linear interpolation, for approximating simpler functions or data where accuracy is not a major factor, the disadvantages of polynomial interpolation may outweigh the advantages.

Lagrange and Newton polynomials are commonly used for interpolation. The following is a brief overview of the two polynomials.

i. Lagrange Polynomials

For a set of n data points in a table, a Lagrange polynomial[16, 22] of less than $(n-1)$ degree that passes through all the points in the table is given by the following linear combination[22]:

$$P^L(x) = \sum_{j=0}^{n} y_j P_j^L(x) \tag{4.3}$$

Where the Lagrange basis polynomial $P_j^L(x)$ is given by:

$$P_j^L(x) = \prod_{\substack{k=0 \\ k \neq j}}^{n} \frac{x_q - x_k}{x_j - x_k} \tag{4.4}$$

For the 1D case, the tabulated data points must have a one-to-one mapping for the basis functions to be non-zero. The Lagrange polynomial is very susceptible to Runge's phenomenon. It is, therefore, essential to choose a polynomial degree that can accurately predict the data with as few oscillations and errors as possible.

ii. Newton Polynomials

Like Lagrange polynomials, Newton polynomials are a linear combination[22]:

$$P^N(x) = \sum_{j=0}^{n} a_j P_j^N(x) \tag{4.5}$$

Unlike Lagrange, however, the coefficients of the polynomials a_j are divided differences for the values of y until the current index:

$$a_j = [y_0, \dots, y_j] \tag{4.6}$$

The divided differences can be computed recursively or with the aid of an upper triangular matrix. The Newton basis polynomials themselves are given by ($j > 0$ and $P_0^N = 0$):

$$P_j^N(x) = \prod_{i=0}^{j-1}(x_q - x_i) \tag{4.7}$$

As compared to Lagrange, Newton polynomials are harder to compute due to the use of divided difference. However, it is much easier to add new terms to the Newton polynomial if new data points are added to the table, unlike Lagrange polynomials where the entire polynomial needs to be re-computed[22].

4.3.4 Spline interpolation

A spline is a special type of polynomial which is piecewise in nature[23-24]. Due to this, they are resistant to Runge's phenomenon and are much more flexible, enabling them to pass through the data points while ensuring there are no discontinuities and unreasonable behaviour.

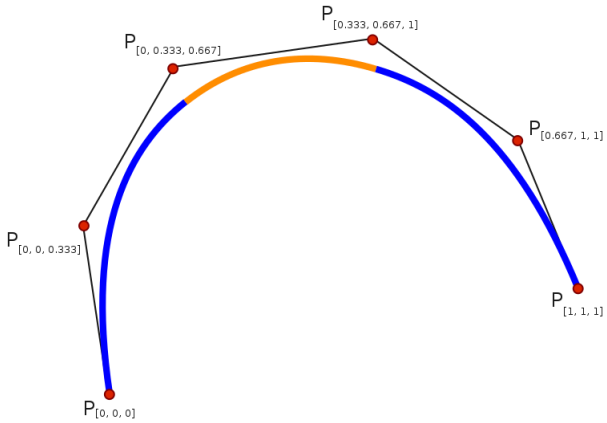


Figure 4.5: A cubic spline approximating a set of data points[25†]

Interpolation is performed by forming a spline curve between consecutive pairs or alternating sets of data points known as knots. A general spline function is provided in Equation 4.8.

† File:Parametic Cubic Spline.svg by User:Garry R. Osgood is licensed under CC BY-SA 3.0

$$S(x) = \begin{cases} S_0(x), & x_0 \leq x \leq x_1 \\ S_1(x), & x_1 \leq x \leq x_2 \\ \vdots & \vdots \\ S_{n-2}(x), & x_{n-2} \leq x \leq x_{n-1} \\ S_{n-1}(x) & x_{n-1} \leq x \leq x_n \end{cases} \quad (4.8)$$

$S_i(x)$ are polynomials of the chosen degree. Despite being piecewise in nature, spline interpolation ensures continuity and differentiability at the data points.

i. Cubic spline interpolation

The third degree or cubic spline is the most commonly used version of spline interpolation since it is twice differentiable and continuous[23-24]. As the name suggests, for a cubic spline, the polynomials making up each case of the piecewise function, Equation 4.8, will be cubic[26]:

$$S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3 \text{ for } x_i \leq x < x_{i+1} \quad (4.9)$$

The interpolation must ensure that at the data points of a set of tabulated values, the corresponding value is returned by the function. Furthermore, the value and the first and second order derivatives of two consecutive piecewise polynomials in the spline must be equal at their shared knot. This ensures the continuity and differentiability of the spline and interpolation. For differentiability at the endpoints of the tabulated values, the spline is clamped to zero (natural cubic spline) or a known value if available[18].

The data points and these conditions result in a system of equations to compute the coefficients of the polynomials in Equations 4.8 and 4.9. The tridiagonal algorithm is generally used to solve this set of equations[27-28]. An alternative version of the algorithm is computed using basis functions[28-29].

4.4 Comparing interpolation algorithms

In choosing the right interpolation for the LUT, it is necessary to look at three main aspects of the algorithms: their computation complexity, multidimensionality and potential parallelism. The aim of the project is to implement a parallel, 4D LUT and the chosen algorithm needs to meet these criteria.

4.4.1 Computation complexity

The cubic spline interpolation has two steps: computation of the coefficients and interpolation. The tridiagonal algorithm used for the computation of the coefficients has the complexity of $O(N^2)$[30]. As established in the case of MD simulations, this complexity is inefficient. However, it is only called once before any interpolations are performed. Further, optimized versions of the routine are available from linear algebra libraries like BLAS[31] and LAPACK[32]. Once the coefficients are known, a single interpolation is generally performed in $O(n)$ time[18].

Both Lagrange and Newton polynomials involve two sets of accumulation which computationally will be translated to a double nested-loop. This means they generally have the inefficient computational complexity of $O(N^2)$[33] when computing their coefficients. However, like the cubic spline interpolation, their actual interpolation can be performed in $O(N)$ time[33].

4.4.2 Multidimensionality

Multidimensionality is the main section where the cubic spline interpolation has a major advantage. 2D and higher polynomial interpolations are not simple algorithms, and while attempts at their implementation have been made[34-35], they are not commonly used.

On the other hand, cubic spline interpolation can be easily used for multivariate interpolation[28, 35]. The bicubic spline interpolation is frequently documented and used in many applications[18, 28-29, 35].

4.4.3 Parallelism

Many parallel implementations to reduce the overall complexity of the algorithms have been devised. In the case of polynomial interpolation, this reduces the complexity to $O(\log N)$[36-37]. An algorithm implementing a parallel cubic spline interpolation has also shown a reduction in complexity to $O(1)$ on a GPU[38].

4.4.4 Conclusion

Overall, both polynomial interpolation and cubic spline interpolation are popular for their high accuracy in estimating between the data points, continuity and differentiability. The cubic spline interpolation is not, however, susceptible to Runge's phenomenon like the polynomial ones. While some of the interpolation algorithms can benefit from parallelization, the motivation in choosing the cubic spline interpolation as the algorithm of choice for the LUT

used in this project has much to do with its potential for multidimensionality which is essential to this work.

4.5 Cubic spline interpolation algorithms

In this section, two variants of the cubic spline interpolation algorithms explored in this thesis are discussed.

4.5.1 A special case of the Lagrange Polynomial

In their FORTRAN-based implementation, William H. Press et al. provided the following general equations for a cubic spline interpolation and its derivatives[18]:

$$y = Ay_i + By_{i+1} + Cy_i" + Dy_{i+1}" \tag{4.10}$$

$$\frac{dy}{dx} = \frac{y_{i+1} - y_i}{x_{i+1} - x_i} - \frac{3A^2 - 1}{6}(x_{i+1} - x_i)y_i" + \frac{3B^2 - 1}{6}(x_{i+1} - x_i)y_{i+1}" \tag{4.11}$$

$$\frac{d^2y}{dx^2} = Ay_i" + By_{i+1}" \tag{4.12}$$

Where:

$$A = \frac{x_{i+1} - x_q}{x_{i+1} - x_i};\ B = 1 - A;\ C = \frac{1}{6}(A^3 - A)(x_{i+1} - x_i)^2;\ D = \frac{1}{6}(B^3 - B)(x_{i+1} - x_i)^2 \tag{4.13}$$

(x_a, y_b) and (x_b, y_b) are the two consecutive points forming an interval between them. The equations are derived from a special case of Lagrange polynomials. For this derivation, the authors had to assume that the second derivatives are known. Further, by insuring that the first derivatives of the splines are continuous and differentiable across the boundaries of the intervals (to ensure the second derivative exists), the authors provided the following system of equations:

$$\frac{x_i - x_{i-1}}{6}y_{i-1}" + \frac{x_{i+1} - x_{i-1}}{3}y_i" + \frac{x_{i+1} - x_i}{6}y_{i+1}" = \frac{y_{i+1} - y_i}{x_{i+1} - x_i} - \frac{y_i - y_{i-1}}{x_i - x_{i-1}} \tag{4.14}$$

The equations exist for $i = 2, \dots, N-1$ and can be solved using the tridiagonal algorithm to obtain the series of $y_i"$ values.

To obtain unique solutions, the second derivative of one or both endpoints of the entire tabulated data needs to be set, since they are not covered by the system of equations (4.14). If the second derivatives are not known, this can be done in one of two ways:

- Setting either or both endpoints to values computed using Equation 4.11
- Setting the endpoints to 0, giving the natural cubic spline.

Along with one of the boundary conditions specified above, the equations take the form of a tridiagonal matrix.

The FORTRAN code provided in the book "Numerical Recipes in FORTRAN"[18] splits the process into two functions:

- A function implementing the tridiagonal algorithm and boundary conditions to compute the second derivatives which are only called once.
- A function that performs the interpolation using a given value of x_q, the computed values from the above routine and the original data table. Optimization techniques are employed to ensure a quick computation.

The book[18] also offers a bicubic spline interpolation algorithm.

4.5.2 Interpolation using B-splines

An alternative and popular method for cubic spline interpolation is using B-splines i.e. basis spline functions. A d-dimensional spline can be constructed from the linear combination of B-splines of the same dimension. Like regular splines, B-splines are also piecewise in nature. The zeroth-degree B-spline is given by the "hat" or shifted step function[28-29]:

$$\beta_0(x) = \begin{cases} 1 & |x| \leq \frac{1}{2} \\ 0 & elsewhere \end{cases} \tag{4.15}$$

The zeroth dimensional spline is an example of a nearest-neighbour interpolation. Higher-dimensional B-splines are obtained using repeated convolution of $\beta_0(x)$ with itself, or computed using the Cox-de Boor recursion formula[39].

The third-degree B-spline is given by:

$$\beta^3(x) = \begin{cases} (2-|x|)^3 & 1 \le |x| \le 2 \\ 4 - 6|x|^2 + 3|x|^3 & |x| < 1 \\ 0 & elsewhere \end{cases} \tag{4.16}$$

Using the basis function, a spline in d-degree is computed using a linear combination:

$$s_d(x) = \sum_k c_k \beta_k^d(x) \tag{4.17}$$

c_k is the set of coefficient values computed from the tabulated data points to correspond with the basis functions.

In their design of a fast, general multidimensional cubic spline interpolation, Kindermann and Habermann[28] use B-splines. The technique requires the index values or dependent variables in the data table to be equidistant. However, a re-scaling function can be applied to non-equidistant nodes prior to applying the same algorithm, provided they fulfil the required criteria. Any index value x_i in the table can then be expressed as:

$$x_i = a + ih \tag{4.18}$$

Where a is the smallest value of x_i in the table and h is the stride. h is computed using:

$$h = \frac{b-a}{h} \tag{4.19}$$

where b is the largest value of x_i in the table. Cubic splines created with this technique exist in the $n+3$ space:

$$s_3(x) = \sum_{k=1}^{n+3} c_k \beta_k^3(x) \tag{4.20}$$

Therefore, the number of coefficients to be computed is $n+3$. The value being queried, x_q also needs to be adjusted to work with this equation:

$$t_k = \left(\frac{x_q - a}{h} - (k-2)\right) \text{ for } k = 1, \ \ldots\ldots, n+3 \tag{4.21}$$

t_k is then input into Equation 4.16 to compute the value of the base for each value of k. $\beta_k^3(x)$ vanishes outside the bounded interval of $[x_{k-4}, x_k] \cap [a, b]$ since $supp(\beta_k^3) = [x_{k-4}, x_k] \cap [a, b]$. So, the equation of the spline can be further re-written as:

$$s_3(x) = \sum_{k=1}^{m} c_k \, \beta_k^3(x), \quad l = \left\lfloor \frac{x_q - a}{h} \right\rfloor + 1, \qquad m = \min(l+3, n+3) \tag{4.22}$$

$s(x_i) = y_i, \ i = 0, \ldots\ldots, n$ gives $n+1$ conditions, but $n+3$ conditions are required. The remaining two conditions are regarding the second derivatives at the end points of the data set:

$$s"(\mathrm{a}) = v, \quad s"(b) = w \tag{4.23}$$

The values for v and w can be explicily chosen or set to 0 to give the natural cubic spline. The $n+3$ conditions are therefore:

$$s_3"(x_0) = \sum_{k=1}^{3} c_k \, \beta_{3,k}"(x_0) = v \tag{4.24}$$

$$s_3"(x_n) = \sum_{i=n+1}^{n+3} c_k \, B_{3,k}"(x_n) = w \tag{4.25}$$

$$s_3(x_i) = \sum_{k=1}^{m} c_k \, \beta_{3,k}(x_i) = y_i \tag{4.26}$$

Where:

$$l = \left\lfloor \frac{x_i - a}{\mathrm{h}} \right\rfloor + 1 \text{ and } m = \min(l+3, n+3), \quad i = 0, \ldots\ldots, n$$

As with the previous algorithm, these conditions make up a system of equations in tridiagonal form, which can be solved to obtain the coefficient values c_k for $k = 1, \ldots\ldots, n+3$. Using the coefficient values and the computation of the basis function, for any $x \in [a, b]$ the corresponding $s_3(x_q) = y_q$ can be found.

This technique is meant to be scalable to any dimension, provided the available resources allow it[28].

4.6 Concluding remarks

Based on the explored literature, the cubic spline interpolation was selected as the algorithm of choice for the look-up table routine for the Free Energy Force Induced (FEFI) CG MD simulation package[40]. In the following chapter, the implementation of three variants of the four-dimensional cubic spline interpolation on the CPU and GPU will be evaluated for their potential in achieving the objectives of this thesis.

4.7 References

1. Campbell-Kelly, M.; Croarken, M.; Flood, R.; Robson, E., *The history of mathematical tables: from Sumer to spreadsheets*. Oxford University Press: 2003.
2. Nelson, M. T.; Humphrey, W.; Gursoy, A.; Dalke, A.; Kalé, L. V.; Skeel, R. D.; Schulten, K., NAMD: a parallel, object-oriented molecular dynamics program. *The International Journal of Supercomputer Applications and High Performance Computing* **1996,** *10* (4), 251-268.
3. Stone, J. E.; Phillips, J. C.; Freddolino, P. L.; Hardy, D. J.; Trabuco, L. G.; Schulten, K., Accelerating molecular modeling applications with graphics processors. *Journal of Computational Chemistry* **2007,** *28* (16), 2618-2640.
4. Naidoo, K. J., FEARCF a multidimensional free energy method for investigating conformational landscapes and chemical reaction mechanisms. *Science China Chemistry* **2011,** *54* (12), 1962-1973.
5. Naidoo, K. J., Multidimensional free energy volumes offer unique insights into reaction mechanisms, molecular conformation and association. *Physical Chemistry Chemical Physics* **2012,** *14* (25), 9026-9036.
6. Gamieldien, M. R.; Strümpfer, J.; Naidoo, K. J., Hydration-Determined Orientational Preferences in Aromatic Association from Benzene Dimer Free Energy Volumes. *The Journal of Physical Chemistry B* **2011,** *116* (1), 324-331.
7. Intel Corporation Intel® Xeon® Processor E5-2690 v3. https://ark.intel.com/products/81713/Intel-Xeon-Processor-E5-2690-v3-30M-Cache-2-60-GHz- (accessed 26-06-2018).
8. NVIDIA Corporation, NVIDIA TESLA V100 GPU Accelerator - Datasheet. 2018.
9. Michie, D., *"Memo" Functions and Machine Learning*. 1968.
10. Norvig, P., Techniques for automatic memoization with applications to context-free parsing. *Computational Linguistics* **1991,** *17* (1), 91-98.
11. Marlow, S. Haskell 2010 language report. https://www.haskell.org/onlinereport/haskell2010/ (accessed 1-07-2018).
12. Jones, S. P., *Haskell 98 language and libraries: the revised report*. Cambridge University Press: 2003.
13. Selan, J., *Using lookup tables to accelerate color transformations*. 2004; Vol. 2, p 381-392.
14. Parker, J. A.; Kenyon, R. V.; Troxel, D. E., Comparison of interpolating methods for image resampling. *IEEE Transactions on Medical Imaging* **1983,** *2* (1), 31-39.

15. Berland, File:Piecewise constant.svg. In *gnuplot*, constant.svg, P., Ed. Wikimedia Commons, 2007; pp Illustration of constant interpolation, more precisely en:Nearest neighbor interpolation on the same dataset as listed in Interpolation.
16. Meijering, E., A chronology of interpolation: from ancient astronomy to modern signal and image processing. *Proceedings of the IEEE* **2002,** *90* (3), 319-342.
17. Berland, File:Interpolation example linear.svg. In *gnuplot*, linear.svg, I. e., Ed. 2007; p Illustration of linear interpolation on a data set. The same data set is used for other interpolation methods in the interpolation article.
18. Press, W. H., *FORTRAN Numerical Recipes: Numerical recipes in FORTRAN 90: the art of parallel scientific computing*. Cambridge University Press: 1996; Vol. 2.
19. Berland, File:Interpolation example polynomial.svg. In *gnuplot*, polynomial.svg, I. e., Ed. Wikimedia Commons, 2007; p IIllustration of polynomial interpolation of a data set. The same data set is used for other interpolation algorithms in the Interpolation.
20. Runge, C., Über empirische Funktionen und die Interpolation zwischen äquidistanten Ordinaten. *Zeitschrift für Mathematik und Physik* **1901,** *46* (224-243), 20.
21. Glosser.ca, File:Runge's phenomenon in Lagrange polynomials.svg. polynomials.svg, R. s. p. i. L., Ed. Wikimedia Commons, 2016; p English: The polynomial interpolation of samples taken from a Gaussian distribution displays significant ringing near the endpoints.
22. Werner, W., Polynomial interpolation: Lagrange versus newton. *Mathematics of Computation* **1984**, 205-217.
23. Schoenberg, I. J., Contributions to the problem of approximation of equidistant data by analytic functions. In *IJ Schoenberg Selected Papers*, Springer: 1988; pp 3-57.
24. Schoenberg, I. J., *Cardinal spline interpolation*. Siam: 1973; Vol. 12.
25. Osgood, G. R., File:Parametic Cubic Spline.svg. Spline.svg, P. C., Ed. Wikimedia Commons, 2008.
26. Mohanty, P. K.; Reza, M.; Kumar, P.; Kumar, P. In *Implementation of cubic spline interpolation on parallel skeleton using pipeline model on CPU-GPU cluster*, Advanced Computing (IACC), 2016 IEEE 6th International Conference on, IEEE: 2016; pp 747-751.
27. Allen, M. P.; Germano, G., Expressions for forces and torques in molecular simulations using rigid bodies. *Molecular Physics* **2006,** *104* (20-21), 3225-3235.
28. Habermann, C.; Kindermann, F., Multidimensional spline interpolation: Theory and applications. *Computational Economics* **2007,** *30* (2), 153-169.
29. Thévenaz, P.; Blu, T.; Unser, M., Interpolation revisited [medical images application]. *IEEE Transactions on Medical Imaging* **2000,** *19* (7), 739-758.
30. Ran, R.-s.; Huang, T.-z.; Liu, X.-p.; Gu, T.-x., An inversion algorithm for general tridiagonal matrix. *Applied Mathematics and Mechanics* **2009,** *30* (2), 247-253.
31. Xianyi, Z.; Qian, W.; Saar, W., OpenBLAS-an optimized BLAS library. *Accedido: Agosto* **2016**.
32. Anderson, E.; Bai, Z.; Dongarra, J.; Greenbaum, A.; McKenney, A.; Du Croz, J.; Hammarling, S.; Demmel, J.; Bischof, C.; Sorensen, D. In *LAPACK: A portable linear algebra library for high-performance computers*, Proceedings of the 1990 ACM/IEEE Conference on Supercomputing, IEEE Computer Society Press: 1990; pp 2-11.
33. Berrut, J.-P.; Trefethen, L. N., Barycentric lagrange interpolation. *SIAM Review* **2004,** *46* (3), 501-517.
34. Saniee, K. *A simple expression for multivariate Lagrange interpolation*; 2008.
35. Ruijters, D.; ter Haar Romeny, B. M.; Suetens, P., Efficient GPU-based texture interpolation using uniform B-splines. *Journal of Graphics Tools* **2008,** *13* (4), 61-69.
36. Sarbazi-Azad, H.; Ould-Khaoua, M.; Mackenzie, L. M.; Akl, S. G., A parallel algorithm for Lagrange interpolation on the star graph. *Journal of Parallel and Distributed Computing* **2002,** *62* (4), 605-621.

37. Eğecioğlu, Ö.; Gallopoulos, E.; Koç, Ç. K., Parallel Hermite interpolation: An algebraic approach. *Computing* **1989,** *42* (4), 291-307.
38. Nyiri, E.; Gibaru, O.; Auquiert, P. In *Nonlinear L 1 C 1 interpolation: application to images*, International Conference on Curves and Surfaces, Springer: 2010; pp 515-526.
39. de Boor, C. *Subroutine Package For Calculating With B-Splines*; Los Alamos Scientific Lab., N. Mex.: 1971.
40. Gamieldien, M. R. Parameterization of the Gay-Berne Coarse-Grain Potential from atomistically detailed anisotropic free energy volumes. Doctoral Thesis, University of Cape Town, 2012.

5. Implementing the 4-dimensional look-up table with a cubic B-spline interpolation

5.1 Introduction

In Chapter 3, the limitations of the Gay-Berne potential[1-2] function to simulate liquid benzene was demonstrated. A conclusion was reached at the end of that chapter suggesting that instead of relying on analytical potential energy functions such as Gay-Berne, a computational means of integrating free energy from a numerical potential energy function should be investigated. This approach is consistent with the theoretical premise of CG models where free energy is used as a source of a more accurate interaction potential function[3]. This chapter focusses on the development of a look-up table (LUT) as an alternative to the Gay-Berne based CGGB module in coarse-grained simulations of liquid benzene. The LUT comprises 4D free energy arrays generated from the FEARCF package[4-6] to be used as the intermolecular numerical potential in the coarse-grained Free Energy Force Induced (FEFI) MD package[7].

As highlighted in Chapter 4, a look-up table can be used with an interpolation algorithm to estimate between the grid points of a function-less dataset like the free energy arrays. The cubic spline interpolation was selected for its efficiency, accuracy and potential for multidimensional applications. Two versions of the algorithm were discussed for one-dimensional data. However, since the FEARCF arrays are four-dimensional (4D), this implementation is the focus of this study.

In this chapter, the three cubic spline interpolation algorithms will be expanded upon to their four-dimensional versions. The algorithms will all use the same 4D arrays built using a 4D polynomial for evaluation. The polynomial and its derivatives will be used to assess the accuracy of the results. The computational performance of the algorithms on a GPU will be compared to their respective CPU-based “golden measure” solutions. Using these metrics, the most suitable algorithm will be chosen for integration into the FEFI package to build its LUT module.

5.2 Algorithm 1: Cubic Spline Interpolation

The first interpolation algorithm is derived from the book "Numerical Recipes in FORTRAN"[8]. The algorithm was previously discussed in Chapter 4 and adapted by researchers at the Scientific Computing Research Unit to compute energies and forces from numerical potentials. Variations of this algorithm have been implemented into packages for large scale molecular computations[4, 6]. It is an accurate algorithm designed for single threaded sequential execution on the CPU. However, that also made it slow for the large amount of interpolations per time step required in FEFI. In my previous work[9], the algorithm was explored for its suitability for a GPU-based adaptation. The implementation and its issues are discussed in this section.

5.2.1 Review of the 1D case

In Chapter 4, the equations and algorithm for a 1D cubic spline interpolation were discussed. The process is divided into two parts:

1) The first step is determining the second derivatives via pre-defined values for the end cases (zero or a specific computed value) and the set of equations defined for $i = 2, \dots, N-1$:

$$\frac{x_i - x_{i-1}}{6} f_{i-1}" + \frac{x_{i+1} - x_{i-1}}{3} f_i" + \frac{x_{i+1} - x_i}{6} f_{i+1}" = \frac{f_{i+1} - f_i}{x_{i+1} - x_i} - \frac{f_i - f_{i-1}}{x_i - x_{i-1}} \qquad (5.1)$$

   These two conditions form a system of equations that can be solved using the tridiagonal algorithm to obtain the second derivatives.

2) The second step involves computing the coefficients:

$$A = \frac{x_{i+1} - x_q}{x_{i+1} - x_i};\ B = 1 - A;\ C = \frac{1}{6}(A^3 - A)(x_{i+1} - x_i)^2;\ D = \frac{1}{6}(B^3 - B)(x_{i+1} - x_i)^2 \qquad (5.2)$$

   and plugging in the coefficients and second derivatives into Equation 5.3 to obtain the interpolated value:

$$y = Ay_i + By_{i+1} + Cy_i" + Dy_{i+1}" \qquad (5.3)$$

   and into Equation 5.4 to get the first derivative:

$$\frac{dy}{dx} = \frac{y_{i+1} - y_i}{x_{i+1} - x_i} - \frac{3A^2 - 1}{6}(x_{i+1} - x_i)y_i" + \frac{3B^2 - 1}{6}(x_{i+1} - x_i)y_{i+1}" \quad (5.4)$$

5.2.2 Performing a 4D interpolation

The 4D case is a complex method that makes use of the 1D cubic spline interpolation multiple times to interpolate a final value. For a given 4D set of input values: (x_q, y_q, z_q, w_q), the interpolation is performed through a "reduction" process illustrated in Figure 5.1:

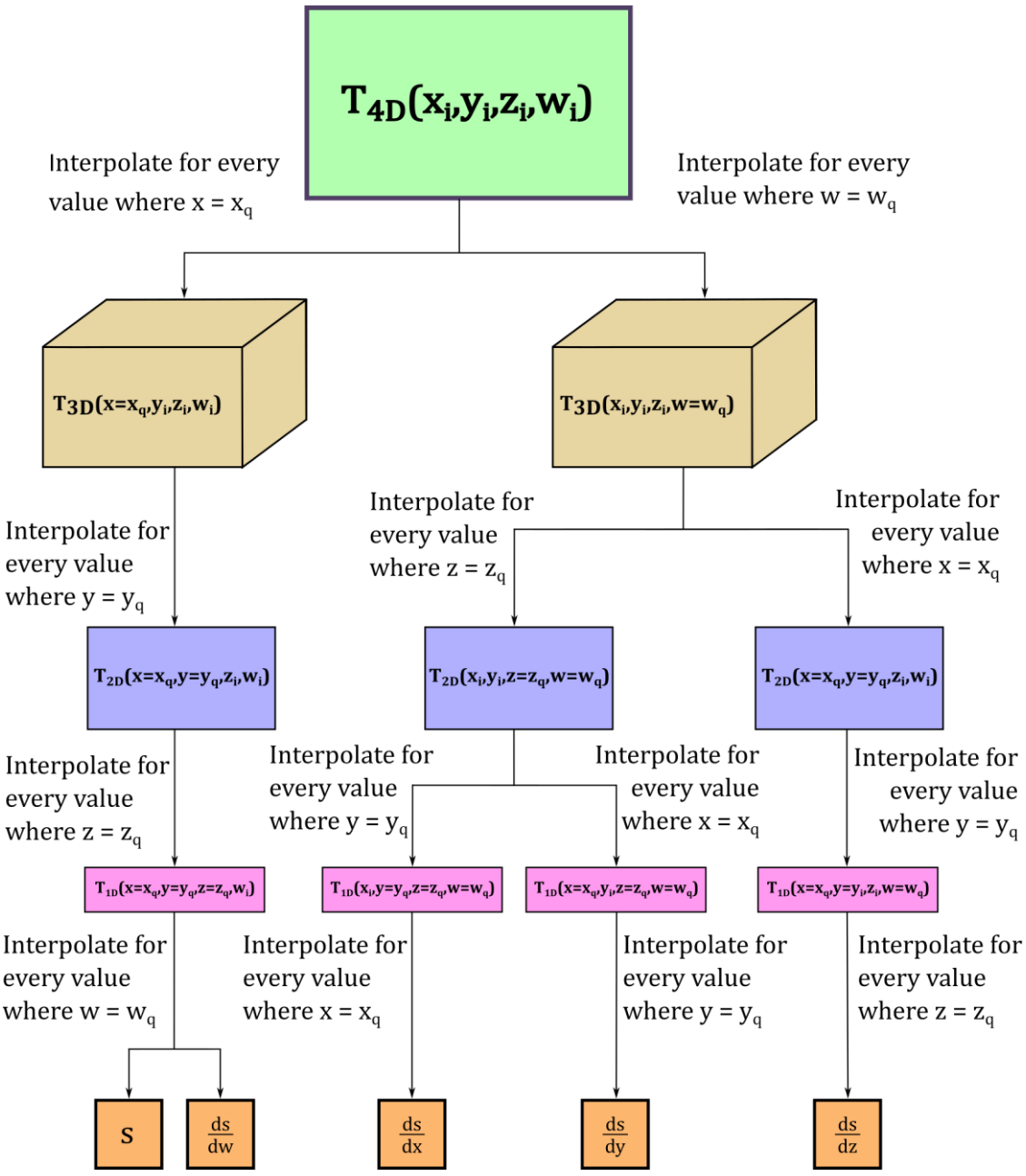


Figure 5.1: Reducing the 4D array using multiple cubic spline interpolations

Each of the branches in Figure 5.1 follows a pathway to obtain the interpolated value and its partial derivatives. In the first branch:

- The 4D array, $T_{4D}(x_i, y_i, z_i, w_i)$, is used to perform 1D interpolations for all values where $x = x_q$ and a combination of the remaining indices, (y_i, z_i, w_i), in the table. The result of this operation is a 3D array: $T_{3D}(x = x_q, y, z, w)$, to be used for further interpolations.
- The 3D array is then queried for $y = y_q$ and (z_i, w_i). The interpolations produce a 2D array: $T_{2D}(x = x_q, y = y_q, z, w)$. The step is again repeated for $z = z_q$ and w_i. giving $T_{1D}(x = x_q, y = y_q, z = z_q, w)$.
- Finally, $s(x_q, y_q, z_q, w_q)$ and $\frac{ds}{dw}(x_q, y_q, z_q, w_q)$ are obtained from the final interpolations for $w = w_q$.

The remaining branches in Figure 5.1 follow the same steps using a different order of the variables (x_q, y_q, z_q, w_q), depending on which partial derivative is being queried.

5.2.3 Implementation

From Figure 5.1, it can be determined that not only is the 4D cubic spline interpolation a complex method, it is memory intensive as well. Overcoming this computational limitation is a challenge because the aim of the interpolation and the LUT is to not only accurately process numerous reaction coordinates, but to do this in a computational time that is competitive with the CGGB module.

In the sequential CPU-based version of CGGB, a double-loop assembles the interactions for every particle with particles in its neighbour list, as explained in Chapter 3. However, to observe the throughput of the interpolation algorithm, its CPU-based sequential "golden measure" solution processes a set of predefined reaction coordinates iteratively in a single loop. The GPU uses the embarrassingly parallel method of "calculation-per-interaction" described in Chapter 3 for the GPU-based GGGB module. Apart from this "high-level" difference, the CPU and GPU algorithms follow the same approach.

i. Performing the interpolation

At the start of the program, the 4D surface T_{4D} is read from a file and stored in a flattened 1D array. The grid-point indices (x_i, y_i, z_i, w_i) are each stored in a 1D array respectively. The algorithm and equations required for the 1D cubic spline interpolation are coded as functions (or device functions in the case of the GPU) which are called multiple times in the main 4D routine.

Given the complexity of the algorithm, as a small upgrade from the previous implementation[9], the interpolation was split between two kernels. The first kernel followed the first major branch in Figure 5.1, while the second kernel follows the second branch. This helped in reducing the amount of memory used per thread in the kernel. The pseudo-code for Function/Kernel 1 is provided in Listing 5.1. As evident in the pseudo-code, the outer-loops iterate through the different combinations of (y_i, z_i, w_i) to get to the specific 1D array with respect to x_i. 1D interpolation is performed on this array with respect to x_q :

```
For every set of 4D variables (xq,yq,zq,wq) being queried:
   - For all values of wi:
       o For all values of zi:
           ▪ For all values of yi:
               • Compute the second derivatives of the 1D array
                 derived from T4D
               • Interpolate for x=xq and store results in T3D
   - For all values of wi:
       o For all values of zi:
           ▪ Compute the second derivatives of 1D array derived from
             T3D
           ▪ Interpolate for y=yq and store results in T2D
   - For all values of wi:
       o Compute the second derivatives of the 1D array derived from
         T2S
       o Interpolate for z=zq and store results in T1D
   - Compute the second derivatives of T1D
   - Interpolate for the final value for w=wq and partial derivative with
     respect to w.
```

Listing 5.1: Pseudo-code for interpolating the first branch in Algorithm 1 on the CPU and GPU

The pseudo-code for Function/Kernel 2 is provided in Listing 5.2:

```
For every set of 4D variables (xq,yq,zq,wq) being queried:
   - For all values of xi:
       o For all values of yi:
           ▪ For all values of zi:
               • Compute the second derivatives of the 1D array
                 derived from T4D
               • Interpolate for w=wq and store results in T3D
   - For all values of xi:
       o For all values of yi:
```

```
            ▪ Compute the second derivatives of 1D array derived from
              T3D
            ▪ Interpolate for z=zq and store results in T2D
    - For all values of xi:
        o Compute the second derivatives of the 1D array derived from
          T2S
        o Interpolate for y=yq and store results in T1D
    - Compute the second derivatives of T1D
    - Interpolate the partial derivative with respect to x for x=xq.
      Process the remaining branches to obtain the remaining partial
      derivatives.
```

Listing 5.2: Pseudo-code for interpolating the second branch in Algorithm 1 on the CPU and GPU

ii. Performance analysis

To test the performance of the 4D cubic spline interpolation, a 4D array was created using a simple polynomial made up of four variables:

$$f(x, y, z, w) = x^4 - 2xy + 5y\sin(z) - 0.25xy^2w^2 \tag{5.5}$$

The grid points of this array were at:

$$x = [2.0: 0.5: 12.0]$$
$$y = [-1.0: 0.1: 1.0]$$
$$z = [-3.1416: 0.31416: 3.1416]$$
$$w = [-10: 0.5: 0.0]$$

The function is differentiable. The partial derivatives are:

$$\frac{\partial f}{\partial x}(x, y, z, w) = 4x^3 - 2y - 0.25y^2w^2 \tag{5.6}$$

$$\frac{\partial f}{\partial y}(x, y, z, w) = -2x + 5\sin(z) - 0.5xyw^2 \tag{5.7}$$

$$\frac{\partial f}{\partial z}(x, y, z, w) = 5y\cos(z) \tag{5.8}$$

$$\frac{\partial f}{\partial w}(x, y, z, w) = -0.5xy^2w \tag{5.9}$$

The function and its derivatives are used to check the accuracy of the results from the LUT.

Kernel structure analysis

The analysis of the kernel for the GPU implementation using the Nvidia Profiler is provided in Table 5.1:

	K1	K2
Threads per block	64	64
Registers per thread	44	44
Shared memory per block	20.0 B	20.0 B
Theoretical occupancy	50%	50%
Actual occupancy	3.1%	3.1%
FLOP (% of total execution count)	8%	8%
Inactive (% of total execution count)	22%	22%
Load/Store (% of total execution count)	35%	35%

Table 5.1: Kernel structure analysis of Algorithm 1

The theoretical occupancy of both kernels is 50% and enough to hide the GPU's latency. The number of threads per block for each kernel was chosen specifically to ensure the highest possible occupancy without causing any errors. The actual occupancies are extremely poor at 3.1% for both kernels.

Even though the register usage per thread was not maximised, there was a large amount of register spillage into the much slower local memory. When observing Figure 5.1, the high memory demand can be seen clearly. To interpolate the value and the partial derivatives for a single set of reaction coordinates, a large amount of memory is required due to the multiple 3D, 2D and 1D arrays required for each interpolation.

Since a high throughput is required, the interpolation for each reaction coordinate set is mapped to one thread. A block-based assignment does not provide an improvement since the total memory required for the computation of one reaction coordinate is too large for shared memory. Furthermore, the maximum number of threads per block, 1024, is too small to effectively decompose the large number of iterations in the kernels.

Since using more than one block per interaction reduces the throughput and the shared memory cannot be used, a block-based solution is impractical. While the thread per interaction method is highly inefficient, it does produce results.

Comparing execution times amongst platforms

The GPU-based LUT's execution time on an Nvidia Tesla K40 was observed against that of the sequential CPU-based LUT on an Intel® Xeon® E5-2620 to measure computational efficiency:

Number of reaction coordinates	CPU (ms)	GPU (ms)	Speed-up
1000	17285	8678	1.99
5000	85954	9195	9.35
10000	172178	9833	17.5
50000	855070	39216	21.8
100000	1713242	77569	22.1

Table 5.2: Execution time of Algorithm 1 on the CPU and GPU

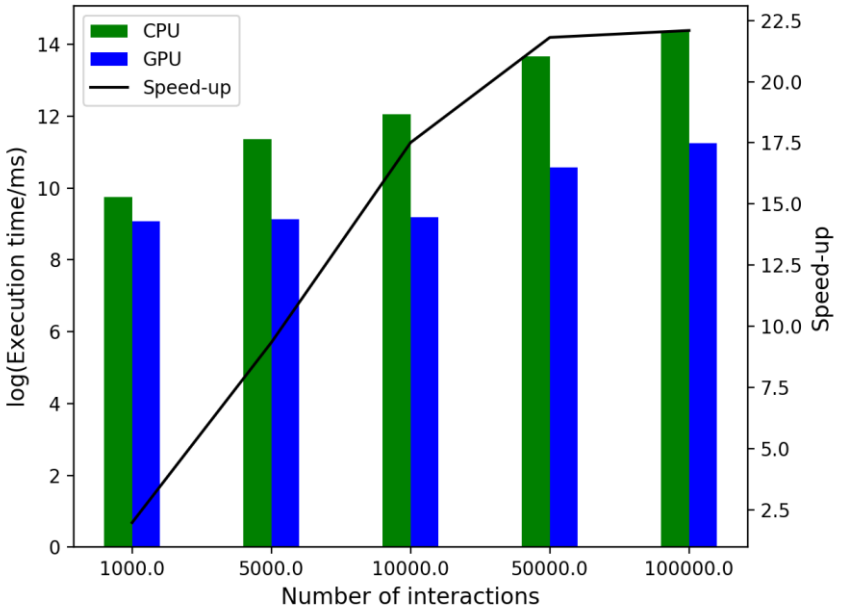


Figure 5.2: Comparing CPU vs GPU execution times on a Logarithmic scale

The improved performance in the GPU algorithm in comparison to the CPU version is noteworthy. Not-withstanding, the GPU's execution time remains too high in comparison to the CGGB module processing a comparable number of computations. For example, the computation of 1000 reaction coordinates with Algorithm 1 takes 8678ms, while the kernels of the CGGB

module take 38.1ms to compute the interactions of 1000 particles, which would be much greater than 1000 computations. Based on the kernel analysis, this result is expected. For a large input size, the program becomes almost sequential since the blocks cannot be computed quickly and efficiently. While the GPU is faster, it is still too slow and inefficient to be used in an MD simulation. The execution time of the GPU LUT must be comparable to that of the GPU-based CGGB module.

Determining the accuracy of the interpolation

The accuracy of the results from the LUT is tested by comparing the values and partial derivatives obtained from the LUT to those computed using the polynomial and its derivatives described in Subsection 5.2.3(ii).

Since the 4D array is quite complex, only a subsection of it is used for testing. For each variable i.e. dimension, a more fine-gridded array of the grid points is created. For example, the grid contains 21 values of variable x from 2.0 to 12.0. The testing array contains 101 values for the same range just with a smaller stride value. The remaining variables are set to the median value of their respective ranges. Therefore, the values will be interpolated and computed for a single variable, while the remaining ones are fixed. This is done for each of the four variables. The Mean Average Percentage Error previously used in Chapter 3 (MAPE) is used as the check for accuracy of the interpolated value, (s) and its derivatives $\left({}^{ds}/_{dx}, {}^{ds}/_{dy}, {}^{ds}/_{dz}, {}^{ds}/_{dw}\right)$.

Selected Variable	**MAPE(%)**				
	$s(\%)$	${}^{ds}/_{dx}\,(\%)$	${}^{ds}/_{dy}\,(\%)$	${}^{ds}/_{dz}\,(\%)$	${}^{ds}/_{dw}\,(\%)$
x	0.0429	0.6163	0.2564	0.0000	0.0000
y	0.0002	0.0005	0.3251	0.3399	1.0447
z	0.0000	0.0001	0.0268	0.0000	0.0000
w	0.0000	0.0001	0.0079	0.0000	0.0000

Table 5.3: Mean average percentage error in the results from Algorithm 1

The accuracy of this LUT algorithm is evidently extremely high; some of the errors are near zero for both the value itself and its partial derivatives.

Conclusions from results

The accuracy of Algorithm 1 is its main advantage. However, it is very inefficient resulting in unfavourable compute times and so is unsuited for a high-throughput implementation on a GPU. For this reason, the algorithm was abandoned to look for a new one that is more suitable for the requirements of this project.

5.3 Algorithm 2: Cubic B-Spline Interpolation

In their article[10], Habermann and Kindermann provided an algorithm for multi-dimensional cubic spline interpolation using B-splines. The 1D cubic B-spline interpolation algorithm was previously discussed in Chapter 4.

This algorithm is well-suited for a high-throughput computational implementation. The time-consuming coefficient computations are done once upfront, making subsequent sets of interpolations low in computational cost as they rely on these precomputed coefficients. Moreover, the algorithm scales well to higher dimensions. These advantageous features make it especially suitable for the GPU-based implementation of the LUT because the coefficients from the free energy array can be pre-computed and stored on the GPU at the start of a simulation. The basis functions used to construct the spline can be processed quickly on-the-fly during the simulation and combined with the pre-computed coefficients to give the interpolated values.

5.3.1 Implementing the 4D interpolation

Following the steps for the general multi-dimensional algorithm given in the work by Habermann and Kindermann[10], the 4D spline interpolation is built. The interpolation is split into two parts, the one-time computation of the coefficients and the interpolation executed for each pairwise interaction.

i. Calculating the 4D coefficients

The equation for the 4D cubic spline interpolation can be described as:

$$s(x,y,z,w) = \sum_{i_x=1}^{n_x+3} \sum_{i_y=1}^{n_y+3} \sum_{i_z=1}^{n_z+3} \sum_{i_w=1}^{n_w+3} c_{i_x i_y i_z i_w} \beta^3{}_{i_x}(x) \beta^3{}_{i_y}(y) \beta^3{}_{i_z}(z) \beta^3{}_{i_w}(w) \qquad (5.10)$$

To obtain the 4D coefficients $c_{i_x i_y i_z i_w}$, the dimensionality of the equation must be reduced iteratively since the tridiagonal algorithm can only be applied to a 1D set of values. For $q_w = 0, \dots, n_w$, Equation 5.10 is reduced to Equation 5.11:

$$s(x,y,z) = \sum_{i_x=1}^{n_x+3} \sum_{i_y=1}^{n_y+3} \sum_{i_z=1}^{n_z+3} c^*_{i_x i_y i_z q_w} \beta^3{}_{i_1}(x) \beta^3{}_{i_2}(y) \beta^3{}_{i_3}(z) \tag{5.11}$$

Equation 5.11 is in 3D, and so must be further reduced to the following equation for $q_z = 0, \dots, n_z$:

$$s(x,y) = \sum_{i_x=1}^{n_x+3} \sum_{i_y=1}^{n_y+3} c^{**}_{i_x i_y q_z q_w} \beta^3{}_{i_1}(x) \beta^3{}_{i_2}(y) \tag{5.12}$$

The 2D Equation 5.12 is reduced to a 1D equation:

$$s(x) = \sum_{i_x=1}^{n_x+3} c^{***}_{i_x q_y q_z q_w} \beta^3{}_{i_1}(x) \tag{5.13}$$

For $q_\mathrm{y} = 0, \dots, n_y$, $c^{***}_{i_x q_y q_z q_w}$ are computed using $f_{i_1 q_2 q_3 q_4}$ and the tridiagonal algorithm. From here, the coefficients $c^{**}_{i_x i_y q_z q_w}$ are computed for $i_x = 0, \dots, n_x + 3$ using $c^{***}_{i_x q_y q_z q_w}$ and the tridiagonal algorithm:

$$s(y) = \sum_{i_y=1}^{n_y+3} c^{**}_{i_x i_y q_z q_w} \beta^3{}_{i_2}(y) \tag{5.14}$$

Once the coefficients $c^{**}_{i_x i_y q_z q_w}$ are computed, $c^*_{i_x i_y i_z q_w}$ can be computed for $i_x = 0, \dots, n_x + 3$ and $i_y = 0, \dots, n_y + 3$ using $c^{**}_{i_x i_y q_z q_w}$ and the tridiagonal algorithm:

$$s(z) = \sum_{i_z=1}^{n_z+3} c^*_{i_x i_y i_z q_w} \beta^3{}_{i_3}(z) \tag{5.15}$$

Finally, $c_{i_x i_y i_z i_w}$ can be computed for $i_x = 0, \dots, n_x + 3$, $i_y = 0, \dots, n_y + 3$ and $i_z = 0, \dots, n_z + 3$ using $c^*_{i_x i_y i_z q_w}$ and the tridiagonal algorithm:

$$s(w) = \sum_{i_w=1}^{n_w+3} c_{i_x i_y i_z i_w} \beta^3{}_{i_4}(w) \tag{5.16}$$

The final coefficients, $c_{i_x i_y i_z i_w}$ are then stored in a 4D array flattened to 1D to be used for the interpolation computations.

ii. Computing the 4D cubic spline

Equation 5.10 translates to a 4D nested loop. Like the 1D case, the equation only produces non-zero results in the intervals:

$$\left[x_{i_x-4}, x_{i_x}\right] \cap [a_x, b_x]$$
$$\left[y_{i_y-4}, y_{i_y}\right] \cap [a_y, b_y]$$
$$\left[z_{i_z-4}, z_{,i_z}\right] \cap [a_z, b_z]$$
$$\left[w_{i_w-4}, w_{i_w}\right] \cap [a_w, b_w]$$

This simplifies the range of Equation 5.10:

$$s(x, y, z, w) = \sum_{i_x=l_x}^{m_x} \sum_{i_y=l_y}^{m_y} \sum_{i_z=l_z}^{m_z} \sum_{i_w=l_w}^{m_w} c_{i_x i_y i_z i_w} \beta^3{}_{i_x}(x) \beta^3{}_{i_y}(y) \beta^3{}_{i_z}(z) \beta^3{}_{i_w}(w) \tag{5.17}$$

Where:

$$l_v = \left\lfloor \frac{v - a_v}{h_v} \right\rfloor + 1 \text{ and } m_v = \min(l_v + 3, n_v + 3) \text{ for } v = x, y, z, w$$

Equation 5.17 also translates to a 4D nested loop ranging from l to m in each dimension. The set of coordinates being queried are all plugged into the basis function and combined with the coefficients in the 4D nested loop to provide the interpolated value.

iii. Computing the partial derivatives

The partial derivative with respect to each reaction coordinate can be expressed in the following ways:

$$\frac{\partial s}{\partial x} = \sum_{i_x=1}^{n_x+3} \sum_{i_y=1}^{n_y+3} \sum_{i_z=1}^{n_z+3} \sum_{i_w=1}^{n_w+3} c_{i_x i_y i_z i_w} \frac{d\beta^3{}_{i_x}}{dx}(x)\beta^3{}_{i_y}(y)\beta^3{}_{i_z}(z)\beta^3{}_{i_w}(w) \tag{5.18}$$

$$\frac{\partial s}{\partial y} = \sum_{i_x=1}^{n_x+3} \sum_{i_y=1}^{n_y+3} \sum_{i_z=1}^{n_z+3} \sum_{i_w=1}^{n_w+3} c_{i_x i_y i_z i_w} u_{i_x}(x) \frac{d\beta^3{}_{i_y}}{dy}(y)\beta^3{}_{i_z}(z)\beta^3{}_{i_w}(w) \tag{5.19}$$

$$\frac{\partial s}{\partial z} = \sum_{i_x=1}^{n_x+3} \sum_{i_y=1}^{n_y+3} \sum_{i_z=1}^{n_z+3} \sum_{i_w=1}^{n_w+3} c_{i_x i_y i_z i_w} \beta^3{}_{i_x}(x)\beta^3{}_{i_y}(y) \frac{du_{i_z}}{dz}(z)\beta^3{}_{i_w}(w) \tag{5.20}$$

$$\frac{\partial s}{\partial w} = \sum_{i_x=1}^{n_x+3} \sum_{i_y=1}^{n_y+3} \sum_{i_z=1}^{n_z+3} \sum_{i_w=1}^{n_w+3} c_{i_x i_y i_z i_w} \beta^3{}_{i_x}(x)\beta^3{}_{i_y}(y)\beta^3{}_{i_z}(z) \frac{d\beta^3{}_{i_w}}{dw}(w) \tag{5.21}$$

To calculate the partial derivatives, the first derivative of the basis function needs to be derived. The basis function is described as:

$$\beta^3{}_i(t) = \begin{cases} (2-|t|)^3 & 1 \le |t| \le 2 \\ 4 - 6|t|^2 + 3|t|^3 & |t| \le 1 \\ 0 & elsewhere \end{cases} \tag{5.22}$$

where:

$$t = \left(\frac{x-a}{h} - (i-2)\right) \text{ for } i = 1, \ \ldots\ldots, n \tag{5.23}$$

By applying the product and chain rules, and the differentiation of absolute functions $\left(\frac{d|t|}{dt} = \frac{|t|}{t} = \text{sgn}(t)\right)$, the derivative of the base can be described as:

$$\frac{d\beta^3{}_i}{dt} = \begin{cases} -3(2-|t|)^2.\text{sgn}(t) & 1 \le |t| \le 2 \\ (-12|t| + 9|t|^2).\text{sgn}(t) & |t| \le 1 \\ 0 & elsewhere \end{cases} \tag{5.24}$$

And for each variable, i.e. for $v = x, y, z, w$:

$$\frac{d\beta^3{}_i}{dv} = \frac{d\beta^3{}_i}{dt}\frac{dt}{dv} = \frac{d\beta^3{}_i}{dt}\frac{1}{h_v} \tag{5.25}$$

The four partial derivatives are calculated using the first derivative basis function and the same coefficients used for the computation of the interpolated value.

5.3.2 Implementation

The two-step algorithm was implemented on the GPU, and on the CPU for a basis of comparison.

i. Performing the interpolation

The computation of the coefficients is complicated and memory intensive. However, it is only performed once at the start of the program. Due to the complex, fine-grained 4D nested loop that uses multiple large arrays in each step, the coefficient computation is not suitable for the GPU. Therefore, the coefficient computation is performed on the CPU for both the CPU and GPU implementations of the algorithm:

```
For q4 from 1 to n4+1:
   - For q3 from 1 to n3+1:
        o For q2 from 1 to n2+1:
             ▪ Compute the coefficients c_str_i1q2q3q4 from
               T4D[i1,q2,q3,q4] using the tridiagonal algorithm
        o For i1 from 1 to n1+3:
             ▪ Compute the coefficients c_str_i1i2q3q4 from
               T4D[i1,i2,q3,q4] using the tridiagonal algorithm
   - For i2 from 1 to n2+3:
        o For i1 from 1 to n1+3:
             ▪ Compute the coefficients c_str_i1i2i3q4 from
               T4D[i1,i2,i3,q4] using the tridiagonal algorithm
For i3 from 1 to n3+3:
   - For i2 from 1 to n2+3:
        o For i1 from 1 to n1+3:
             ▪ Compute the coefficients c_i1i2i3i4 from
               T4D[i1,i2,i3,i4] using the tridiagonal algorithm
```

Listing 5.3: Pseudo-code for computing the coefficients on the CPU

It is important to note that the coefficient array is bigger than the actual 4D surface. The surface has the size $(n_x \times n_y \times n_z \times n_w)$ while the coefficient array, due to the nature of the algorithm, is $(n_x + 3 \times n_y + 3 \times n_z + 3 \times n_w + 3)$. However, since computations in this project are performed in single-precision, this adds only about 48 bytes of data. Since most modern GPUs have large global memory sizes this is not an issue.

Like Algorithm 1, the interpolation is performed for each set of input reaction coordinates in an embarrassingly parallel execution on the GPU and within a single-loop on the CPU. The coefficient array is bound to the GPU's texture memory. The minimum value (a) and stride (h) of each grid-point index is stored in constant memory. The interpolation on the CPU and GPU is performed as follows:

```
For every set of 4D variables (xq,yq,zq,wq) being queried:
   - Compute the values for (l1,l2,l3,l4) and (m1,m2,m3,m4) per reaction
     coordinate
   - For i1 from l1 to m1:
       o Compute the base and derivative base wrt to xq
       o For i2 from l2 to m2:
           ▪ Compute the base and derivative base wrt to yq
           ▪ For i3 from l3 to m3:
               • Compute the base and derivative base wrt to zq
               • For i4 from l4 to m4:
                   o Compute the base and derivative base wrt to
                     wq
                   o Fetch the coefficient for (i1,i2,i3,i4)
                   o Compute the interpolated value and 4 partial
                     derivatives via summation
```

Listing 5.4: Pseudo-code for interpolating using Algorithm 2 on the CPU and GPU

While there is a 4D loop in this kernel, it is limited to a total of 81 iterations at most. Therefore, its execution should not add much to the computation time. Unlike Algorithm 1, there are no large arrays allocated per thread and memory usage and leakage is limited.

ii. Performance analysis

As with Algorithm 1, the kernel is analysed in detail and the performance of the CPU and GPU results are compared to each other for execution time. The interpolation results from the GPU implementation of this algorithm are also compared with the values computed using the 4D polynomial and its derivatives.

Kernel structure analysis

The analysis of the kernel using the Nvidia Profiler provides the information in Table 5.4:

	K1
Threads per block	546
Registers per thread	49
Shared memory per block	19.195 KiB
Theoretical occupancy	56.2%
Actual occupancy	54%
FLOP (% of total execution count)	38%
Inactive (% of total execution count)	35%
Load/Store (% of total execution count)	8%

Table 5.4: Kernel structure analysis of Algorithm 2

The theoretical occupancy is well above 50% and therefore enough to hide the GPU's latency. The number of threads per block was chosen specifically to ensure the highest possible occupancy. The actual occupancy is about 54% which is very close to the theoretical value and over the required value. The kernel did not experience any register spillage into local memory.

Despite the good occupancy, the kernel's efficacy was hampered by execution dependency, memory dependency and divergent threads. Execution efficiency can be addressed by improving the instruction-level parallelism within the kernel. The global memory access can be optimized to improve the memory dependency. However, solving the problem of divergent threads presents a challenge.

The main cause of divergent threads is the branching statements resulting from the piecewise basis functions and the loops to compute the interpolation. This means that multiple threads within a warp differ at that point resulting in reduced efficiency of the warp execution as well as a higher number of inactive threads. Due to the nature of the algorithm, the loops and branching statements are necessary. However, the issues arising from their presence do not extensively impact the overall kernel execution.

Comparing execution times amongst platforms

The performance of the GPU when compared to the CPU version of the algorithm running on a single CPU core is as follows:

Number of reaction coordinates	CPU (ms)	GPU (ms)	Speed-up
1000	30	0.6906	43
5000	170	0.6705	254
10000	340	0.6959	489
50000	1690	2.4145	692
100000	3370	4.7344	712

Table 5.5: Execution time of Algorithm 2 on the CPU and GPU

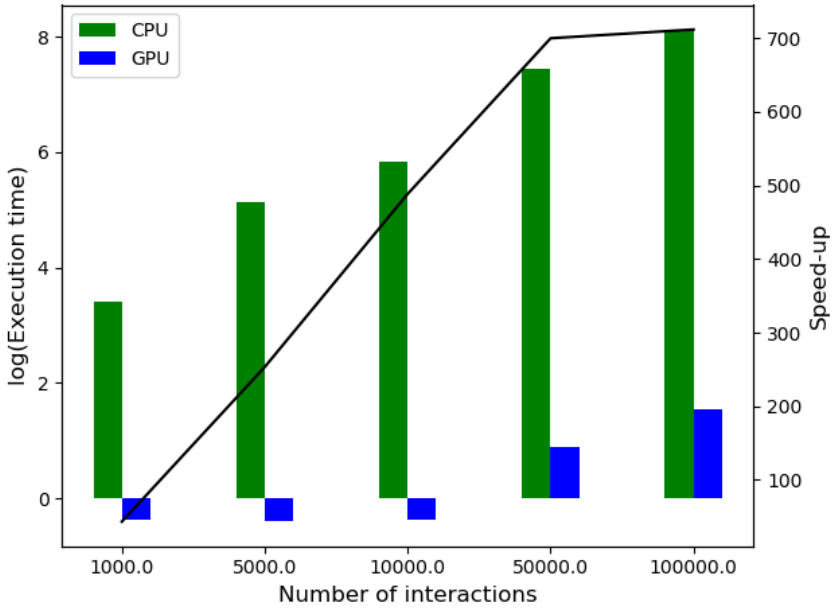


Figure 5.3: Comparing CPU vs GPU execution times on a logarithmic scale

The initial coefficients are computed for both platforms on the CPU which takes approximately 860ms. In terms of execution time, the performance of the GPU-based interpolation is not only superior when compared with its CPU counterpart, but it is a drastic improvement on Algorithm 1. The execution of Algorithm 2 on the GPU for 100,000 reaction coordinates is over 16,000 faster than that of Algorithm 1.

Comparing accuracy amongst platforms

The MAPE values when comparing the interpolated values from the GPU and the on-the-fly calculations from the polynomial are provided in Table 5.6.

Selected Variable	**MAPE(%)**				
	$s(\%)$	$ds/_{dx}\,(\%)$	$ds/_{dy}\,(\%)$	$ds/_{dz}\,(\%)$	$ds/_{dw}\,(\%)$
x	11.6119	14.0301	4.4002	21.1703	0.7961
y	12.8116	14.6948	134.545	22.8821	89.9606
z	38.3415	39.7558	35.4645	11.4965	32723
w	13.3443	14.8241	3.1857	22.5190	0.3111

Table 5.6: Mean average percentage error in the results from Algorithm 2

The accuracy of Algorithm 2 is much poorer when compared to that of Algorithm 1. For the partial derivatives, the average error is very high. This is because of the signum function, $\text{sgn}(t)$. As per the nature of the function, it is undefined at 0 i.e. when $t = 0$, the derivatives do not exist. To avoid divide-by-zero errors and NaN values, the signum function was set to return 0 when $t = 0$. This added to the errors in the estimation.

Conclusions from results

While the overall performance of the algorithm on the GPU is improved, a more accurate algorithm was required that addressed the computational accuracy of the partial derivatives since the main goal of the MP simulation is to compute forces and torques. Due to errors arising from the signum function in the first derivative basis function (Equation 5.24), the accuracy of the partial derivative is compromised. The piecewise basis functions also caused too many divergent threads within a warp and a more efficient algorithm in this respect would be beneficial.

5.4 Algorithm 3: GPU Texture-based Cubic B-Spline Interpolation

This method[11-13] of the cubic B-spline interpolation takes advantage of the GPU's in-built linear interpolation using texture memory and fetch that was discussed in Chapter 4. The primary use for this implementation is to apply an efficient filter on images with the aid of a cubic B-spline interpolation. The authors of this solution argue that the method is not only faster, but more

accurate than other B-spline interpolations. As with the implementations discussed earlier, the process is divided in two steps: obtaining the coefficients and performing the interpolation. The required 4D algorithm is built upon the source code provided for the 1D, 2D and 3D cases.

5.4.1 Interpolation prefilter

Unlike most cubic spline interpolation methods that compute the coefficients using the tridiagonal algorithm, this method computes the coefficient with the aid of a "prefilter"[11]. The prefilter is constructed using the cubic B-spline equation and is designed so that the data points sit perfectly on the splines, which the authors suggest is not necessarily the case for other B-spline based implementations. Since the sample points in the table containing the data points are discrete in nature, it is very simple to create a digital filter for the coefficients from them.

In its discrete version, the cubic B-spline equation can be re-written as:

$$s_3(x) = \sum_{k\prime\epsilon\mathbb{Z}} c[k']b^3_{k\prime}(x - k') \tag{5.26}$$

In this form, the basis functions are treated as integer shifted splines that are centred around the x value being queried. Equation 5.22 can be re-written as the following to include the $\frac{1}{6}$ term found in cubic spline equations:

$$\beta^3(x) = \begin{cases} \frac{1}{6}(2 - |x|)^3 & 1 \le |x| \le 2 \\ \frac{2}{3} - \frac{1}{2}|x|^2.(2 - |x|) & |x| < 1 \\ 0 & elsewhere \end{cases} \tag{5.27}$$

For the system to work, the spline must equal the sample points $f[k]$ at the values of x at the indices:

$$f[k] = s_3(k) = \sum_{k\prime\epsilon\mathbb{Z}} c[k']b^3(k - k') \tag{5.28}$$

The equation above has the form of a discrete convolution. However, the basis function $\beta^3_{k\prime}$ is a continuous signal. In this case however, the equation can be replaced with $b_3[k] = \beta^3(x)$ at $x = k$. So, now Equation 5.28 can be replaced with a convolution:

$$f[k] = s_3(k) = (c * b_3)[k] \tag{5.29}$$

Applying a discreet Fourier transform gives the equation:

$$F(z) = C(z)B_3(z) \xrightarrow{re-arrange} C(z) = F(z){B_3}^{-1}(z) \tag{5.30}$$

$F(z)$ is simply the set of constant points from the table. By converting $b_3[k]$ to Fourier space and re-arranging, the following formula for a digital filter is obtained:

$${B_3}^{-1}(z) = \frac{\lambda}{1 - z_p z^{-1}} \cdot \frac{-z_p}{1 - z_p z} \tag{5.31}$$

In the above, the gain, $\lambda = 6$ and the pole, $z_p = \sqrt{3} - 2$. The filter is implemented as a causal filter and an anti-causal filter linked recursively. The anti-causal filter can be implemented easily since the entire data sequence to be filtered is already known for both images and a LUT with pre-computed values in it.

The discrete data points are passed through the filter to obtain the final coefficient values that will ensure the resulting interpolation makes splines that pass through the data points exactly. Like with the previously discussed cubic spline interpolation algorithms, the filtration is performed once at the start. In the multidimensional cases, the filtration in each dimension is done with the aid of the parallel threads.

i. GPU implementation of the 1D, 2D and 3D prefilters

To understand how the 4D filter is built, it is first necessary to have a brief look at the filters of the previously implemented 1D, 2D and 3D forms. Analysing these gives a general idea of how the multidimensional filtering works that aids with the design of the 4D algorithm.

The 1D case is straight forward. The sample values are first put through the causal filter in the given order. Then, the intermediate values from the causal filter are put through the anti-causal filter in reverse. This is because the anti-causal filter relies on the next sample value, as opposed to the causal filter which needs the previous value. The 1D case simply needs one single threaded kernel to perform the filtration. However, it is an essential building block for higher dimensional cases.

Figure 5.4: A 1D filter processing a 1D array

The 2D case is slightly more complex than the 1D case but follows the same principle. Two filters are used to cover both dimensions of the grid: x and y. The filters are kernels with a thread assigned to each 1D array of values in their respective dimension. First, filterX is applied to each array in the direction of the x-dimension. Then, filterY is applied to the arrays in the y-dimension. After the application of both filters, the final coefficient values for the interpolation are obtained. Figure 5.5 graphically explains the operation of the filters.

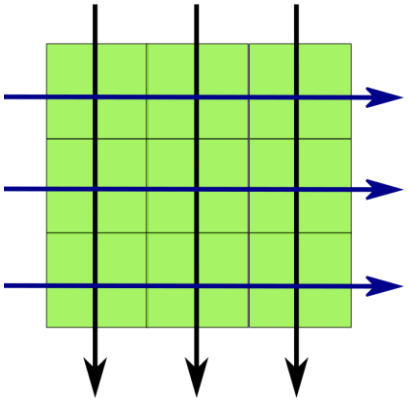


Figure 5.5: A 2D filter processing a 2D array

The 3D case further builds on the 1D and 2D cases. The 3D filter processes the multiple cross-sectional surfaces of the 3D surface in the direction of x, y and z. As with the 2D cases, each dimension is processed by a filter kernel (graphically explained in Figure 5.6): filter for the horizontal slices (black arrows), filter for the vertical slices (red arrows) and filterZ for the width-wise slices (blue arrows).

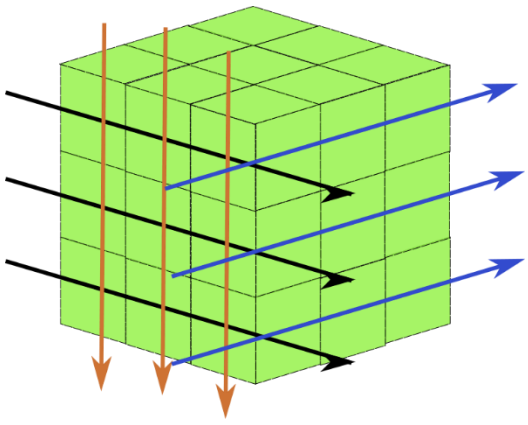


Figure 5.6: A 3D filter processing a 3D array

ii. The 4D prefilter

Using the 1 to 3-dimensional cases described in the previously available source-code[14], the 4D prefilter was built. The 4D filter kernels process the multiple 3D surfaces in the direction of x, y, z and w the 4D surface can be broken down to. As with the 2D and 3D cases, each dimension is processed by its own kernel. The 4D array is stored as a row-major flattened 1D array. Within these kernels, the standard 1D filter is applied to each 1D array the 3D surface can be broken down to.

For each combination of y, z and w-coordinates, filterX directly accesses the first x-coordinate (0) at the given surface. Since the 4D surface is flattened into a 1D array, the location accessed is at: $data + (0 * nw * ny * nz * nw) + (y * nz * nw) + (z * nw) + w$. The selected array of length nx is then filtered. filterY, filterZ and filterW repeat the same technique with the surfaces in their respective dimensions. Since the data at the lower dimensions are not stored contiguously (row-major order), a stepping value built into the filter is used to directly access the value in the 4D surface. Once the filtering process is completed, the final coefficient values for the interpolation are obtained.

5.4.2 Performing the interpolation

Given the coefficients and cubic B-spline basis functions mentioned earlier, the cubic B-spline for any data set can be calculated. The unique feature of this interpolation is the use of in-built linear interpolation of GPUs to compute a cubic spline[15]. The cubic B-spline interpolation equation re-arranged for this purpose is:

$$s(i + \alpha) = g_0 c(i + h_0) + g_1 c(i + h_1) \tag{5.32}$$

The interpolation is divided into the linear interpolation of coefficients and the computation of the weights. In the above equation, x, the independent variable to be interpolated is decomposed into its integer part i, and its fractional part, α.

$$x = i + \alpha;\ i = \lfloor x \rfloor;\ \alpha = x - i \tag{5.33}$$

The fractional part is used for calculating the weights:

$$
\begin{aligned}
w_0(\alpha) &= \beta^3(-\alpha - 1) = \frac{1}{6}(1-\alpha)^3 \\
w_1(\alpha) &= \beta^3(-\alpha) = \frac{2}{3} - \frac{1}{2}\alpha^2(2-\alpha) \\
w_2(\alpha) &= \beta^3(1-\alpha) = \frac{2}{3} - \frac{1}{2}(1-\alpha)^2(1+\alpha) \\
w_3(\alpha) &= \beta^3(2-\alpha) = \frac{1}{6}(\alpha)^3
\end{aligned}
\tag{5.34}
$$

The weights are derived from the B-spline basis function. They eliminate the need for branching statements since the piecewise nature of the basis functions is removed. This is thanks to the decomposition of x to its integer and fractional part. Since only the fractional part is used, the variables will always fall in one of the categories of the piecewise B-spline basis function. For the 1st derivative, the weights are:

$$
\begin{aligned}
\frac{dw_0}{d\alpha}(\alpha) &= \frac{d\beta^3}{d\alpha}(-\alpha - 1) = -\frac{1}{2}\alpha^2 + \alpha - \frac{1}{2} \\
\frac{dw_1}{d\alpha}_1(\alpha) &= \frac{d\beta^3}{d\alpha}(-\alpha) = \frac{3}{2}\alpha^2 - 2\alpha \\
\frac{dw_2}{d\alpha}(\alpha) &= \frac{d\beta^3}{d\alpha}(1-\alpha) = -\frac{3}{2}\alpha^2 + \alpha + \frac{1}{2} \\
\frac{dw_3}{d\alpha}(\alpha) &= \frac{d\beta^3}{d\alpha}(2-\alpha) = \frac{1}{2}\alpha^2
\end{aligned}
\tag{5.35}
$$

To fit into Equation 5.32, the weights are combined for both the actual interpolation and the derivatives:

$$
\begin{aligned}
g_0 &= w_0 + w_1 \\
g_1 &= w_2 + w_3 \\
h_0 &= \left(\frac{w_1}{g_0}\right) - 1 \\
h_1 &= \left(\frac{w_3}{g_1}\right) + 1
\end{aligned}
\tag{5.36}
$$

g_0 and g_1 are combined weights, and h_0 and h_1, summed with the integer part of x i.e. i, are used for obtaining the coefficients. For ease of coding, h_0 and $h_1$are combined with i:

$$h_0 = \left(\frac{w_1}{g_0}\right) - 1 + i$$
$$h_1 = \left(\frac{w_3}{g_1}\right) + 1 + i \tag{5.37}$$

The values of h_0 and h_1 are used to perform the linear interpolation of the coefficients. This is done using the GPU's in-built texture's memory fetch configured to the linear filtering mode.

Since the GPU textures are up till 3D, the algorithm is implemented for 1D, 2D and 3D cubic B-spline interpolation. All cases use parallel threads and increase in dimensionality generally has no effect on the computation complexity. As with the filtration, it was necessary to understand how the interpolation takes place at the lower dimensions to formulate the 4D version. Since a 4D linear interpolation is not available on a GPU, additional code that implements a 4D linear interpolation had to be written. The 4D linear interpolation is inspired by how the GPU performs the linear interpolation in the lower dimensional cases.

i. Adjusting the algorithm

The cubic B-spline algorithm had to be further adapted because the original algorithm was used for image filtering. The 4D linear interpolation also had to be designed to mimic the GPU's texture memory.

Adjusting the indices

The cubic B-spline interpolation algorithm and the GPU's texture memory use the indices or positions of the data points, instead of the actual values (x_i, y_i, z_i, w_i). This is not an issue when querying the actual data points in the LUT. However, when interpolating for a point that lies in-between the data points, it is essential to map the point to a corresponding "index" value:

$$value_{new} = min_{new} + (value_{old} - min_{old}) * \frac{(max_{new} - min_{new})}{(max_{old} - min_{old})} \tag{5.38}$$

The "index" of any point lying in between the grid data will not be a whole number. If the minimum, maximum and total number of data points in the grid are known, the following simplified equation (for languages where the starting index is 0) can be used to get the corresponding "index value, m_v:

$$m_v = (v - v_{min}) * \frac{n_v}{(v_{max} - v_{min})} \tag{5.39}$$

Using m_v, the interpolation can be performed, and the returned value will correspond to the original value. There is, however, another consideration that needs to be made when computing the derivatives. Since the interpolated derivative is actually $\frac{ds}{dm_v}$, and not $\frac{ds}{dv}$, chain rule must be used to compute the required derivative:

$$\frac{ds}{dv} = \frac{ds}{dm_v}\frac{dm_v}{dv} \tag{5.40}$$

$$\frac{ds}{dm_v} = \frac{n_v}{(v_{max} - v_{min})} = \frac{1}{\omega_v} \tag{5.41}$$

In Equation 5.41, ω_v is the stride or spacing of the values of the independent variables in the table. Therefore:

$$\frac{ds}{dv} = \omega_v \frac{ds}{dm_v} \tag{5.42}$$

Using the stride of the variable, the first derivative with respect to that variable can be obtained.

Clamping

A further issue that requires attention is the need for clamping. This arises when the value is close to the boundaries extra checks need to be performed as it is possible that the indices will go beyond the boundaries, specifically to -1, n or $n + 1$. As a result, it is important to clamp the index values when retrieving the coefficients, mimicking the way the texture itself can be clamped at the boundaries. If $i < 0$, then the returned value would be $T[0]$ and if $i \geq n$, then the returned value would be $T[n - 1]$. These conditions are only applicable at the boundaries, in which case the returned values will be the ones expected. In all other cases, the clamping will return the actual index being queried.

ii. Performing the texture fetches

Linear filtering is performed with the values of h_0 and h_1. To replicate the behaviour of CUDA's texture linear interpolation, the documentation was consulted[16]. The linear interpolation is implemented as:

$$tex1D(x_T) = (1 - \alpha_T)T[i_T] + \alpha_T T[i_T + 1] \tag{5.43}$$

Where:

$$x_T = i_T + \alpha_T;\ i_T = \lfloor x_T \rfloor;\ \alpha_T = x_T - i_T \tag{5.44}$$

T is the array bound to the texture memory; in this case the coefficients. In this algorithm, the h_0 and h_1 values are plugged-in for x to provide the interpolated value of the coeffcient. The equation and code get more complex for higher dimensions. In 2D:

$$\begin{aligned} tex2D(x_T, y_T) \\ &= (1 - \alpha_T)(1 - \beta_T)T[i_T, j_T] \\ &+ (1 - \alpha_T)\beta_T T[i_T, j_T + 1] + \alpha_T(1 - \beta_T)T[i_T + 1, j_T] \\ &+ \alpha_T \beta_T T[i_T + 1, j_T + 1] \end{aligned} \tag{5.45}$$

Where:

$$y_T = j_T + \beta_T;\ j_T = \lfloor y_T \rfloor;\ \beta_T = y_T - j_T \tag{5.46}$$

And in 3D:

$$\begin{aligned} tex3D(x_T, y_T, z_T) \\ &= (1 - \alpha_T)(1 - \beta_T)(1 - \gamma_T)T[i_T, j_T, k_T] \\ &+ (1 - \alpha_T)(1 - \beta_T)\gamma_T T[i_T, j_T, k_T + 1] \\ &+ (1 - \alpha_T)\beta_T(1 - \gamma_T)T[i_T, j_T + 1, k_T] \\ &+ (1 - \alpha_T)\beta_T\gamma_T T[i_T, j_T + 1, k_T + 1] \\ &+ \alpha_T(1 - \beta_T)(1 - \gamma_T)T[i_T + 1, j_T, k_T] \\ &+ \alpha_T(1 - \beta_T)\gamma_T T[i_T + 1, j_T, k_T + 1] \\ &+ \alpha_T\beta_T(1 - \gamma_T)T[i_T + 1, j_T + 1, k_T] \\ &+ \alpha_T\beta_T\gamma_T T[i_T + 1, j_T + 1, k_T + 1] \end{aligned} \tag{5.47}$$

Where:

$$z_T = k + \gamma_T;\ k = \lfloor z_T \rfloor;\ \gamma = z_T - k_T \tag{5.48}$$

Finally, the 4D version of the "texture fetch" is derived using the same algorithm:

$$
\begin{aligned}
tex4D(x_T, y_T, z_T, w_T) \\
&= (1-\alpha_T)(1-\beta_T)(1-\gamma_T)(1-\delta_T)T[i_T, j_T, k_T, l_T] \\
&+ (1-\alpha_T)(1-\beta_T)(1-\gamma_T)\delta_T T[i_T, j_T, k_T, l_T+1] \\
&+ (1-\alpha_T)(1-\beta_T)\gamma_T(1-\delta_T)T[i_T, j_T, k_T+1, l_T] \\
&+ (1-\alpha_T)(1-\beta_T)\gamma_T\delta_T T[i_T, j_T, k_T+1, l_T+1] \\
&+ (1-\alpha_T)\beta_T(1-\gamma_T)(1-\delta_T)T[i_T, j_T+1, k_T, l_T] \\
&+ (1-\alpha_T)\beta_T(1-\gamma_T)\delta_T T[i_T, j_T+1, k_T, l_T+1] \\
&+ (1-\alpha_T)\beta_T\gamma_T(1-\delta_T)T[i_T, j_T+1, k_T+1, l_T] \\
&+ (1-\alpha_T)\beta_T\gamma_T\delta_T T[i_T, j_T+1, k_T+1, l_T+1] \\
&+ \alpha_T(1-\beta_T)(1-\gamma_T)(1-\delta_T)T[i_T+1, j_T, k_T, l_T] \\
&+ \alpha_T(1-\beta_T)(1-\gamma_T)\delta_T T[i_T+1, j_T, k_T, l_T+1] \\
&+ \alpha_T(1-\beta_T)\gamma_T(1-\delta_T)T[i_T+1, j_T, k_T+1, l_T] \\
&+ \alpha_T(1-\beta_T)\gamma_T\delta_T T[i_T+1, j_T, k_T+1, l_T+1] \\
&+ \alpha_T\beta_T(1-\gamma_T)(1-\delta_T)T[i_T+1, j_T+1, k_T, l_T] \\
&+ \alpha_T\beta_T(1-\gamma_T)\delta_T T[i_T+1, j_T+1, k_T, l_T+1] \\
&+ \alpha_T\beta_T\gamma_T(1-\delta_T)T[i_T+1, j_T+1, k_T+1, l_T] \\
&+ \alpha_T\beta_T\gamma_T\delta_T T[i_T+1, j_T+1, k_T+1, l_T+1]
\end{aligned}
\tag{5.49}
$$

Where:

$$
w_T = l_T + \delta_T;\ l_T = \lfloor w_T \rfloor;\ \delta_T = w_T - l_T \tag{5.50}
$$

The in-built texture memory shifts the indices being interpolated for by 0.5 before the interpolation. The cubic spline interpolation algorithm itself performed shifts to h_0 and h_1 to compensate for that. This is due to the use of the linear interpolation as well as the cubic spline interpolation in image filtering. However, for this application this shift was removed.

iii. The 4D case

Using the 1D, 2D and 3D cases described above, the 4D case was built. The m value is computed for each index x, y, z and w and used to perform the interpolation, but not directly. m is split into two parts: a fractional and an integer part, just like how h_0 and h_1 are for the linear interpolation. The integer and fractional parts are used to compute the weights and coefficient indices in each dimension.

$(h_0^x, h_0^y, h_0^z, h_0^w)$ and $(h_1^x, h_1^y, h_1^z, h_1^w)$ are computed using Equations 5.34 to 5.37 and used to perform the 4D linear "texture fetch" for the coefficients. The interpolated coefficients and weights are then plugged into the complex formula for the 4D cubic spline interpolation:

$$\begin{aligned} s(x,y,z,w) &= s(i+\alpha, j+\beta, k+\gamma, l+\delta) \\ &= g_0^x g_0^y g_0^z g_0^w c(h_0^x, h_0^y, h_0^z, h_0^w) + g_0^x g_0^y g_0^z g_1^w c(h_0^x, h_0^y, h_0^z, h_1^w) \\ &+ g_0^x g_0^y g_1^z g_0^w c(h_0^x, h_0^y, h_1^z, h_0^w) + g_0^x g_0^y g_1^z g_1^w c(h_0^x, h_0^y, h_1^z, h_1^w) \\ &+ g_0^x g_1^y g_0^z g_0^w c(h_0^x, h_1^y, h_0^z, h_0^w) + g_0^x g_1^y g_0^z g_1^w c(h_0^x, h_1^y, h_0^z, h_1^w) \\ &+ g_0^x g_1^y g_1^z g_0^w c(h_0^x, h_1^y, h_1^z, h_0^w) + g_0^x g_1^y g_1^z g_1^w c(h_0^x, h_1^y, h_1^z, h_1^w) \\ &+ g_1^x g_0^y g_0^z g_0^w c(h_1^x, h_0^y, h_0^z, h_0^w) + g_1^x g_0^y g_0^z g_1^w c(h_1^x, h_0^y, h_0^z, h_1^w) \\ &+ g_1^x g_0^y g_1^z g_0^w c(h_1^x, h_0^y, h_1^z, h_0^w) + g_1^x g_0^y g_1^z g_1^w c(h_1^x, h_0^y, h_1^z, h_1^w) \\ &+ g_1^x g_1^y g_0^z g_0^w c(h_1^x, h_1^y, h_0^z, h_0^w) + g_1^x g_1^y g_0^z g_1^w c(h_1^x, h_1^y, h_0^z, h_1^w) \\ &+ g_1^x g_1^y g_1^z g_0^w c(h_1^x, h_1^y, h_1^z, h_0^w) + g_1^x g_1^y g_1^z g_1^w c(h_1^x, h_1^y, h_1^z, h_1^w) \end{aligned} \tag{5.51}$$

5.4.3 Implementation

Unlike Algorithms 1 and 2, this algorithm is difficult to represent using a few equations. However, this means that there are no loops or branching statements adding to the execution time and stalls in this program. Like Algorithm 2, the program follows a two-step process of computing the coefficients once followed by multiple interpolations. The key difference is that the coefficients can be computed on the GPU. Like the previous two algorithms, each thread on the GPU is mapped to a unique interaction. The CPU version computes each interaction iteratively.

i. Performing the interpolation

As described earlier, the coefficient filtering is done in each direction. This is performed sequentially by 4 functions on the CPU and 4 kernels on the GPU. Within the GPU kernels, the filtration of the slices is independent of each other and so can be performed in an embarrassingly parallel way. On the CPU, it is done in a loop:

```
For tid from 1 to (ny*nz*nw) [in parallel on the GPU]:
   - Split the flattened tid to indices iy, iz and iw in each direction
   - Use indices to directly access the slice of T4D to be filtered.
   - Perform filter on the slice and store
Repeat [in parallel on GPU] for remaining indices.
```

Listing 5.5: Pseudo-code for performing the pre-filter on the 4D array to obtain the interpolation coefficients on the CPU and GPU

The coefficient array is of the same size as the 4D array. Again, the interpolation is performed per set of input reaction coordinates in an embarrassingly parallel execution on the GPU and within a single-loop on the CPU. The coefficient array is bound to the GPU's texture memory for simple fetches and without any linear interpolation. Like Algorithm 2, the grid-point indices (x_i, y_i, z_i, w_i) are not required except for the smallest value, the largest value and the stride per index variable. The interpolation on the CPU and GPU is performed as follows:

```
For every set of 4D variables (xq,yq,zq,wq) being queried:
   - Map the (xq,yq,zq,wq) values to the indices (ixq,iyq,izq,iwq)
   - Compute the interpolated value using the indices
   - Compute the four interpolated partial derivatives using the indices
   - Adjust the partial derivatives to be wrt (xq,yq,zq,wq)
```

Listing 5.6: Pseudo-code for performing the interpolation on the CPU and GPU

ii. Performance analysis

As with the previous algorithms, the five kernels in this algorithm are analysed first. Then, the performance of the GPU and CPU implementations are compared in terms of their execution times. The accuracy of the LUT results is tested by comparing it to the computed values from the polynomial and its partial derivatives.

Kernel structure analysis

The analysis of the kernel using the Nvidia Profiler provides the information in Table 5.7:

	K1	K2	K3	K4	K5
Threads per block	512	512	512	512	1024
Registers per thread	40	40	40	29	63
Shared memory per block	0.0KiB	0.0KiB	0.0KiB	0.0KiB	36.0KiB
Theoretical occupancy	75.0%	75.0%	75.0%	100%	50.0%

Actual occupancy	65.3%	63.2%	64.5%	92.6%	48.9%
FLOP (% of total execution count)	22%	22%	22%	33%	58%
Inactive (% of total execution count)	4%	4%	4%	7%	2%
Load/Store (% of total execution count)	20%	20%	20%	31%	25%

Table 5.7: Kernel structure analysis of Algorithm 3

Kernels 1-4 are the prefilter kernels. To observe their best performance, a large 4D array of size 51^4 was used. The theoretical occupancy of each of these kernels was very high with 75% for kernels 1-3 and 100% for kernel 4. The difference in register usage arises from the fact that kernel 4 accesses contiguous data and therefore uses fewer variables. There was no leakage into local memory. The actual occupancy of the kernels when processing the large 4D array is quite high.

The kernels are not only capable of processing a large amount of data but perform best when they are given more work. While some improvements can be made to the memory access to reduce stalls from memory dependency, the overall computational performance of the prefilter kernels is very good.

For kernel 5, the interpolation routine, the theoretical occupancy is 50% and enough to hide the GPU's latency. The number of threads per block was chosen specifically to ensure the highest possible occupancy. The actual occupancy is a little less at about 49.8% but still close enough. There was no local memory spillage.

Some improvements that can help increase performance is reducing memory dependency and optimizing texture access. Unlike Algorithm 2 which only needs to access the texture memory once per thread, this algorithm needs to access it five times: once for the interpolated value and once each for the four partial derivatives. This results in the texture sub-system getting full more often. This however can be dealt with by optimizing texture memory access, using multiple kernels for each of the five interpolated values and parallelizing them with the aid of streams to reduce memory dependency. There are also no absolute values, loops, branching statements (besides the ones insuring the threads on the GPU do not access illegal memory) in this algorithm and therefore intra-warp divergence is minimal.

Comparing execution times amongst platforms

The performance of the GPU when compared to the CPU version of the algorithm:

Number of reaction coordinates	CPU (ms)	GPU (ms)	Speed-up
1000	140	0.822144	170
5000	680	0.95712	710
10000	1350	1.23962	1089
50000	6380	4.69578	1359
100000	13480	8.17085	1645

Table 5.8: Execution time of Algorithm 3 on the CPU and GPU

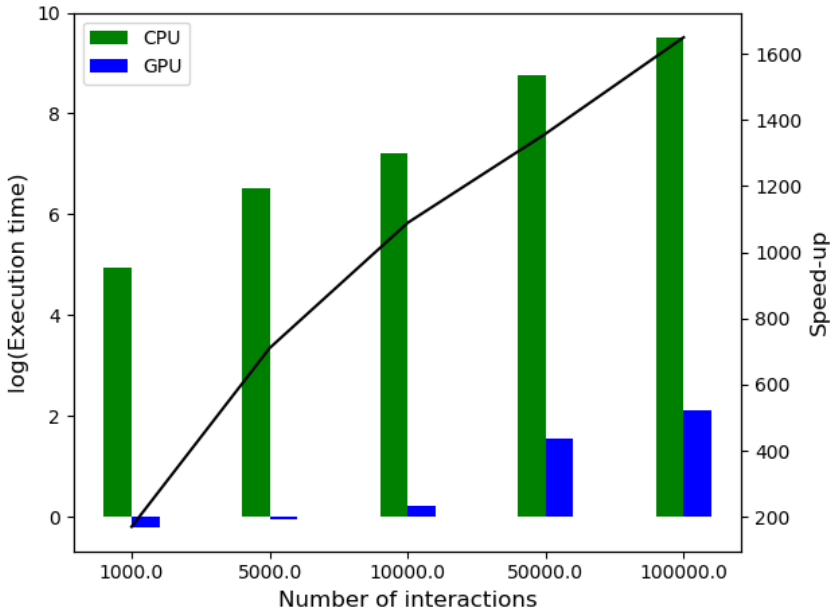


Figure 5.7: Comparing CPU vs GPU execution times on a logarithmic scale

The computation of the coefficients took about 20ms on the CPU, while taking only 0.337152ms on the GPU. Since this algorithm is derived from a GPU-based algorithm, the CPU-based solution is technically an unfair basis for comparison. However, the GPU does show the expected impressive speed-up over the CPU. Compared to Algorithm 2 however, Algorithm 3's performance is weaker. Since, the 4D version requires a custom made 4D linear interpolation, the algorithm uses many variables and registers per thread. While there is no register spillage, the large register usage reduces the occupancy of the SMs on the GPU making the execution a

bit slower. However, since there are no iterations and very few branching statements, the algorithm still does perform very well on the GPU.

Comparing accuracy amongst platforms

The MAPE values between interpolated values from the GPU and the on-the-fly calculations from the polynomial.

Selected Variable	**MAPE(%)**				
	$s(\%)$	$ds/_{dx}\,(\%)$	$ds/_{dy}\,(\%)$	$ds/_{dz}\,(\%)$	$ds/_{dw}\,(\%)$
x	0.0930	1.7400	0.3276	0.0754	0.024
y	0.0027	0.0024	2.4702	0.3959	1.1337
z	0.0000	0.0000	0.0770	0.0383	0.0118
w	0.0000	0.0000	0.0115	0.0246	0.0186

Table 5.9: Mean average percentage error in the results from Algorithm 3

The accuracy of the results is very high and comparable to that from Algorithm 1 and much higher than Algorithm 2. The partial derivatives especially are better because this algorithm does away with the absolute values in and the piece-wise nature of the basis function that lead to errors in the partial derivatives from Algorithm 2.

Conclusions from results

Based on the results, Algorithm 3 provides a good balance of accuracy and efficiency which the previous algorithms were unable to.

5.5 Concluding remarks

Algorithm 1 produced accurate results but was inefficient and unsuitable for a GPU-based implementation. Algorithm 2 was highly efficient and near perfect for the GPU, but its accuracy was too low for MD simulations. Algorithm 3 was much more capable of giving good results in each of these categories. It is very accurate and comparable to Algorithm 1 in this respect while being much more efficient. While it is a bit slower and less efficient than Algorithm 2, it outperforms it in terms of accuracy, especially for the partial derivatives which are essential to the force routine in the MD simulation. As a result, Algorithm 3 was selected for the LUT interpolation algorithm.

5.6 References

1. Gay, J.; Berne, B., Modification of the overlap potential to mimic a linear site–site potential. *The Journal of Chemical Physics* **1981,** *74* (6), 3316-3319.
2. Golubkov, P. A.; Ren, P., Generalized coarse-grained model based on point multipole and Gay-Berne potentials. *The Journal of Chemical Physics* **2006,** *125* (6), 064103.
3. Voth, G. A., *Coarse-graining of condensed phase and biomolecular systems*. CRC press: 2008.
4. Naidoo, K. J., FEARCF a multidimensional free energy method for investigating conformational landscapes and chemical reaction mechanisms. *Science China Chemistry* **2011,** *54* (12), 1962-1973.
5. Gamieldien, M. R.; Strümpfer, J.; Naidoo, K. J., Hydration-Determined Orientational Preferences in Aromatic Association from Benzene Dimer Free Energy Volumes. *The Journal of Physical Chemistry B* **2011,** *116* (1), 324-331.
6. Naidoo, K. J., Multidimensional free energy volumes offer unique insights into reaction mechanisms, molecular conformation and association. *Physical Chemistry Chemical Physics* **2012,** *14* (25), 9026-9036.
7. Gamieldien, M. R. Parameterization of the Gay-Berne Coarse-Grain Potential from atomistically detailed anisotropic free energy volumes. Doctoral Thesis, University of Cape Town, 2012.
8. Press, W. H., *FORTRAN Numerical Recipes: Numerical recipes in FORTRAN 90: the art of parallel scientific computing*. Cambridge University Press: 1996; Vol. 2.
9. Gangopadhyay, A. A GPU-based Look-up table for non-bonded interactions in Molecular Dynamics simulations. Bachelor's Thesis, University of Cape Town, Cape Town, 2015.
10. Habermann, C.; Kindermann, F., Multidimensional spline interpolation: Theory and applications. *Computational Economics* **2007,** *30* (2), 153-169.
11. Ruijters, D.; Thévenaz, P., GPU prefilter for accurate cubic B-spline interpolation. *The Computer Journal* **2012,** *55* (1), 15-20.
12. Thévenaz, P.; Blu, T.; Unser, M., Interpolation revisited [medical images application]. *IEEE Transactions on Medical Imaging* **2000,** *19* (7), 739-758.
13. Ruijters, D.; ter Haar Romeny, B. M.; Suetens, P., Efficient GPU-based texture interpolation using uniform B-splines. *Journal of Graphics Tools* **2008,** *13* (4), 61-69.
14. Ruijters, D. CUDA Cubic B-Spline Interpolation (CI). http://www.dannyruijters.nl/cubicinterpolation/ (accessed June).
15. Sigg, C.; Hadwiger, M., *Fast third-order texture filtering*. 2005; Vol. 2, p 313-329.
16. NVIDIA Corporation CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (accessed 25-06-2018).

6. Coarse-grained simulations of benzene using FEFI and the four-dimensional Look-up table module

6.1 Introduction

Expanding on the literature and the work presented in the previous chapters, the discussion in this chapter explores the feasibility of a LUT module within a coarse-grained MD simulation. More importantly, the task of demonstrating the use of a multidimensional Free Energy Volume (FEV) as a numerical potential in a coarse-grained MD simulation will be evaluated.

The LUT module for the Free Energy Force Induced (FEFI) coarse-grained molecular dynamics package[1] was created and integrated into the main simulation code. The aim of the module is to use a four-dimensional FEV for a pair of interacting benzene molecules as the intermolecular pair potential. Using the LUT, the energy values and their partial derivatives are computed from the FEV for a set of four reaction coordinates. The interpolated values are then used to compute the energy and forces per molecule in the simulation. The optimal four-dimensional cubic B-spline interpolation routines were discussed in Chapters 4 and 5.

To measure the validity of using a multidimensional LUT module in an MD simulation, a Potential Energy Volume (PEV) was constructed using the Gay-Berne (GB) analytical function and the set of Golubkov and Ren[2] parameters for benzene. The FEFI LUT MD simulations' efficiency and computational speed is compared with the performance of the FEFI CGCB MD simulation (Chapter 3). Following this, the LUT modules accuracy when using the GB PEV was established by comparing its results to those from the CGCB module. This is done using a battery of structural and transport property analysis measures.

After verifying the LUT module's effectiveness it was used to simulate a coarse-grained benzene solvent using a free energy numerical intermolecular pair potential generated from FEARCF[3-5]. The results from this simulation are then compared to the results from the fully atomistic CHARMM simulation as well as the FEFI CGGB MD simulation using the Golubkov and Ren parameters.

6.2 Free Energy Surfaces and coarse-grained simulations

The aim of coarse-grained simulations is to provide computational speed-up by removing some degrees of freedom, while still maintaining a considerable amount of atomistic detail. However, as has been observed from literature[6] and within this thesis, ensuring that a CG model is capable of capturing the atomistic information is challenging.

It was previously discussed in Chapter 1 that Voth et al.[6] established that a surface containing the free energies from the fully atomistic interactions of a molecule could be used as a numerical potential in a coarse-grained simulation. This relationship is established in Equation 6.:

$$\exp\left(\frac{-F}{k_B T}\right) = (const.)\int d\boldsymbol{x} exp\left[\frac{-V(\boldsymbol{x})}{k_B T}\right] \approx (const.')\int d\boldsymbol{x}_{CG} exp\left[\frac{-V(\boldsymbol{x}_{CG})}{k_B T}\right] \qquad (6.1)$$

Researchers at the Scientific Computing Research Unit (SCRU) have created four-dimensional, free energy arrays[4] for benzene using the Free Energy from Adaptive Reaction Coordinate Forces (FEARCF) sampling method[3, 5]. These arrays preserve the structural information of two interacting molecules for various distances and individual and relative orientations and can therefore be used to develop a reduced parameter course-grained model that replicates the corresponding atomistic definition of benzene.

After the integration and testing of the LUT module, the surface will be used to perform a CG simulation of liquid benzene using FEFI and the LUT module.

6.3 Integration and testing

While integrating the LUT into FEFI, it was important to keep in mind that the main goal is to use an FEV as a numerical intermolecular potential facilitated by the LUT module. This is a novel approach and consequently the FEV LUT implementation does not have an analytical benchmark by which its performance can be measured. The standalone LUT routine was rigorously tested using a simple 4-variable polynomial in Chapter 5. Here, its performance as a module within FEFI will be tested using the known performance and property measures from CGGB, the analytical potential-based module in FEFI. To achieve this, the GB analytical function

was used to generate a 4D Potential Energy Volume (PEV). The GB PEV was then used as a numerical potential in the LUT module and became the source of the intermolecular energy, forces and molecular torques for each FEFI MD step.

6.3.1 LUT integration

The LUT module operates similarly to the CGGB module. In Chapter 3, the workflow of the GPU-based CGGB module was broken down into three steps and kernels. The first kernel is used to compute the four reaction coordinates from the interaction. The second kernel computes the interaction energy and its four partial derivatives using the GB analytical functions. The third kernel computes the total energy, force and torque per molecule in the system, and the total system energy. The GB PEV was described in terms of the same four reaction coordinates as the GB analytical function. Therefore, Kernel 1 in the CGGB module was incorporated into the LUT module as well to compute the four reaction coordinates to interpolate from the GB PEV.

The force and torque definitions used for the GB analytical function in the CGGB module[2] are also applicable to the 4D energy volume and LUT. Not only do they use the same reaction coordinates, they are both parametrized with cosine values or scalar product values. As per Allen and Germano's discussion[7] on forces and torques in MD simulations, both the GB potential and the LUT are scalar product potentials and will therefore use the same expression for forces and torques. Therefore, the CGGB module's Kernel 3 was also incorporated into the LUT module.

While the CGGB module relied on an analytical function to compute the intermolecular energies and forces, the LUT module is used to interpolate energies and forces from the 4D energy volume. The LUT routine evaluated in Chapter 5 is incorporated as Kernel 2 in the LUT module.

6.3.2 Constructing the GB PEV

The GB potential function and the Golubkov and Ren parameters were used to create a PEV for a set of reaction coordinates. The grid points for each of the reaction coordinates were:

$$r = [2.0: 0.35: 16.0]$$
$$a = [-1.0: 0.1: 1.0]$$
$$b = [-1.0: 0.1: 1.0]$$
$$g = [-1.0: 0.1: 1.0]$$

There are 57 points for r while there are 21 points each for a, b and g. It was necessary to have more grid points for r since the gradient of the surface changes rapidly with respect to it, especially at close contant distances. The ranges of a, b and g are small and the potential's gradient is not heavily reliant on them. So the number of points for those coordiantes is smaller. The grid can be made finer if necessary however this is limited by memory usage on both the CPU (for generating the grid and copying it to the the GPU) and the GPU (size of the RAM).

A 4D volume is, of course, difficult to visualize. However, using Boltzmann averaging, surfaces with reduced dimensionality can be created and observed. The contour plots of the 2D cross-sections of the 4D surface, created by applying the Boltzmann average to the energies, are provided in Figure 6.1. The plots for $r\ vs. cos(\theta_1)$ and $r\ vs. cos(\varphi)$ are contoured at 0.5 kCal/mol and the ones for cos $(\theta_1)\ vs. cos(\theta_2)$ and $cos(\theta_2)\ vs. cos(\varphi)$ are contoured at 0.25 kCal/mol.

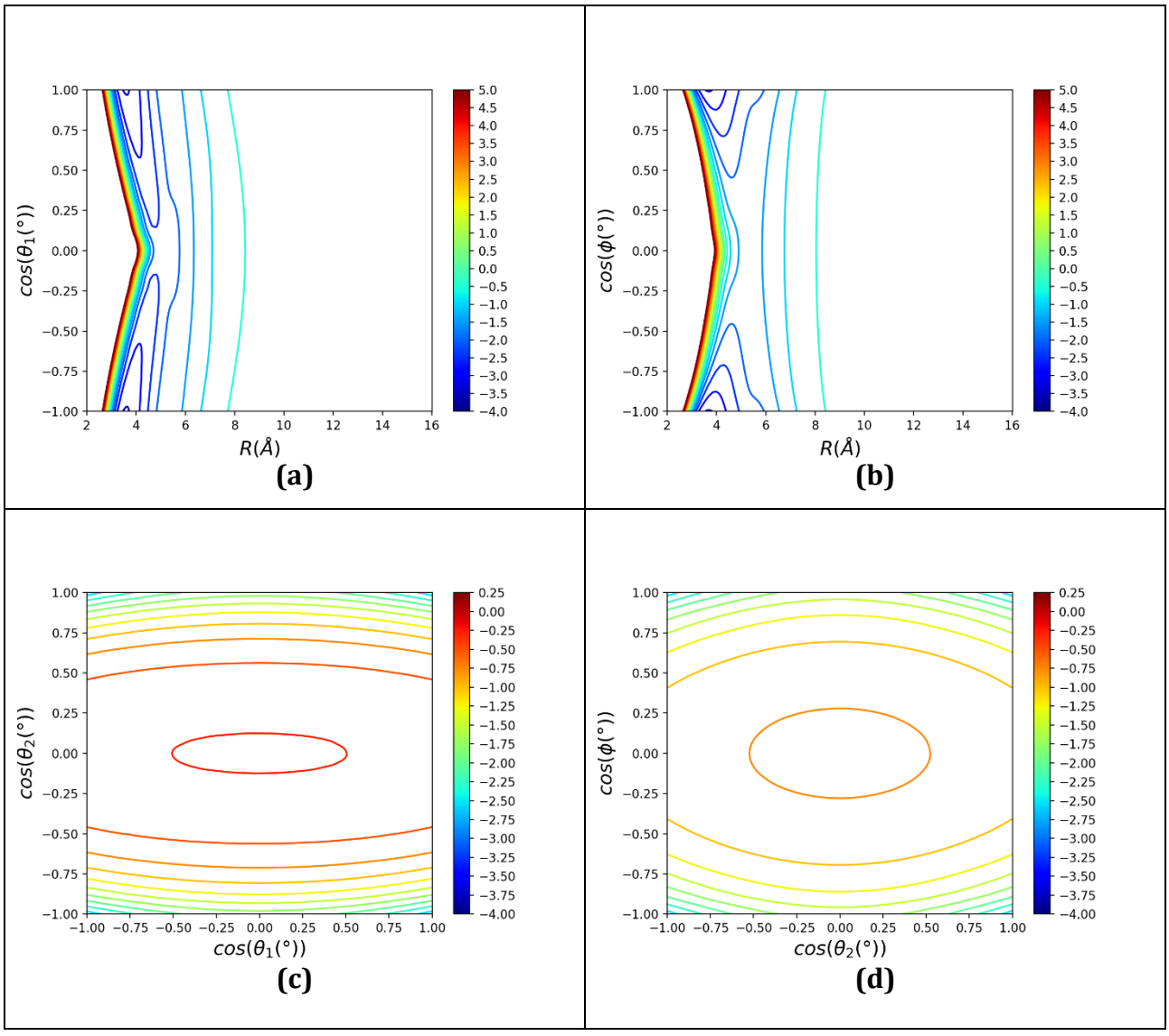


Figure 6.1: The 2D Boltzmann averaged surfaces from the 4D PEV of Benzene.

The plots display the high gradient section at close-contact distances representing the strong repulsive energy. At long distances, the potential asymptotically approaches 0. The angular plots reveal the symmetry of the potential.

6.3.3 Numerical accuracy analysis

Before running the simulation, the LUT was tested for numerical accuracy using the same test used in Chapter 5, Section 5.2.3(ii) with the simpler polynomial. The accuracy of the results from the LUT is tested by comparing the values and partial derivatives obtained from the LUT to those computed using the GB potential function and its derivative functions specified in Chapter 3. As done in Chapter 5, the accuracy is tested against a more gridded range of the selected variable while the rest of the variables are set to the mid-point of their ranges. The MAPEs generated from the accuracy testing are provided in Table 6.1.

Selected Variable	**MAPE (%)**				
	$U(\%)$	$dU/_{dr}\,(\%)$	$dU/_{da}\,(\%)$	$dU/_{db}\,(\%)$	$dU/_{dg}\,(\%)$
r	3.9871	2.7464	386.505	0.5689	1.3479
a	0.4645	17.134	100.445	0.0000	0.0000
b	4.4444	86.467	97.257	960.25	99.751
g	12.1254	6.6772	0.0000	0.0000	16.997

Table 6.1: MAPEs for the interpolated energy value and the partial derivatives

For the GB PEV, the LUT does not produce the level of accuracy observed for the polynomial in Chapter 5. This is expected, since the GB potential has much steeper gradients when compared with the polynomial, resulting in pronounced errors in the partial derivatives.

The accuracy of the partial derivatives is important in FEFI since the forces and torques that determine the dynamics of the simulation are computed from them. As a result, the accuracy of the partial derivatives must be ensured. One way of doing this is by increasing the number of grid points, especially with respect to r since the gradient changes rapidly against it. Ideally, this should be done for all dimensions but that is not feasible in terms of memory usage. It is also much more difficult to develop an FEV using FEARCF[3] with numerous grid points since not only will it take a long time, it will add more errors to the volume due to the nature of sampling.

The main source of the errors is that the partial derivatives can deviate greatly at the end points of the grid. This can be observed in the plot of $\frac{\partial U}{\partial r}$ with respect to r in Figure 6.2. This is due to the nature of the cubic B-spline interpolation that ensures that the first derivative is continuous and differentiable, but not necessarily accurate. This can result in some oscillations at the end points.

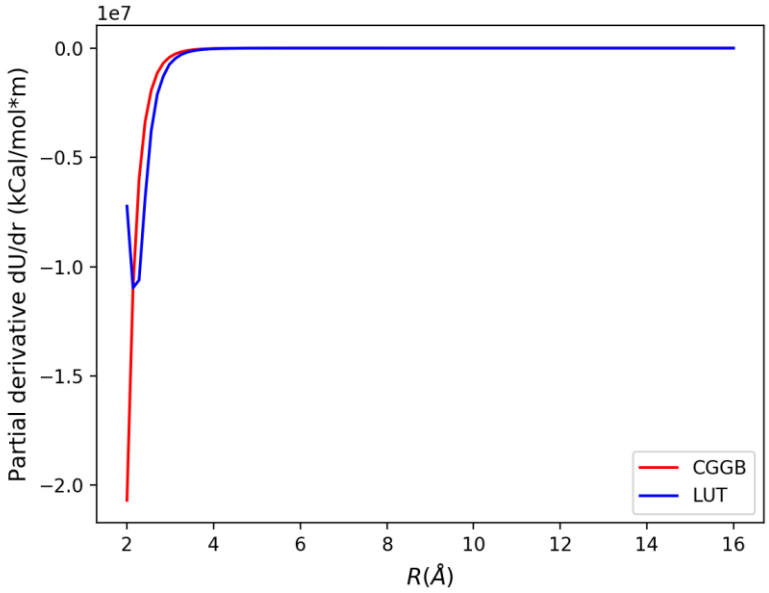


Figure 6.2: The plot of $\frac{\partial U}{\partial r}$ with respect to r

Depending on the magnitude of the derivatives at the end points, the errors may or may not affect the simulation. This can be tested by performing a FEFI CG simulation with the LUT module and GB PEV and comparing its results to a simulation executed using the CGGB module.

6.3.4 Testing the FEFI LUT module's performance with a GB PEV

As with the CGGB module (Chapter 3), only the LUT module was implemented on the GPU. The remaining FEFI simulation was executed on the CPU.

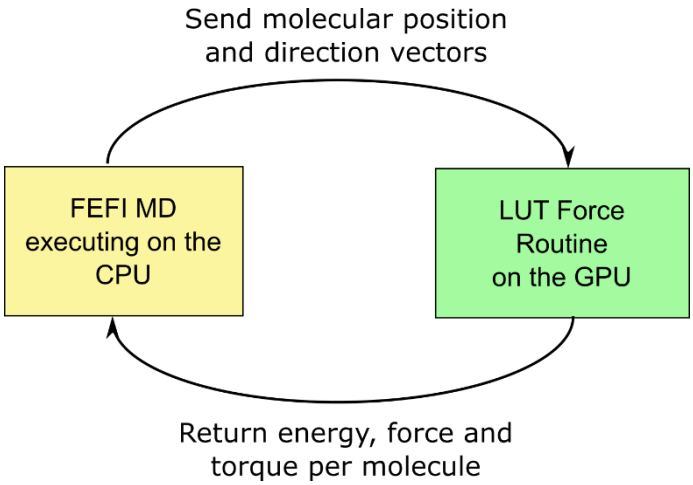


Figure 6.3: FEFI MD executes on the CPU and requests the LUT module on the GPU for energy, forces and torques at every time step

The LUT module executed on an Nvidia Tesla K40, while the remaining simulation ran on an Intel® Xeon® E5-2620 CPU clocked at 2.00GHz. Since the kernels making up the LUT routine were discussed previously in Chapter 3 and 5, further kernel analysis is not required. The accuracy and timing results comparing the two modules will however be further discussed.

i. Comparing accuracy

The Mean Average Percentage Errors (MAPEs) were computed for the energy, force and torque values from the LUT module compared to the results from the CGGB module. These results are provided in Table 6.2. The simulation was run for one time-step for various box sizes. Since the forces and torques are vectors, the percentage error is reported on their magnitude.

No. of molecules	Box side length(Å)	MAPE (%)		
		Energy	Force	Torque
100	24.0	0.2832	52.6329	22.0398
250	34.0	0.1419	73.2304	36.4219
500	42.0	0.1698	41.1192	42.3825
1000	53.0	0.1528	40.6613	25.3250
2500	72.0	0.2761	46.0261	29.7176

Table 6.2: Comparing accuracy between the CPU and GPU

From Table 6.2, it is evident that the 4D interpolation is working well since the interpolated energies have minimal errors. However, the forces and torques have relatively larger errors. This is expected since the forces and torques are reliant on the partial derivatives which are not

always accurate at the grid points. These errors add up per molecule resulting in a force and torque which is erroneous. Despite that, some of the errors, especially ones on the torques, are smaller.

ii. Comparing execution times

Since they are not algorithmically related, an analysis of the execution times of FEFI running the CGCB module and the LUT module does not provide information such as speed-up. Rather it gives an indication of whether the LUT would be feasible when running a long simulation. In Table 6.3, the time taken for the execution of a single force routine of each type is compared. The comparison is explained graphically in Figure 6.4.

No. of molecules	Box side length(Å)	CGGB without latency(ms)	CGGB with latency(ms)	LUT without latency (ms)	LUT with latency (ms)
100	24.0	0.190	13.4	0.51	14.9
250	34.0	0.184	13.8	0.57	14.7
500	42.0	0.207	17.2	0.97	17.8
1000	53.0	0.315	38.1	1.57	37.8
2500	72.0`	0.615	89.0	3.58	123.0

Table 6.3: Execution time per time-step for various simulation sizes

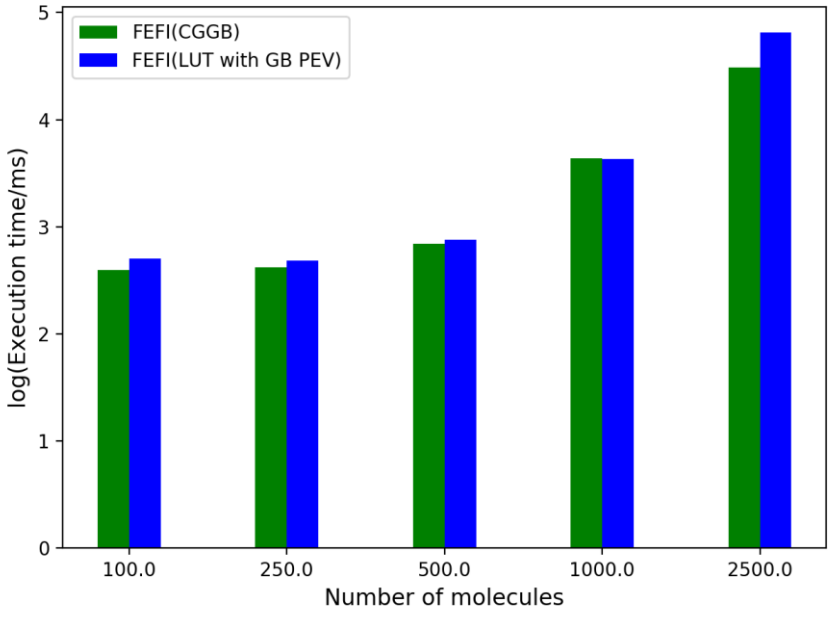


Figure 6.4: Plot of the execution time per time-step for various simulation sizes

For the smaller box sizes, the execution time of the CGCB and the LUT modules are very close. The trend starts to change however for the larger box of 2500 molecules. This is expected since the memory copy and other latencies were no longer hidden due to the interpolation kernel's structure and occupancy (discussed in Chapter 5, Section 5.4).

In Table 6.4, the speed-up achieved in long simulations is analysed. The simulations were conducted for a 42.0Å × 42.0Å × 42.0Å cubical box with 500 benzene molecules at a 1fs time-step while printing out to console at every 10 steps.

No. of steps	Total simulated time(ps)	CGGB(s)	LUT(s)
10000	10	148	152
25000	25	359	382
50000	50	750	782
100000	100	1566	1667

Table 6.4: Execution time and speed-up of CGGB on GPU and CPU

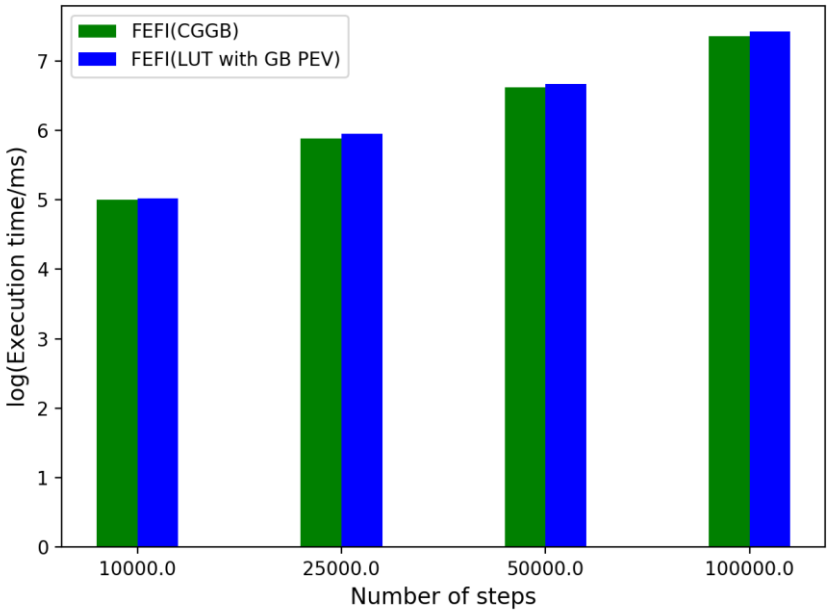


Figure 6.5: Plot of execution times for different simulation lengths

From the Table 6.4 and Figure 6.5, it is visible that the overall performance of FEFI using each of the two modules is relatively similar for this box size. This shows that at least for smaller boxes, the LUT module has a sustainable execution time, making it suitable as a numerical potential in a long coarse-grained simulation.

Following the evaluation of the execution times and numerical accuracy, in the next section, the efficiency of the LUT in its ability to mimic the analytical function is observed.

6.4 Simulation results using the GB PEV

The analysis of the simulation results applies the tests used for evaluating the CGGB module in Chapter 3. However, the aim is to prove how well the LUT module running the GB PEV reproduces the CGGB module's simulation performance. Consequently, a comparison with the fully-atomistic liquid benzene CHARMM simulation is not a part of this analysis. All simulations were run for 100,000 steps with timesteps at $1fs$ in length.

6.4.1 Probability Distribution Functions

In this section, the ability of the LUT module to reproduce the configurational structures of liquid benzene as modelled by the CGGB module is evaluated. Both radially averaged (RDFs) as well as orientational distribution functions are used to make this evaluation.

i. Radial Distributions

The plots in Figure 6.6 and Figure 6.7 were obtained from the position trajectories of the molecules in the system and CHARMM's provided RDF calculator[8].

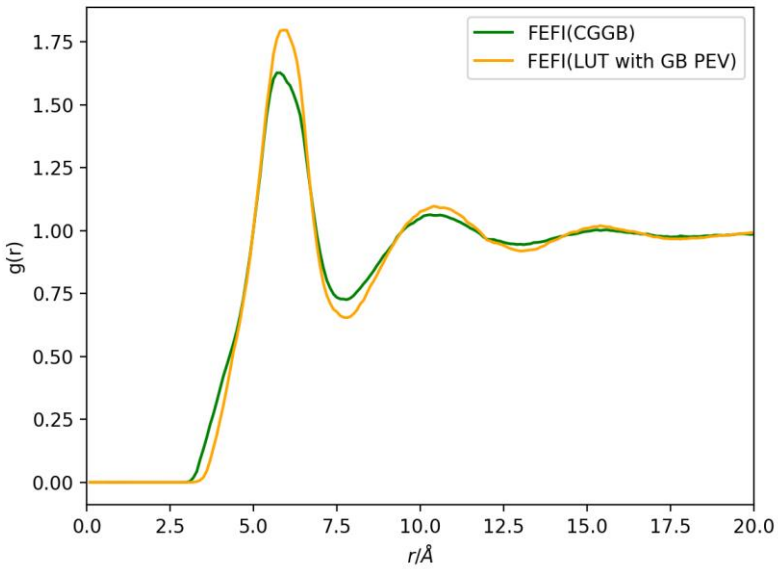


Figure 6.6: The Radial Distribution of Benzene centre-of-mass to centre-of-mass

The center-of-mass to center-of-mass RDF from the LUT mimics the one from CGGB in terms of position; the peaks are perfectly in-line. However, the LUT RDF has slightly higher peaks, most likely due to the incorrect larger forces at close contact distances.

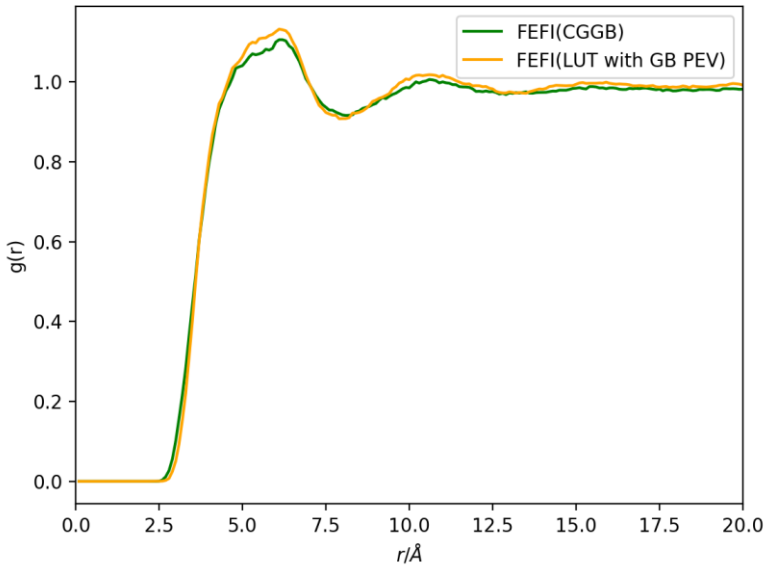
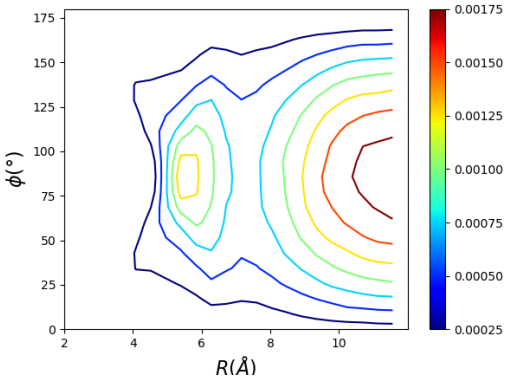
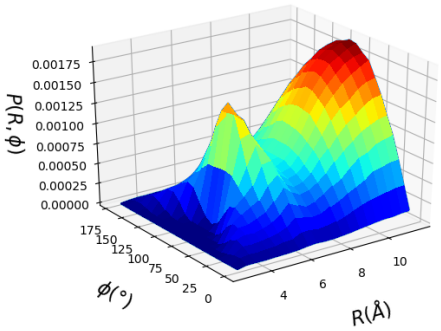


Figure 6.7: The Radial Distribution of Benzene carbon to carbon

The carbon to carbon RDFs are nearly identical. As with the CoM-CoM RDFs, the peaks are in-line but the one from the LUT is slightly higher.

ii. Radial-Angular Distributions

The 2D contour and 3D surface plots of $r\ vs.\varphi$ in Figure 6.8 were obtained from the displacement trajectories of the simulations.

(a)		

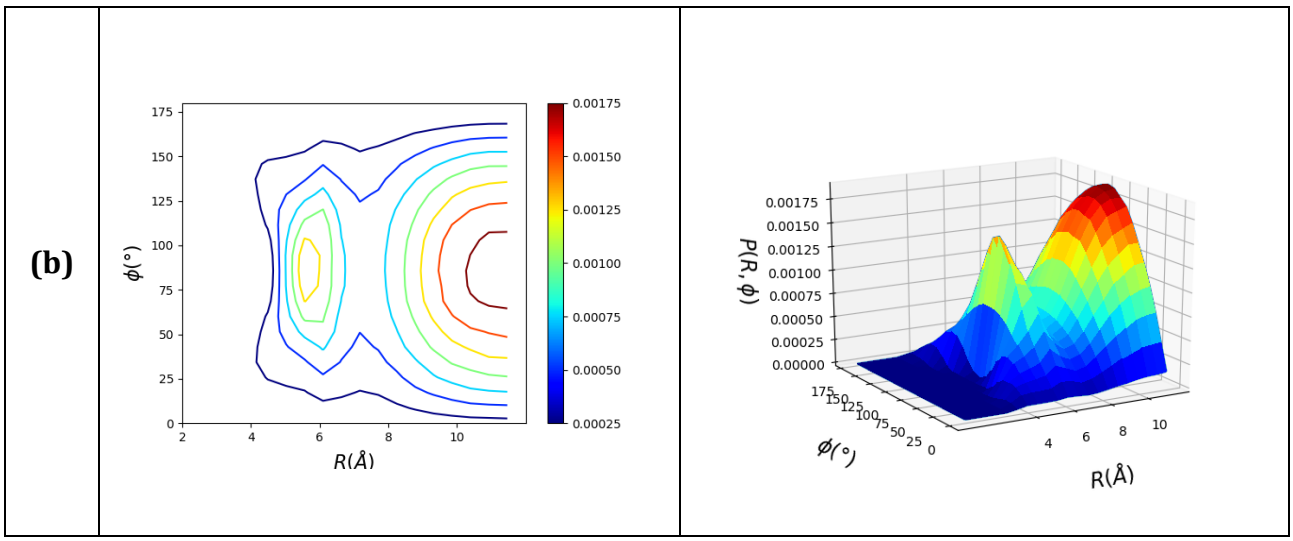


Figure 6.8: 2D contour and 3D surface plots of $r\ vs.\ \varphi$ a) for the CGGB module with G&R parameters and b) the LUT module with the GB PEV

The two orientational distributions are nearly identical, however like the RDFs, the peaks in the plots from the LUT are slightly higher. The distribution from the LUT is also slightly cleaner in comparison to the one from CGGB. This is because the interpolation produces smother values than the analytical function.

6.4.2 Diffusion coefficient

The distribution for the LUT simulation was computed to compare with the one from the CGGB module:

	Diffusion coefficient (10^{-9} m²/s)
FEFI(CGGB)	3.85
FEFI(LUT with GB PEV)	3.70

Table 6.5: Comparing the diffusion coefficients from the CGGB module with G&R parameters and the LUT module with the GB PEV

As with the results, the two diffusions are very similar but not the same. The differences could be attributed again to the slightly higher forces computed from the erroneous partial derivatives obtained from the LUT at close contact distances.

6.4.3 Computational performance

Estimates for execution times of FEFI for a $1ns$ simulation at $1fs$ time-steps, using the CGGB module and the LUT module are provided in Table 6.6.

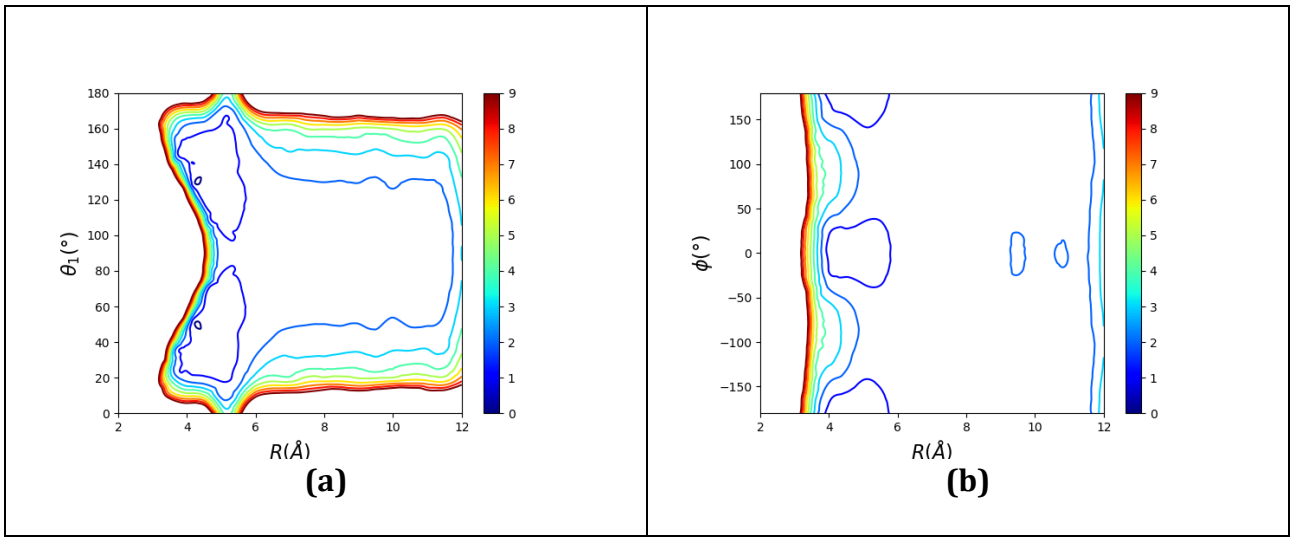
	Execution time (hrs)
FEFI(CGGB)	4.35
FEFI(LUT with GB PEV)	4.63

Table 6.6: Estimated long timescale execution times from the CGGB module and the LUT module

The LUT, despite being more complex, adds very little to the time. However, as is evident from the step-times listed in Table 6.6, this can progressively get worse for larger boxes. This is due to the complex nature of the spline interpolation kernel which uses much more resources than the kernel for the analytical function.

6.5 Using the Free Energy Volume for the LUT

The primary aim of creating the LUT module was to use a free energy surface as the intermolecular potential. In their work, Gamieldien et al.[1] used FEARCF to produce 4D free energy volumes for water and benzene. The Boltzmann averaged plots in in Figure 6.9, are all contoured at 1.0 kCal/mol.



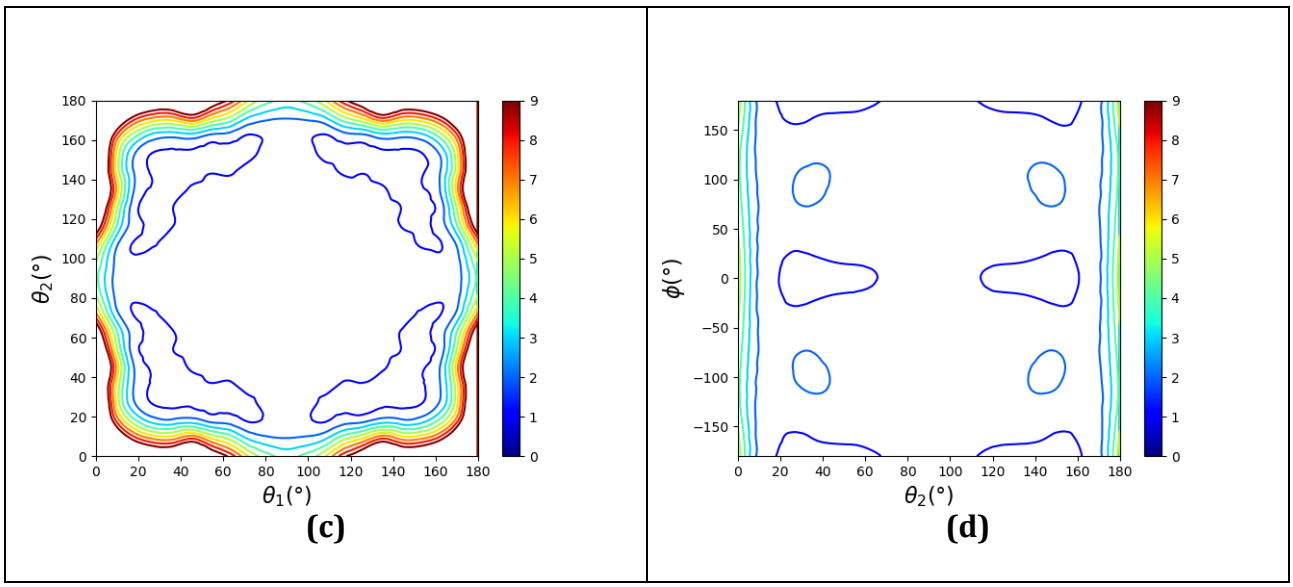


Figure 6.9: The 2D Boltzmann-averaged surfaces from the 4D FEV of Benzene.
(a) $r\ vs.\theta_1$, (b) $r\ vs.\varphi$, (c) $\theta_1\ vs.\theta_2$, (d) $\theta_2\ vs.\varphi$

The free energy volumes can be described with:

$$U_{FE} = W(r, \theta_1, \theta_2, \varphi_{dihe}) \tag{6.2}$$

The four reaction coordinates describing this interaction are:

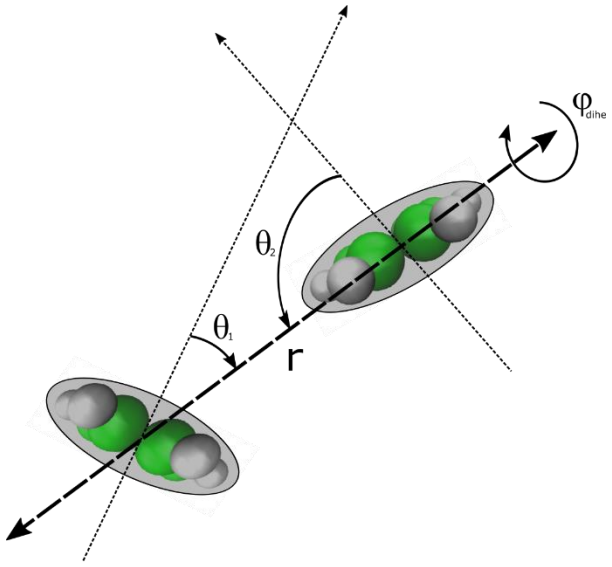


Figure 6.10: Two benzene molecules and the reaction coordinates describing their interaction[4]

- r: The distance between the center of masses of the two molecules.
- θ_1: The molecular vector angle (orientation) of the first molecule.
- θ_2: The molecular vector angle (orientation) of the second molecule.

- φ_{dihe}: The dihedral angle providing the relative orientation and rotation of the molecules.

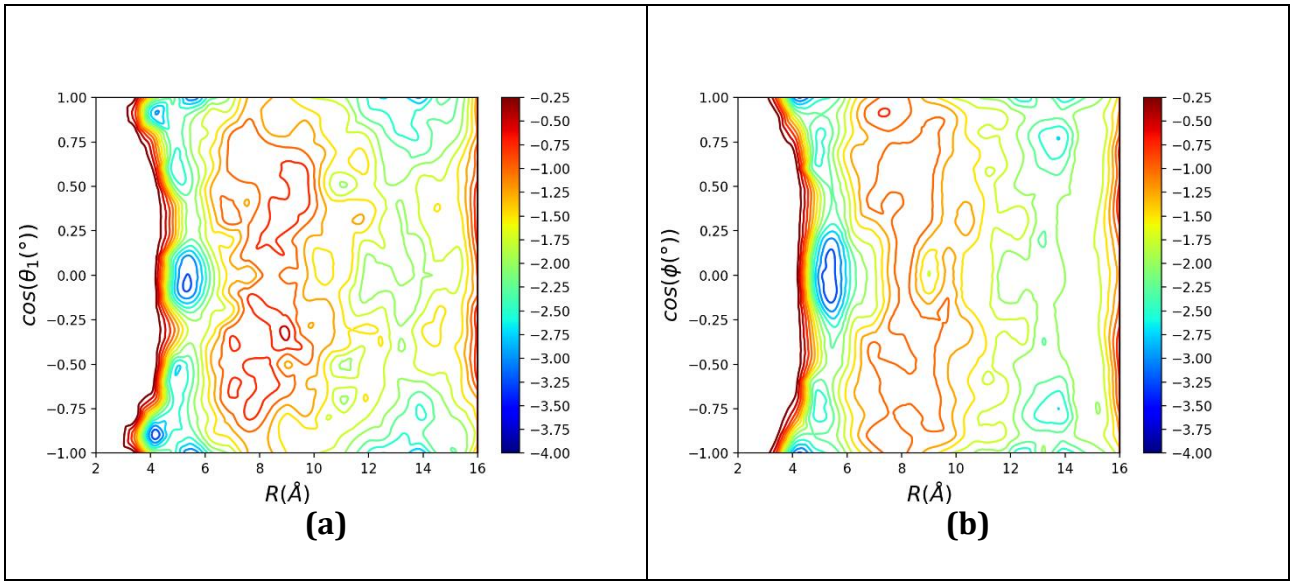
It is important to note that while the first three reaction coordinates are present in the GB potential, the dihedral angle φ_{dihe} does not correspond to the GB definition of φ. As a result, the volume described above cannot be used to parametrize the GB potential. Therefore, a new volume that corresponds to the GB reaction coordinates was created.

6.5.1 A new surface and issues

The new volume can be described with:

$$U_{FE} = W(r, \cos\theta_1, \cos\ \theta_2, \cos\varphi) \tag{6.3}$$

The angle φ was changed to the GB version described in Chapter 3, Section 3.2.3(ii)) and the surface was also described in terms of the angles' cosine values. This is like the GB PEV described in Figure 6.1. The plots in Figure 6.11 are contoured at 0.25 kCal/mol.



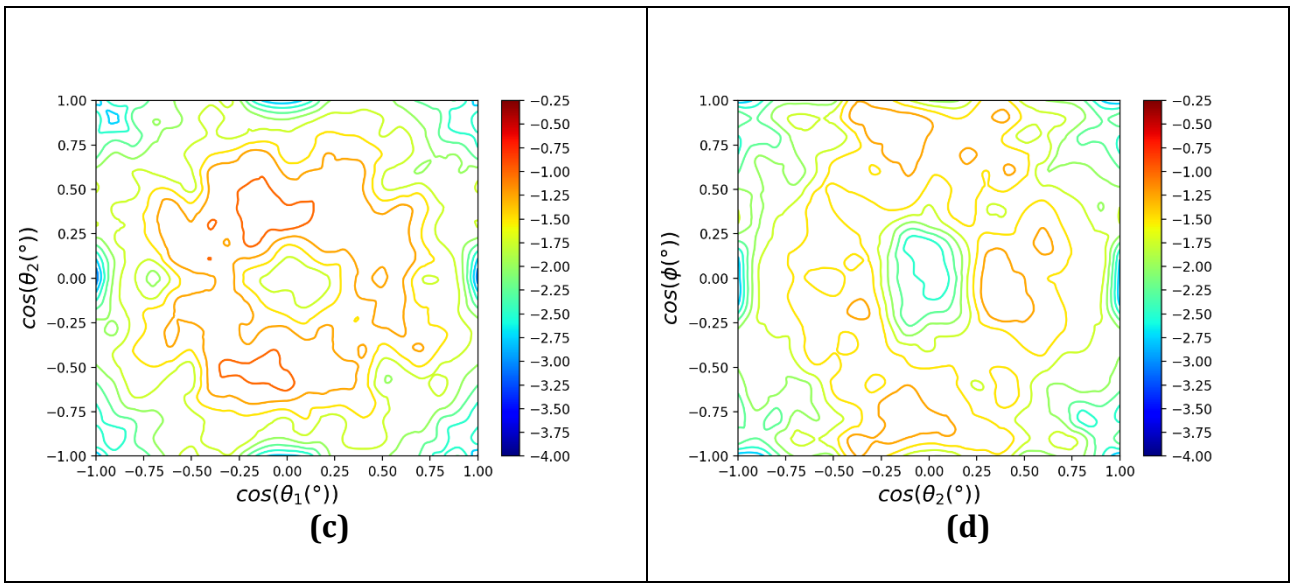


Figure 6.11: The 2D Boltzmann-averaged surfaces from the updated 4D FEV of Benzene. (a) $r\ vs.\theta_1$, (b) $r\ vs.\varphi$, (c) $\theta_1\ vs.\theta_2$, (d) $\theta_2\ vs.\varphi$

The surface was used in FEFI via the LUT module. However, the results were not as expected. The simulation resulted in all the particles clustering together. The cause, however, is visible from the plots of the surface in Figure 6.11. The highest value in the plot is in fact about 0, and the close contact distances display no change in value. This that meant there is no force in the region. Consequently, any molecules in close contact did not repel each other resulting in clustering. This is due to a sampling error in FEARCF that is unable to sample the high-gradient close contact energy properly. At long distances, the surface becomes noisy and unreliable as well.

6.5.2 The soft-core potential and switching function

A way to tackle the clustering problem is to introduce a surface with a high gradient at the short contact distances. A good candidate for this is the soft-core potential[9].

$$U_{SC}(r,a,b,g) = m[r - \sigma(r,a,b,g)] \tag{6.4}$$

The potential has been used by Berrardi et al. in their work[9] to reduce the sharp recoil arising from the close-contact forces in the GB potential. Equation 6.4 shows that the potential uses the anisotropic σ term from the GB potential to tailor it for the various orientations. The parameter m is the gradient of the function and the source of the close contact forces. Given the nature of intermolecular potentials, m must be negative.

To merge the soft-core potential with the GB potential surface, Berrardi et al. used a switching function based on an exponential:

$$f(r,a,b,g) = \frac{exp\{k[r-\sigma(r,a,b,g)]\}}{1+exp\{k[r-\sigma(r,a,b,g)]\}} \tag{6.5}$$

Like the soft-core potential, the switching function also relies on σ . The parameter k is used to tune the function to obtain the best possible switch between the two surfaces. If $k < 0$, the function tends to unity $r < \sigma$. For $r > \sigma$, it tends to 0 asymptotically. This can be used to blend the surfaces in a way that, depending on the distance parameter, r, one potential is predominantly available over the other. The FEV and the soft-core potential can be combined using[9]:

$$U_{AFEV}(r,a,b,g) = (1-f)U_{FEV} + fU_{SC} \tag{6.6}$$

6.5.3 Parameterization

To use the soft-core potential and switching function, the σ parameter must be computed. The GB description of the term relies on two parameters:

- l: The length of the molecule/CG ellipsoid. In benzene's case, the diameter.
- d: The width or thickness of the molecule/CG ellipsoid.

Furthermore, since FEFI runs in reduced units a value for ε_0: the well-depth of the potential when the molecules are in cross-configuration, is also required. These parameters can be obtained by analysing the surface and extracting the plots of two specific configurations. Given the noisy nature of the surface, a few adjustments to the plot had to be made to focus on the sections required to obtain the parameters.

i. The cross-configuration

In this position, the molecules form a cross shape in a way that their molecular vectors are perpendicular to each other, They are also perpendicular to the distance vector running between them. The three angles for this interaction are therefore, $\theta_1 = 90^0, \theta_2 = 90^0$, and $\varphi = 90^0$.

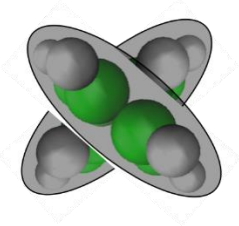


Figure 6.12: The cross-configuration

The plot in Figure 6.3 was extracted from the FEV for this position:

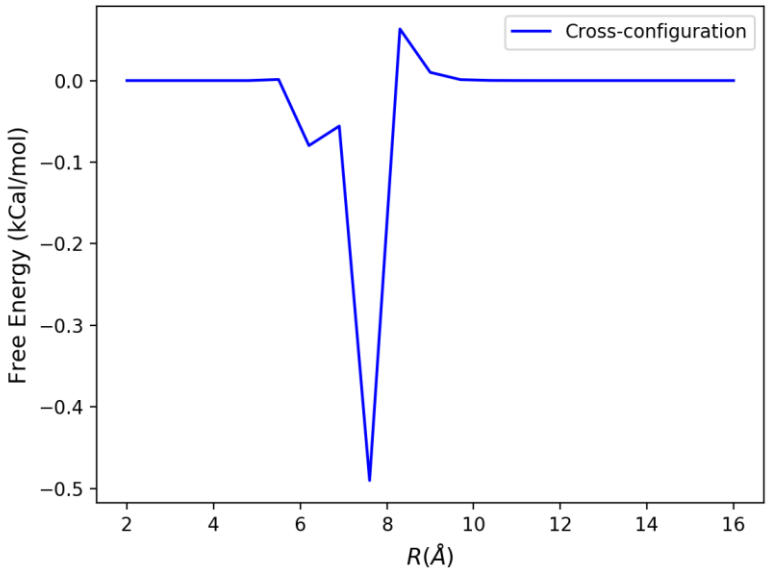


Figure 6.13: The plot for the cross-configuration energy extracted from the FEV

In this position, $\varepsilon(\hat{\boldsymbol{u}}_i, \hat{\boldsymbol{u}}_j, \vec{\boldsymbol{r}}_{ij}) = \varepsilon_0$ and $\sigma(\hat{\boldsymbol{u}}_i, \hat{\boldsymbol{u}}_j, \vec{\boldsymbol{r}}_{ij}) = \sigma_0$. From Figure 6.13, the minimum energy is approximately -0.49 Therefore, $\varepsilon_0 = -0.49\ kCal/mol$. Similarly, the close contact distance from the plot is about 5.75Å. Using $\sigma(\hat{\boldsymbol{u}}_i, \hat{\boldsymbol{u}}_j, \vec{\boldsymbol{r}}_{ij})$, this gives $\sigma_0 = 5.75$Å and $d = 4.07$Å.

ii. The end-to-end/face-to-face configuration

In this position, the "wider" faces of each benzene molecules face each other i.e. they are parallel:

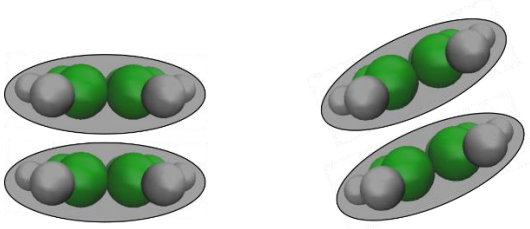


Figure 6.14: The end-to-end/face-to-face configuration

Since the molecular vector is perpendicular to the plane of the molecule in benzene, the molecular vectors are not only parallel to each other, but also to the distance vector between them. The three angles for this interaction are therefore, $\theta_1 = 0^0, \theta_2 = 0^0$, and $\varphi = 0^0$. Using the cosine values of the angles, the plot in Figure 6.15 was obtained from the FEV:

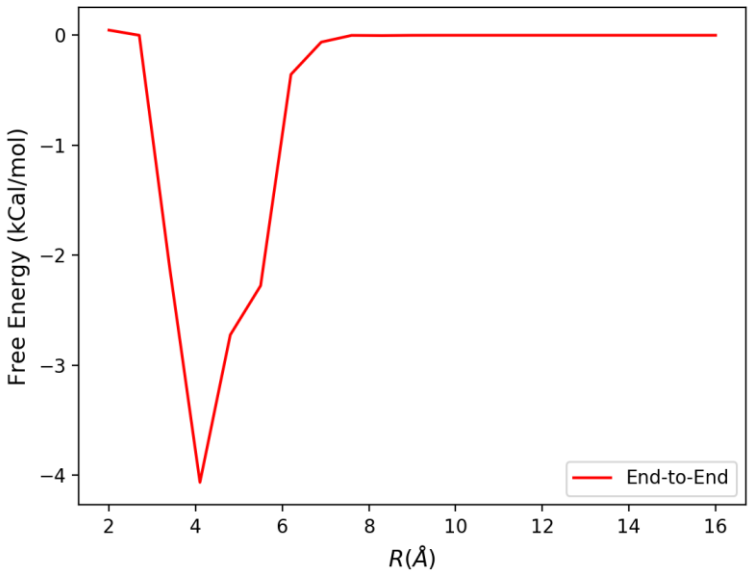


Figure 6.15: The end-to-end/face-to-face section of the FEV

The plot is used to determine the value for the length parameter l by obtaining the close contact distance which is approximately 2.75Å. By plugging in the value and cosines of the angles into the equation for $\sigma(\hat{\boldsymbol{u}}_i, \hat{\boldsymbol{u}}_j, \vec{\boldsymbol{r}}_{ij})$, l is determined to be about 2.75Å as well.

iii. Parameters for the soft-core potential and the switching function

To obtain a high enough gradient, the maximum value of the end-to-end configuration was set to be about $15kCal/mol$ at $r = 2.0$Å. Based on that, m was computed to be about $-1834.5\frac{\varepsilon_0}{\sigma_0}$.

Similarly, the value of k had to be set to ensure that the minima in the surfaces were not removed. For the cross-configuration, the value of the switching function must be very close 0 when the energy is ε_0. The best fit was obtained using $k = \frac{-812.0}{\sigma_0}$.

iv. The adjusted surface

The parameters computed to adjust and combine the FEV with the soft-core potential are listed in Table 6.7. Using the switching function, the soft-core potential was merged with the FEV entirely. For $r < \sigma$ values, the soft-core potential was predominant. For $r > \sigma$ however the values were obtained almost entirely from the FEV.

l	d	ε_0	m	k
2.75	4.07	0.73	$-1834.5\frac{\varepsilon_0}{\sigma_0}$	$-812.0\frac{1}{\sigma_0}$

Table 6.7: Parameter set to combine the FEV with the soft-core potential

Since the FEV contained quite a bit of noise at larger distances as well, with the aid of the switching function all these values were set to 0. The value of k was set to -100 to ensure a smooth but rapid switch to 0. The Boltzmann reduced plots of the final altered surface are provided in Figure 6.16. The plots for $r\ vs.cos(\theta_1)$ and $r\ vs.cos(\varphi)$ are contoured at 0.5 kCal/mol and the ones for cos $(\theta_1)\ vs.cos(\theta_2)$ and $cos(\theta_2)\ vs.cos(\varphi)$ are contoured at 0.25 kCal/mol.

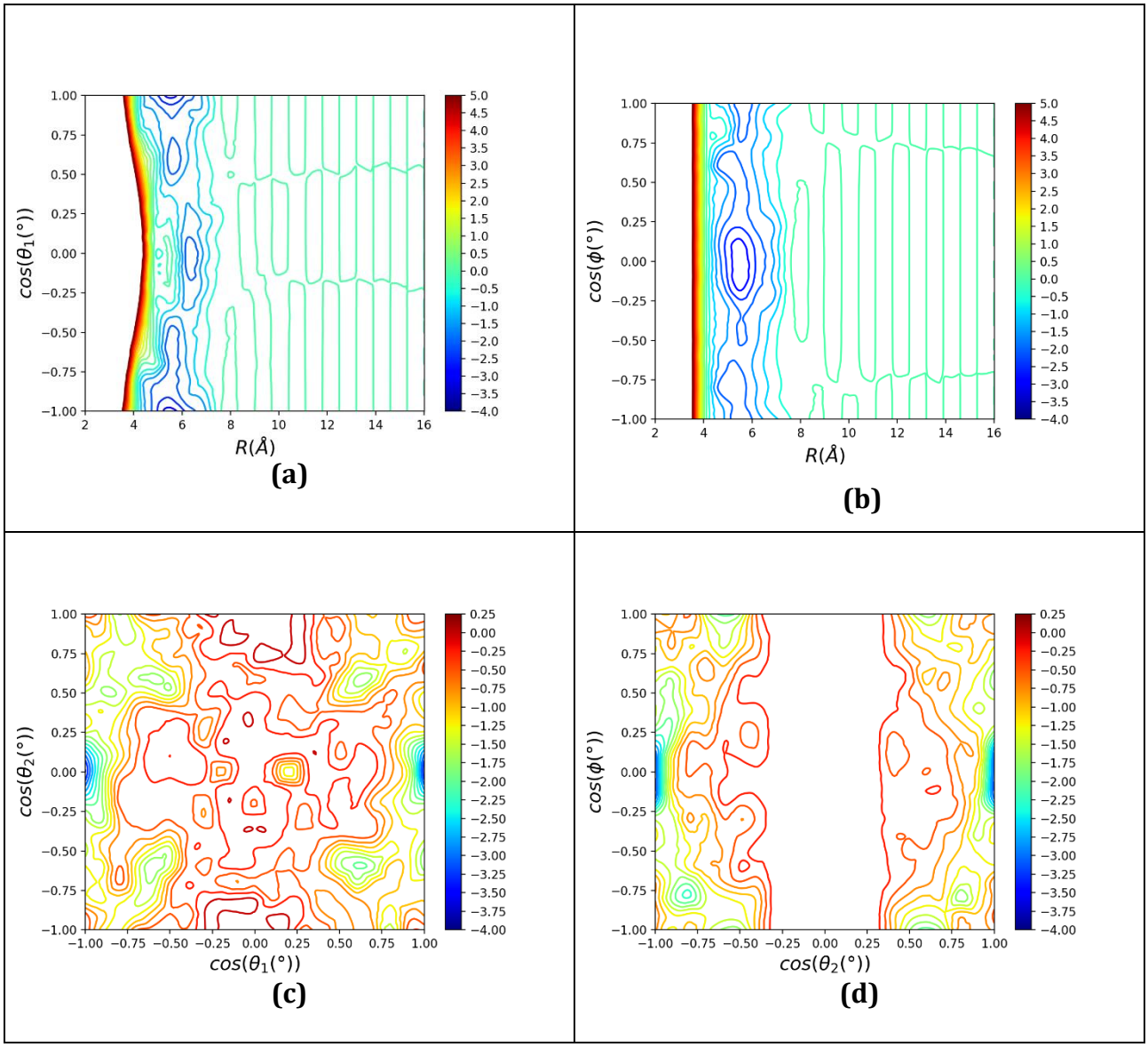


Figure 6.16: The 2D Boltzmann-averaged surfaces from the new FEV altered using the soft-core potential. (a) $r\ vs.cos(\theta_1)$ (b) $r\ vs.cos(\varphi)$ (c)cos $(\theta_1)\ vs.cos(\theta_2)$ (d) $cos(\theta_2)\ vs.cos(\varphi)$

6.6 Simulation results using the altered FEV

The altered 4D FEV was used in a simulation to determine how well it performs. The surface is compared to the fully atomistic simulation results from CHARMM and the results of the GB analytical function using the Golubkov and Ren[2] parameters. In analysing the accuracy of the following results, it is important to remember the slight errors present in the results arising from some incorrect partial derivative values.

6.6.1 Probability Distribution Functions

Both radially averaged (RDFs) as well as orientational distribution functions are used to evaluate the ability of the LUT module in approximating the configurational structures of liquid benzene as modelled by the fully atomistic CHARMM simulation.

i. Radial Distributions

The plots in Figure 6.17 and Figure 6.18 were obtained from the position trajectories of the molecules in the system and CHARMM's provided RDF calculator.

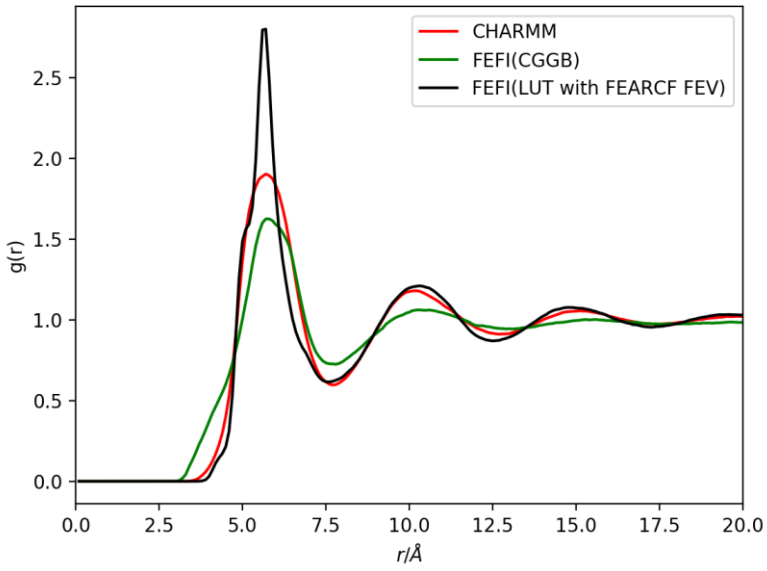


Figure 6.17: The centre-of-mass to centre-of-mass Radial Distribution of Benzene

For the centre-of-mass to centre-of-mass plots in Figure 6.17, the RDF from the FEV-driven simulation correlates well with the RDF from CHARMM in terms of position. The FEV plot also does not have the structure at close contact distances that the Golubkov and Ren plot does. However, the peak of the FEV RDF plot is much higher and sharper, indicating that the well-depth in the FEV is quite deep. The remaining peaks, however, have near identical heights to

the CHARMM RDF, something that the Golubkov and Ren parameters were unable to achieve. Another source of error is from the computation of the partial derivatives when derived from the endpoints of the surface as described in Section 6.3.3. Like the results from the PEV-driven simulation (Section 6.4), the erroneous derivatives may be contributing to the high peak.

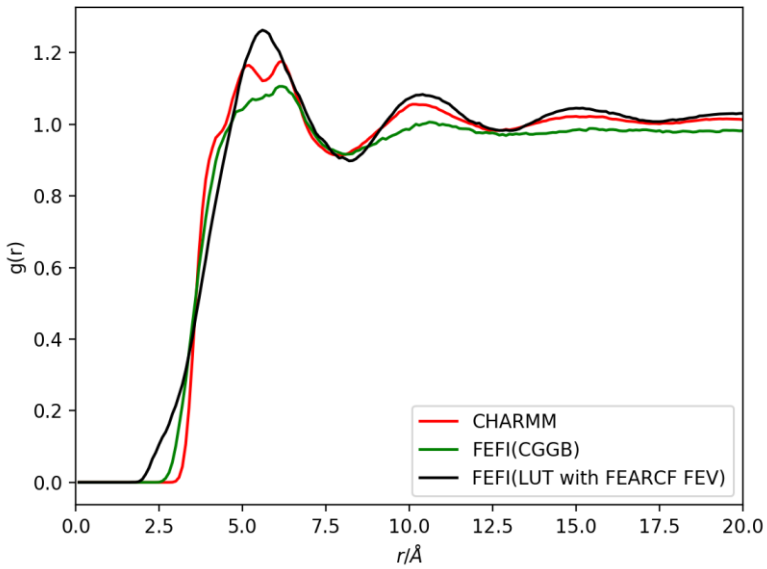


Figure 6.18: The carbon to carbon Radial Distribution of Benzene

For carbon to carbon plots in Figure 6.18, the FEV was also unable to offer the level of detail present in the CHARMM simulation. However, this is expected since the carbons are placed randomly on the plane of the benzene molecule when mapping from a CG description to the fully atomistic one. There was also some structure present at short contact distances. However, all the peak heights from the FEV plots are similar to the ones from CHARMM, correlating better than the Golubkov and Ren parameters and the GB potential.

ii. Radial-Angular Distributions

The plots for the 2D contour and 3D surface of $r\ vs.\varphi$ in Figure 6.19 were obtained from the displacement trajectories of the simulations.

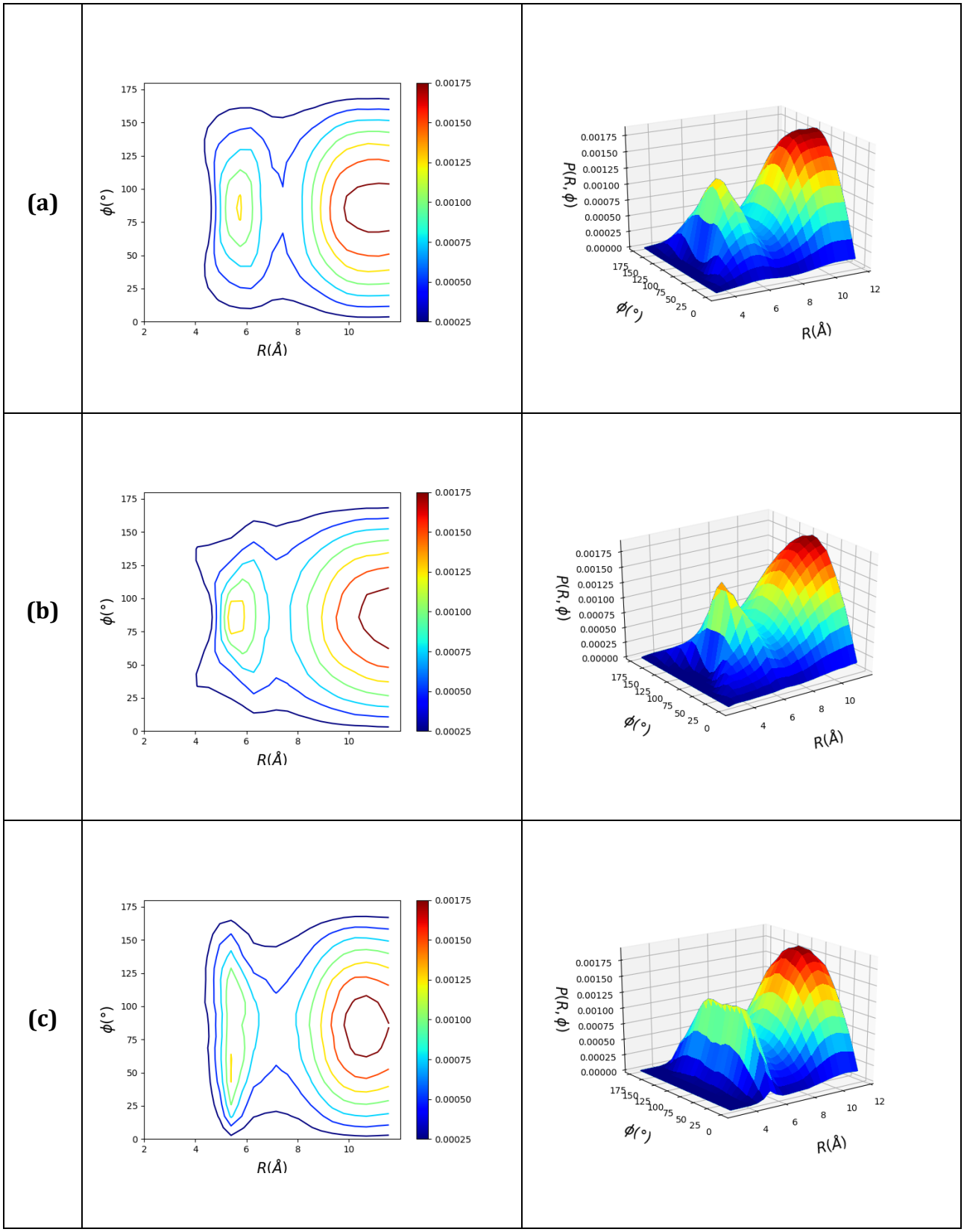


Figure 6.19: 2D contour and 3D surface plots of r $vs.$ φ from a) CHARMM b) FEFI with the CGGB module and G&R parameters c) FEFI with the LUT module and the FEARCF FEV

The CHARMM and the GB potential results form the basis of comparison for the results from the FEV. The FEV results are nosier and less structured than the CHARMM results. The

orientations appear to be more spread out. However, the heights of both peaks are similar to the ones from the CHARMM simulation.

6.6.2 Diffusion coefficient

The translational diffusion coefficients were obtained for all the simulations. The published, experimental value[10] is also provided for comparison.

	Diffusion coefficient (10^{-9} m^2/s)
CHARMM	2.18
FEFI(CGGB)	3.70
FEFI(LUT with FEARCF FEV)	2.23
Published[10]	2.20

Table 6.8: Comparing the diffusion coefficients from the three simulations

While the CGGB results over-estimated the diffusion, the one from the LUT and FEV is much closer not only to the value from CHARMM, but to the published experimental value as well.

6.7 Concluding remarks

The LUT module's computational performance is similar to that of the CGGB module while providing the ability to use a function-less 4D numerical potential. With the GB PEV, it could replicate the results from the CGGB module and the Gay-Berne potential with near perfection. Furthermore, the results from the simulation with the FEARCF FEV appreciably outperformed those from the CGGB module and Golubkov and Ren parameters when compared to the fully atomistic results from CHARMM. While some more work must be done to make the LUT module more accurate and efficient, overall the LUT was able show the performance expected in terms of the objectives established for this thesis.

6.8 References

1. Gamieldien, M. R. Parameterization of the Gay-Berne Coarse-Grain Potential from atomistically detailed anisotropic free energy volumes. Doctoral Thesis, University of Cape Town, 2012.
2. Golubkov, P. A.; Ren, P., Generalized coarse-grained model based on point multipole and Gay-Berne potentials. *The Journal of Chemical Physics* **2006,** *125* (6), 064103.
3. Naidoo, K. J., FEARCF a multidimensional free energy method for investigating conformational landscapes and chemical reaction mechanisms. *Science China Chemistry* **2011,** *54* (12), 1962-1973.

4. Gamieldien, M. R.; Strümpfer, J.; Naidoo, K. J., Hydration-Determined Orientational Preferences in Aromatic Association from Benzene Dimer Free Energy Volumes. *The Journal of Physical Chemistry B* **2011,** *116* (1), 324-331.
5. Naidoo, K. J., Multidimensional free energy volumes offer unique insights into reaction mechanisms, molecular conformation and association. *Physical Chemistry Chemical Physics* **2012,** *14* (25), 9026-9036.
6. Voth, G. A., *Coarse-graining of condensed phase and biomolecular systems*. CRC press: 2008.
7. Allen, M. P.; Germano, G., Expressions for forces and torques in molecular simulations using rigid bodies. *Molecular Physics* **2006,** *104* (20-21), 3225-3235.
8. Nilsson, L., CHARMM Analysis Tools. 2006.
9. Berardi, R.; Zannoni, C.; Lintuvuori, J. S.; Wilson, M. R., A soft-core Gay–Berne model for the simulation of liquid crystals by Hamiltonian replica exchange. *The Journal of Chemical Physics* **2009,** *131* (17), 174107.
10. McCool, M.; Collings, A.; Woolf, L., Pressure and temperature dependence of the self-diffusion of benzene. *Journal of the Chemical Society, Faraday Transactions 1: Physical Chemistry in Condensed Phases* **1972,** *68*, 1489-1497.

7. Future Work

7.1 Introduction

In Chapter 1, the two main objectives established for this thesis were:

- Designing and evaluating a parallel software solution for the determination of non-bonded interactions for a coarse-grained Molecular Dynamics simulation.
- Developing a four-dimensional look-up table (LUT) to enable the use of 4D Free Energy Surfaces (FEVs) as the accurate numerical intermolecular potential in CG simulations.

The work done in this thesis was established as a proof of concept for both objectives. In this regard, the work detailed in Chapters 3, 5 and 6 was able to meet both objectives. However, further work can be done to improve on these results.

7.2 Optimizing the GPU implementation of the CGGB and LUT modules

The optimization of the algorithms used in this thesis lay beyond its scope. It is, therefore, one of the first things that should be considered in the future development of the work. The occupancy and latency issues in some of the kernels used for the two modules were highlighted in Chapters 3 and 5. Using the Nvidia Profiler[1], solutions can be obtained for their optimization. For example, memory dependency issues can be mitigated by parallelizing memory copy and kernel launches using GPU streams[1]. The instruction-level parallelism of the code can be improved by employing techniques to group memory transactions and other similar operations. A combination of techniques can be attempted and evaluated to ensure optimum performance.

7.3 Parallelizing the computationally intensive routines in FEFI

NAMD[2] is a prime example of a highly parallelized and efficient MD simulation program. Following, its example, more of the routines of the Free Energy Force Induced (FEFI)[3] MD package could be implemented on the GPU, including setting up and updating the neighbourhood list and computing and updating the motion of the particles. Having more of the

code on the GPU will reduce the memory copy latencies and allow for more effective use of the GPU's resources.

7.4 Scaling the algorithms to multiple GPUs

While both modules have performed effectively on a single GPU, for very large systems, more than one GPU might be necessary. The load of the simulation can be split between multiple GPUs that will execute in an embarrassingly parallel fashion independent of each other. This is similar to NAMD's "patch" allocations[2].

7.5 Increasing the accuracy of the interpolation derivatives

One of the key issues highlighted in Chapter 6 regarding the cubic B-spline interpolation algorithm is the accuracy of the partial derivatives at the end points of the grid. Since the MD simulations rely heavily on the partial derivative values for force and torque computations, the accuracy must be ensured. Currently, there is a not much literature available on this. Therefore, extensive and creative work must be done to improve the results.

7.6 References

1. NVIDIA Corporation CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (accessed 25-06-2018).
2. Nelson, M. T.; Humphrey, W.; Gursoy, A.; Dalke, A.; Kalé, L. V.; Skeel, R. D.; Schulten, K., NAMD: a parallel, object-oriented molecular dynamics program. *The International Journal of Supercomputer Applications and High Performance Computing* **1996,** *10* (4), 251-268.
3. Gamieldien, M. R. Parameterization of the Gay-Berne Coarse-Grain Potential from atomistically detailed anisotropic free energy volumes. Doctoral Thesis, University of Cape Town, 2012.