Graphics Processing Unit Accelerated Coarse-Grained Protein-Protein Docking

IAN WILLIAM TUNBRIDGE

Thesis Presented for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

UNIVERSITY OF CAPE TOWN

January 2011

This thesis is all the author's own work.

Supervised by

Dr M. M. Kuttel

Assoc. Professor J. E. Gain

Dr R. B. Best

Plagiarism Declaration

1. I know that plagiarism is wrong. Plagiarism is to use another's work and pretend that it is one's own.
2. I have used the IEEE transactions convention for citation and referencing. Each contribution to, and quotation in, this thesis from the work(s) of other people has been attributed, and has been cited and referenced.
3. This thesis is my own work.
4. I have not allowed, and will not allow, anyone to copy my work with the intention of passing it off as his or her own work.

Signature	______________________
Date	______________________

Abstract

Graphics processing unit (GPU) architectures are increasingly used for general purpose computing, providing the means to migrate algorithms from the SISD paradigm, synonymous with CPU architectures, to the SIMD paradigm. Generally programmable commodity multi-core hardware can result in significant speed-ups for migrated codes. Because of their computational complexity, molecular simulations in particular stand to benefit from GPU acceleration.

Coarse-grained molecular models provide reduced complexity when compared to the traditional, computationally expensive, all-atom models. However, while coarse-grained models are much less computationally expensive than the all-atom approach, the pairwise energy calculations required at each iteration of the algorithm continue to cause a computational bottleneck for a serial implementation.

In this work, we describe a GPU implementation of the Kim-Hummer coarse-grained model for protein docking simulations, using a Replica Exchange Monte-Carlo (REMC) method. Our highly parallel implementation vastly increases the size- and time scales accessible to molecular simulation. We describe in detail the complex process of migrating the algorithm to a GPU as well as the effect of various GPU approaches and optimisations on algorithm speed-up.

Our benchmarking and profiling shows that the GPU implementation scales very favourably compared to a CPU implementation. Small reference simulations benefit from a modest speed-up of between 4 to 10 times. However, large simulations, containing many thousands of residues, benefit from asynchronous GPU acceleration to a far greater degree and exhibit speed-ups of up to 1400 times.

We demonstrate the utility of our system on some model problems. We investigate the effects of macromolecular crowding, using a repulsive crowder model, finding our results to agree with those predicted by scaled particle theory. We also perform initial studies into the simulation of viral capsids assembly, demonstrating the crude assembly of capsid pieces into a small fragment.

This is the first implementation of REMC docking on a GPU, and the effectuate speed-ups alter the tractability of large scale simulations: simulations that otherwise require months or years can be performed in days or weeks using a GPU.

Publications and Presentations

Sections of this work have been published in the following article:

1. *Simulation of Coarse-Grained Protein-Protein Interactions with Graphics Processing Units,* I. Tunbridge, R. Best, J. Gain and M. Kuttel, *Journal of Chemical Theory and Computation*, 2010, 6 (11), pp 3588-3600. DOI:10.1021/ct1003884

and presented at the following conferences:

1. CHPC National Meeting 2009, Johannesburg, RSA
   Accelerating Course-Grained Protein-Protein Docking Using Graphics Processing Units, I. Tunbridge

2. American Chemical Society Fall 2009 National Meeting and Exposition, Washington D.C., USA
   Implementation of Coarse-Grained Models for Molecular Simulation on GPU Architecture, M.Kuttel, I. Tunbridge, J. Gain and R. Best.

3. CHPC National Meeting 2008, University of KwaZulu-Natal, Durban, RSA
   Accelerating A Course-Grained Replica Exchange Monte-Carlo Method for Protein-Protein Docking Using Graphics Processing Units, I. Tunbridge.

Acknowledgements

The story of my PhD has had its fill of exciting, trying, boring and extremely frustrating times. I wish to formally thank my supervisors, Michelle Kuttel, James Gain and Robert Best. I think my supervision experience has been great. Thank you all for investing your time, effort and funding in me over the last four years.

Although I abandoned my initial project, the process was valuable, eventually allowing me to end up here. Thank you James for initially supervising and funding my MSc and trip to the NCSA, Black Ginger for workstations and nVIDIA for the free graphics card. Thank you Michelle for assuming the role of my primary supervisor thereafter, the funding and all the graphics cards. Thanks Rob for all the advice and emails in the last two years and work over Christmas to get this thesis done in time and allowing me access to the University of Cambridge GPU cluster. And finally, thanks to all of you for correcting my terrible grammar and pointing out my plentiful typos.

I also thank the CHPC for the lecturing experience (and pay) from the CUDA workshops and the national meetings, John Stone for his input when my project was in its infancy and Marco Gallotta for the use of his machine when cycles were scarce on my own.

My lab-mates: Ashley, Bertus, Dave, Jannie, Simon, Bruce, Carl, Andrew, Marco and Jason (apologies for any omissions) have made the lab so much more than the health hazard it is. Thank you for the hours of interesting, amusing and/or stimulating discussions.

Thanks to all my friends, the pub-lunch team and the ballroom dancers for balancing my life.

Thank my parents, Alan and Judy, for the support throughout my university life and for the roof over my head.

Finally, I wish to thank my fiancé, Michelle, for putting up with me through this lengthy academic adventure.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This work describes the implementation and development of fast parallel code for multi-protein docking studies. An existing coarse-graining model using replica exchange Monte-Carlo simulations is implemented, specifically using a Graphics Processing Unit (GPU) for its costly electrostatic potential calculation to drastically, improve the tractability of simulations performed using this model.

The formation of both transient and permanent multi-protein complexes is integral to many biological processes. Some examples are antibody-antigen and protease-inhibitor complexes; protein complexes involved in cellular signal transduction processes; structural proteins that maintain the shape of a biological cell, and the very large multi-protein complexes represented by the proteasome, the nuclear pore complex and viral capsids. Identification of the docking sites and binding characteristics of these proteins provides understanding of their cooperative roles in common cellular functions. This improves our understanding of disease mechanisms and provides the basis for new therapeutic approaches. Consequently, the prediction of protein binding sites has been identified as one of the ten most sought-after solutions in protein bioinformatics [1] and is closely related to the well-known NP-hard "protein folding problem" of predicting the three-dimensional structure of a protein from its primary sequence [2].

In the absence of sufficient experimental data on the atomic structure of protein complexes, popular methods such as Molecular Dynamics or Monte Carlo simulations of protein complex components can assist in determining both their mode of interaction and the location of the interaction site(s) [3]. Molecular simulations generate an ensemble of configurations, from which both structural and thermodynamic data can be extracted. The configurations representing bound protein complexes enable identification of both docking sites and the relative orientation of the proteins, while an estimate of their binding affinity, a numerical description of the strength with which proteins bind, can be obtained from the proportion of bound samples occurring in the ensemble.

However, all-atom simulations of multi-protein complexes are highly computationally expensive and are therefore limited in scale by the available computing resources. Simulations are typically restricted to simple biological systems (e.g, small binary protein complexes without solvent) and nanosecond time scales. Accurate coarse-grained models have helped to extend molecular simulations to more biologically relevant lengths and time-scales [4–12]. These reduced molecular models aggregate single atoms into large spherical beads to significantly decrease the computational cost of a simulation. There is considerable potential for further accelerating molecular simulations by combining coarse-grained models with the computational power of massively parallel graphics processing units (GPUs).

1.1 Macromolecular GPU Implementations

Modern GPUs have floating-point computation capabilities far in excess of current CPUs. GPUs are Single Instruction Multiple Thread (SIMT) compute devices; organising data into homogeneous streams of elements and executing a function, or *kernel*, on all elements of a stream simultaneously. Current high-end GPUs also have high memory bandwidth compared to CPUs. For example, the nVIDIA GTX280 has 240 fragment or stream processors and theoretical memory bandwidth of 141GB/s. As a consequence, these compact devices are capable of rapid high-throughput numeric operations and can be employed effectively by non-graphical, data-parallel, memory-bound algorithms of high arithmetic intensity, such as the N-body problem inherent in molecular simulations. For all-atom and coarse-grained potentials, evaluation of the total interaction potential between all bodies, N, in a protein molecule is an $O(N^2)$ operation and the chief performance bottleneck - a common feature of N-body simulations in general. The independence of each pairwise interaction means that the calculation of all such potentials suits the vector-like GPU architecture, promising good speed-ups over CPU-only implementations.

The difficult task of porting algorithms to the GPU architecture, while maintaining effective use of the CPU, has been made easier with the development of general application programming interfaces. In 2007, nVIDIA released the Compute Unified Device Architecture (CUDA) API, which allows the general programmer direct access to the nVIDIA GPU hardware. CUDA allows for operations not supported by graphics APIs, such as local data communication between kernels and scatter and gather operations. However, CUDA GPU programming is not trivial [13]. Programmers must be mindful of the GPU memory hierarchy, which requires explicit management to minimise access latency and effective packing of data to enable a coalesced memory access pattern. In addition, maximising GPU performance often requires latency hiding through exploitation of the multi-threading capabilities of the CPU cores [14], adding the difficulties of conventional multi-threaded asynchronous programming to the GPU-specific programming techniques.

However, despite these difficulties, there are increasing reports of successful CUDA implementations of N-body algorithms achieving good speed-ups over CPU implementations [15–17].

Specifically, GPU-based calculations of the expensive long-range electrostatics and other non-bonded forces necessary for molecular mechanics simulations are typically 10 - 100 times faster than heavily optimised CPU-based implementations [18–20]. Friedrichs et al. [17] show speed-ups over a single CPU implementation of up to 700 times for large all-atom protein molecular dynamics running entirely on the GPU. However, such massive speed-ups are not always achievable: a recent implementation of an acceleration engine for the solvent-solvent interaction evaluation of molecular dynamics simulations shows speed-ups of up to a factor of 54 for the solvent-solvent interaction component, but only 6-9 for the simulations as a whole [21].

Previous implementations of N-body dynamics on a GPU, such as the GRAPE implementation [16], translate the potential evaluation into a convenient map-reduce problem [15–17], the map calculates the net potential contribution from each body and the reduce sums these together to determine the free energy of the system. The Kim and Hummer coarse-graining model, implemented in this work, requires very frequent random-access lookups, determined by indirection, in evaluation of the interaction potential. Standard GPU memory-use models typically discourage indirection since it produces divergent branch behaviour in threads on the GPU, which result in degraded performance [22,23]. Therefore, in order to optimise the parallel performance of our implementation, we assessed the performance impact of storing the structural data and potential lookup table using the various types of memory available on a GPU to establish the optimal memory usage configuration.

The parallelization approach developed in this work is generally applicable to N-body problems that require similar random access lookups. This often occurs where the aspects of the interaction between bodies are dependent on their type or state. One instance is the commonly-used energy functions in molecular dynamics simulations, in which the interactions depend on the type of each atom, particularly in the case of bonded forces in all atom models [24] and coarse grain models [11,12].

1.2 The Kim and Hummer Coarse-Grained Method

The aforementioned cases of coarse-graining [4–12] already help to extend simulations to more biologically relevant time-scales and sizes, but additional speed-ups offered by GPU acceleration stand to further increase the tractability of such simulations, and in turn, allow for the simulation of much larger systems in tractable time periods.

Kim and Hummer have developed a coarse grain model specifically for protein-protein docking simulation by replica exchange Monte-Carlo [12]. Their model aggregates the atoms of each amino acid into a single spherical bead, representative of the charge, radius and amino acid type. Monte-Carlo simulations are biased random search algorithms; random mutations are performed upon a system and a scoring function used to evaluate the change, accepting improvements and discarding worse state. Monte-Carlo simulations exploring the docking space are performed using rigid body translations of the proteins with chain flexibility applied amino acids linking secondary protein structures. Monte-Carlo moves are evaluated using the free energy of the system, namely the sum of the non-bonded potentials resulting from the Lennard-Jones and Coulomb forces between the beads and potential arising from the stretching, torsional and angular forces between bonded atoms.

This model is shown to correctly determine the binding affinity of the complexes studied in addition to discovering the correct binding interfaces within 2Å to 5Å DRMS. Their research suggests that their model and energy function should be transferable to other protein-complex studies [12].

The combination of replica exchange Monte-Carlo, coarse graining and a verified model offer a valuable starting point for a viable GPU accelerated docking application. Numerous works show that the parallel nature of electrostatic calculations makes them amenable to GPU implementation [17, 18, 25, 26], coarse-graining improves tractability by reducing the time required to perform simulations owing to fewer bodies per system, enabling the study of longer simulations and/or larger simulations [10]. Finally, the parallel nature of replica exchange makes such a method scalable to multiple GPU processes on one or many compute nodes, affording a maximal amount of high level parallelism in addition to the low level parallelism of the electrostatics on the GPU.

This work reports a hybrid CPU-GPU parallel implementation of a coarse-grained Replica Exchange Monte Carlo simulation protocol for simulation of multi-protein complexes, recently developed by Kim and Hummer [12]. We implement the original Kim and Hummer model and simulation methods, focusing on the development of a general, highly scalable GPU implementation with the goal of increasing the size of tractable simulations. This implementation is used to simulate docking on a biologically relevant scale, under either crowded conditions or in a large assembly, illustrating the utility of the implementation and method. Specifically, two test simulations are performed, the first application investigates the affects of macromolecular crowding upon a docking simulation. The second investigates the construction of a viral sub-capsid.

1.3 Aims

The foremost aim of this project is to develop a generic application for the simulation of multi-protein docking based on the Kim and Hummer. coarse-graining and potential model and accelerated by a GPU. This implementation aims to reproduce results attained by this model in accelerated time compared to a CPU.

Performance is benchmarked, with the goal of determining the critical performance factors of a coarse-grained model on a GPU, considering that a degree of indirection arises from the coarse graining in contrast to the recommended GPU memory model. Determining an optimal performance configuration for the generic case is coupled with the development of the application, thus, benchmarking will aim to determine the manner in which GPU memory should be managed and the effects of such management in the context of a generic n-body like algorithmic decomposition with an indirect data dependency.

Achievement of a high performance implementation allows the study of two biologically relevant simulations. Simulating large systems and assemblies of the order of 100,000 atoms (macromolecular crowding and sub-capsid simulations) is intractable on a single GPU. Using the aforementioned application, investigation of the affects of macromolecular crowding and evaluating the results in the context of scaled particle theory will indicate the utility of the Kim and Hummer model for studying such large-scale interactions. Second to this is the investigation of sub-capsid and whole capsid viral assembly simulations. This investigation seeks to explore the ability of this application to simulate this assembly.

1.4 Approach

1.4.1 GPU Design and Implementation

Implementation is approached in an iterative manner: initial design and implementation of the model using only the CPU is followed by synchronous GPU implementation with validation against the CPU and finally asynchronous and multiple GPU functionality. Extensive validation follows, inspecting the accuracy of the GPU versus the CPU as well as binding affinities and structural evaluation. Finally, the success of the implementation is evaluated through quantitative benchmarking, and qualitatively via application to relevant macromolecular simulations. Each developmental phase builds upon prior work, culminating in the use of the implementation to produce a biological result.

The relevant GPU programming techniques applicable to such a model are isolated and refined to implement the Kim and Hummer coarse-grained model on a GPU. The implementation is mindful of the need for both accuracy and speed, designing a scalable, heterogeneous solution capable of fully utilising the CPU and zero to many GPU devices through multi-threading and asynchronous GPU utilisation.

Expensive electrostatic calculations are performed using the GPU. Because of the separable parallel relationship between the pairwise potentials, such a scheme scales favourably on a GPU. Summation of these potentials is performed in part on the GPU using parallel reduction at a kernel level, followed by a final summation on the CPU. Other operations such as Monte-Carlo mutations and replica exchange are performed on the CPU. The division of work is based on the algorithmic complexity of the parts, electrostatics is $O(n^2)$ while mutations, random number generation and CPU summation are linear in complexity. The final optimisation of multiple GPU streams results in full utilisation of both GPU and CPU, with the GPU performing electrostatics calculations for one replica in parallel with mutations and Monte-Carlo acceptance/rejection on the CPU for other replicas.

Validation is performed with comparisons of the potentials between CHARMM [27], the CPU and GPU. Verification of the simulations is performed by reproducing the results of two simulations from Kim and Hummer [12] and comparing the binding affinities and the emergent structures to known values and structures.

Both GPU and CPU implementations are benchmarked, determining an optimal generic case with which simulations can be performed. Low level benchmarks, such as block size and memory configuration, measure the affect of simulation parameters such that kernels may be tuned and a single Monte-Carlo simulation optimised. Higher level benchmarking measures the affects of multi-threading and asynchronous GPU usage, maximising resource utilisation on the host machine.

1.4.2 Macromolecular Crowding and Viral Capsid Applications

Two studies are selected as tests for this implementation, one simulates the formation of a protein-protein complex while influenced by other molecules while the other simulates the formation of a multi-protein complex from symmetric protein molecules.

The first application simulates the macromolecular crowding effects of crowder proteins on the binding characteristics of a specific complex, yeast cytochrome *c* to cytochrome *c* peroxidase [28]. This study attempts to verify the predictions of scaled particle theory through simulation. Protein simulations are typically performed in isolation: only the participating molecules of interest are present but, with a GPU accelerated simulation the affects of introducing additional molecules into the simulation can be studied, aiming to more closely mimic the crowded conditions occurring within living cells.

The second application of this implementation is to simulate viral capsid assembly. This also serves as an investigation into the feasibility of using the Kim and Hummer's model for such systems. Performing replica exchange Monte-Carlo simulations to investigate the binding strength, clustering characteristics and produce emergent bound structures will guide the development of future simulation models.

1.5 Contributions

This is the first implementation of a heterogeneous CPU-GPU course-grained replica exchange Monte-Carlo model. The implementation of this model for macromolecular docking using nVIDIA's CUDA technology is a necessary step in the advancement of docking simulations, adding evidence to the growing body of scientific work illustrating that the use of GPU technology is essential for macromolecular simulation. The novelty of the implementation arises from the heterogeneous parallelization scheme and synthesis of n-body electrostatics CUDA schemes and the coarse-grain potential from Kim and Hummer. We exploit the heterogeneity of the GPU-CPU architecture as opposed to attempting to fully implement our simulations on the GPU, as is the case in many MD simulations [17, 25] favouring overall throughput over single thread performance.

A new software tool designed for hybrid high performance computing architectures combining multi-core CPUs with GPU accelerators is produced. The relatively low-cost parallel architecture of the GPU when combined with such software shows that significant results are attainable at very low cost when compared to more traditional, exceptionally expensive CPU clusters. This tool exhibits speedups comparable to those of other implementations, speeding up simulations by factors ranging from 10 to 1400 times that of a serial CPU-only simulation.

In validating the GPU implementation, investigation of the affects of using either a truncated or full length ubiquitin protein showed that the presence of ubiquitin's C-terminus tail increases its binding affinity and the ratio of correctly to incorrectly bound helix orientations in simulations and furthermore increases the specificity of the binding.

Macromolecular docking simulations provide compelling evidence of the predicted effects of macromolecular crowding on weakly bound complexes, resulting in increased binding affinity in agreement with the scaled particle theory (SPT) derived theoretical model. The use of our software is necessary for this verification due to the intractability of these simulations using only conventional CPU clustering technology.

Finally, we show that Kim's coarse grained model can successfully predict the binding interface and configuration of the dimeric viral molecule 2g34 from the HBV virus. Initial studies reveal the shortcomings of the typical two protein-complex simulation model in simulating entire capsid assemblies and why these methods will ultimately fail, allowing us to derive a new method which has the potential to accurately assemble course grain viral capsids.

The application studies outline the utility of our novel implementation and the importance and potential in the combination of course-grained reduced complexity models and GPU acceleration in simulating massive assemblies or systems.

1.6 Thesis Organisation

The remaining chapters of this thesis are organised as follows: Chapter 2 reviews the GPU programming model and protein-protein simulations with discussion regarding the overlap of these fields and prior results pertaining to molecular simulation using GPUs. This chapter serves as an entry point for both computer science and computational chemistry practitioners, containing sufficient explanation to motivate the decisions and provide perspective to the results obtained in later chapters. Chapter 3 presents a system design, describing the implementation and the rationalisation employed to produce our implementation. In Chapter 4, we discuss the implementation in depth, providing sufficient detail for others to accurately reproduce our results using only this document. Chapter 5 validates our implementation though a series of accuracy and simulation test cases. Chapter 6 reports our GPU kernel benchmarking and profiling results leading to the overall performance results reported in Chapter 7. Chapter 8 presents our macromolecular crowding and partial viral capsid studies, ending with a discussion of future applicability to simulation. Conclusions are presented in chapter 9 together with a general future work discussion to end the main body of work.

Appendices present supplementary data required for the simulations, pseudo-code and performance figures omitted form the main text.

Chapter 2

Graphics Processing Units

Here we outline the development of general purpose GPU programming in recent years, with particular focus on changes in the hardware that have enabled porting of algorithms to the GPU. This is followed by an overview of protein folding and docking techniques, with reference to existing GPU implementations of these methods in the following chapter.

Discrete graphics processing units are designed primarily as high performance devices for rendering images and geometry in applications such as computer games. The term GPU was coined by nVIDIA in 1999 for its GeForce 256: a processor with integrated transform, lighting, triangle setup/clipping and rendering engines [29]. The steady increase in the demand for graphics hardware performance, driven predominantly by the gaming industry, has resulted in highly parallel, high performance commodity hardware [22].

In terms of compute performance, GPUs and CPUs were initially similar. However, by 2003 the demands of real-time 3D graphics applications resulted in GPU FLOPS and bandwidth outstripping the CPU. Currently, GPUs from both nVIDIA and ATI far outperform that of Intel processors (Figure 2.1).

Unlike the CPU, the GPU does not use large data caches to decrease memory latency because spatially local caches only benefit a few threads. Instead, GPUs use fast context switching and massive multi-threading to hide the performance gap between on-chip and off-chip memory access. Computation and I/O are independent, so threads can be either processing, waiting, or performing I/O, which ensures that all the cores and the wide data bus are always in use. This strategy sacrifices single thread performance to overall performance across all threads.

Like CPUs, the clock speeds of GPUs are limited both by the materials from which they are fabricated and power consumption. Thus, meeting the demands of the gaming industry required wider architectures, culminating in the current flagship offerings from nVIDIA and ATI. From nVIDIA, the 480 core *Fermi* architecture (GF100) is capable of 1.3 TeraFLOPS single precision performance and 177.4 GB/s memory bandwidth [30]. ATI has the *Cypress XT*, with 1600

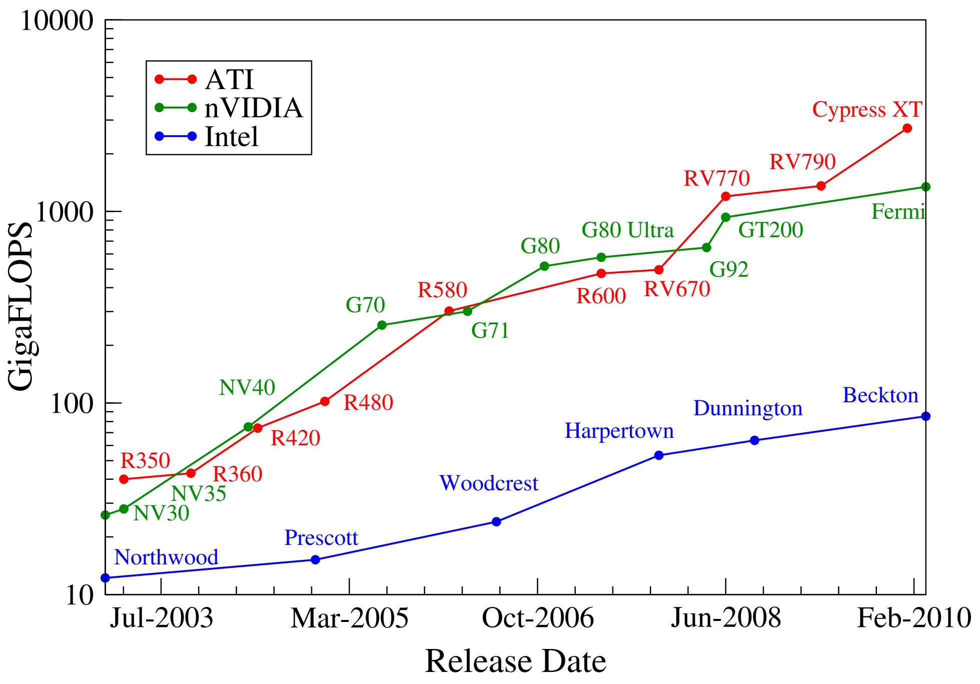


Figure 2.1: GPU and CPU 32-bit FLOPS Performance
Current GPU offerings from both ATI and nVIDIA outperform CPUs by an order of magnitude. Fermi and Cypress chip-sets are capable of 1345 and 2720 GFLOPS peak theoretical 32-bit performance, respectively, in a single GPU per PCB configuration compared to an octacore CPU theoretical peak performance approaching the 100 GFLOP mark. In early 2003, GPUs and CPUs showed similar performance figures but, driven by the need for higher throughput in computer graphics applications GPUs adopted wider architectures opening up a performance gap between GPUs and CPUs.Peak theoretical performance for nVIDIA GPUs can be calculated as the number of cores multiplied by the shader clock rate multiplied by 2 for each fused multiply and add (FMAD) that these GPUs can perform per cycle. The GT200 can perform both a FMAD and multiply in the same cycle due to its dual issue capability. ATI performance is calculated by multiplying the shader clock speed by the number of unified shader cores, each of which performs one operation per clock cycle.Each CPU core can perform 4 operations per cycle per core. Data up to 2007 from Owens et al. [37]. Data for each processor is available from one of either the nVIDIA, AMD or Intel websites.

stream processors capable of 2.7 TeraFLOPS single precision compute performance and 153.6 GB/s memory bandwidth [31].

GPUs are especially well suited to data-parallel algorithms with a high ratio of floating point calculations to memory operations (or arithmetic intensity) Data-parallel computation is ideally embarrassingly parallel - the same sub-program is executed on many data elements independently suiting programs - lacking the requirement for sophisticated control flow hardware found on conventional processors such as CPUs [22].

While similar in many ways, it is the local load and store capability of GPU that distinguishes them from true SIMD/vector processors. The GPU can be viewed as a streaming processor because the programming model encapsulates computational locality through the use of *streams* and *kernels*, which are able to manage the local memory of each process; this is in contrast to vector processors, which read data from off-chip memory and write back data to off-chip memory. The term *stream* refers to a collection of records requiring a similar computational operation performed on each element, while *kernels* encapsulate the computational operations for each element of a stream. The streaming processor executes a kernel over all the elements of an input stream, writing the result to an output stream.

Computer graphics algorithms map massive numbers of pixels and vertices to parallel threads. Multimedia processing applications (such as post-processing, video encoding and decoding, image scaling, stereo vision, and pattern recognition) map image blocks and pixels to parallel processing threads in a similar way. Further generalisation of these type of algorithms have applications in many fields which stand to benefit from data-parallel processing, such as signal processing, physics simulations, computational finance and computational biology [22].

GPU computation has matured in recent years, evolving from the GPGPU (General processing using Graphics Processing Units) approach of mapping problems to geometry and texture domain operations and using the graphics API to perform the calculations, to the fully programmable approach afforded by current generation hardware exposed through the Compute Unified Device Architecture (CUDA) [22] and Close to Metal (CTM) [32] APIs.

2.1 General Processing using Graphics Processing Units

GPGPU programming originates in programmable frame-buffer devices and systems such as the *UNC PixelPlanes* series [33] where SIMD embedded pixel processors on a single chip operated as frame-buffer memory. The abstraction of OpenGL as a SIMD processor using programmable shaders [34] allowed programmers to use GPUs through OpenGL API calls. Then, APIs such as Cg, GLslang, HLSL allowed shaders to be written in a high level, C-like language [35]. However, a shortcoming of all these techniques is that the programmer had no control over the remaining components of the graphics pipeline, such as memory allocation, loading shader programs or constructing primitives, which forced not only an understanding of the latest graphics hardware and APIs, but also that the programmer express all of their algorithms in terms of graphics primitives, i.e. polygons or textures.

Graphics processors circa 2004 featured programmable vertex and fragment processors, where each processor executed an assembly level program consisting of standard mathematical instructions, e.g. 3 or 4 component dot products, texture fetch instructions and special purpose functions. The programming model was a set of kernels acting upon streams, but this had to be programmed as a sequence of shading operations acting upon graphics primitives. The programmer also had to perform explicit stream management, explicitly calling a graphics API function on data manually packed into textures or vertices and transferred to the GPU [35]. Program-

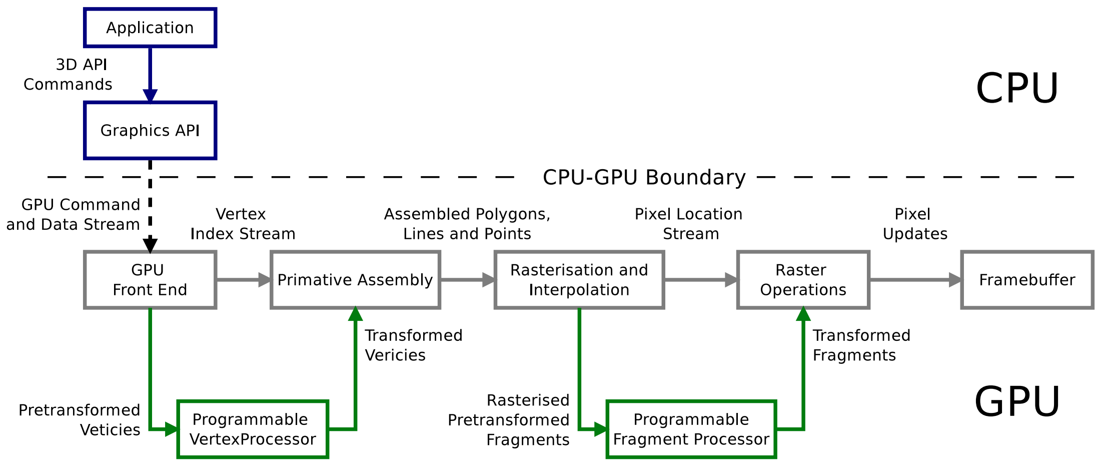


Figure 2.2: The Programmable Graphics Pipeline
On a programmable GPU, vertex and fragment processing is assigned to programmable units running the vertex and fragment programs written by the programmer (green), these commands bypass parts of the fixed function pipeline (grey) that would other wise perform vertex and fragment processing. All computation is driven through the graphics API on the CPU (blue). Results stored in the frame-buffer can be transferred back to the CPU or recirculated in the pipeline. (Diagram derived from The CG Tutorial [36])

mers needed to be mindful of hardware limitations, texture sizes, shader operation counts and shader outputs. Furthermore, the lack of some fundamental computing constructs (e.g.. scatter operations and integer operands) made GPUs ill-suited to certain computationally intensive tasks, such as cryptography [37].

Nevertheless GPUs were effectively used for cell-based simulation techniques, such as cellular automata [37], dynamics simulations governed by PDEs [38], and lattice simulation approaches such as Lattice Boltzmann Methods (LBM) for modelling fluids [39].

Hardware improvements (e.g., integer support and improved single precision) enabled more sophisticated general computation on a GPU, such as finite difference and finite element techniques for solving partial differential equations [37], implementations of the Navier-Stokes equations [40, 41] and smooth particle hydrodynamics [42]. Rigid body dynamics were successfully implemented on a GPU achieving over than 10 times speed-up compared to a multi-core CPU implementation [43].

GPUs have also been used for image, video and signal processing [44, 45], such as segmentation of MRI and CT scans [46], computer vision [47], fast Fourier transforms [48], discrete cosine and wavelet transforms for use in MPEG and JPEG compression [49] and linear algebra applications [40, 50, 51]. Geometric applications include ray tracing, photon mapping, radiosity calculations, subsurface scattering [52,53], constructive solid geometry operations [54], distance fields and skeletons, collision detection, transparency, particle tracing, geometric compression and level of detail techniques [55,56]. GPUs have even been used to speed-up database queries [57].

Efforts to abstract the GPU programming model such that it no longer involved graphics APIs emerged in forms such as *BrookGPU*. Brook, from Stanford University was originally designed for streaming supercomputers [35] and exposes the GPU as a streaming coprocessor to the CPU via the use of *streams*, *kernels* and *reduction* operators. This enables a programmer to produce program in Brook which will compile and run on any hardware for which there is a Brook implementation.

Cognisant of the increasing need for more programmable GPUs, vendors nVIDIA and ATI have developed architectures that no longer restrict developers to a fixed function pipeline, instead offering a fully programmable scalable architecture and associated APIs. CUDA from nVIDIA [22] and CTM from ATI [32] expose current generation GPUs at a low level, bypassing the graphics programming APIs. More general approaches such as OpenCL have also emerged, allowing general computation on any GPU independent of vendor [58]. With these technologies, no knowledge of graphics programming is required in order to use a GPU for general computation. Since their initial releases: nVIDIA continue to develop CUDA, while ATI is focussing its efforts on *ATI Stream Technology* implementing OpenCL for general GPU computation. nVIDIA supports both CUDA and OpenCL for its GPUs.

Other GPU computing technologies include Microsoft's DirectCompute[1] and Intel's Larrabee project. Larrabee is not a GPU specific hardware, but features a full x86 instruction set and cache coherency [59]. The relative importance of this proposed architecture has been diminished by 2010's GPU offerings, such as *Fermi* from nVIDIA and *Cypress* from ATI, which further generalise computation on the GPU. The Fermi architecture implements unified address spaces with full C++ support, full IEEE floating point precision, true cache hierarchies with unified caches, error correction code memory support [60] and thus incorporates many features of Larrabee which would have made it superior to true GPUs. The *Fermi* architecture improves on previous generations with 32 CUDA cores (SPs) per SM and improved double precision floating point performance, a dual warp scheduler, an increase in the amount of shared memory and double the number of special function units (SFU), resulting in an even more generally programmable GPU architecture in addition to the increase in performance [60].

With the increase in speed and accuracy coupled with the new programming interfaces, the number of general purpose applications ported to GPUs have increased. nVIDIA's CUDAZone[2] and ATI's Stream Developer Showcase[3] offer showcases for each companies' GPU programming technology. Fields benefiting from GPU acceleration, specifically CUDA, include molecular dynamics and computational chemistry [17,18,25,61–63], life sciences and bioinformat-

[1]http://www.microsoftpdc.com/2009/P09-16, accessed 2010-12-06

[2]http://www.nvidia.com/object/cuda_home.html, accessed 2010-12-06

[3]http://developer.amd.com/samples/streamshowcase/, accessed 2010-12-06

ics [64,65], physics [16,66,67], biological and medical imaging [68–70], financial mathematics [71], mathematics [14], climatology and oceanography [72–74] and video, imaging and computer vision [43, 75–79][4]. Many more applications benefit from CUDA by using CUDA mathematics libraries [70] and random number generators [43] as well as CUDA accelerated versions of MATLAB, R and Mathematica. Computational chemistry makes good use of both nVIDIA and ATI GPUs for research. Stanford's Folding@Home, the most powerful distributed computing cluster in the world, reports that of the 5 PetaFLOPS of compute power at its disposal[5], 3.2 PetaFLOPS are attributed to GPUs (2.6 to nVIDIA and 0.7 to ATI respectively). Clients for Folding@Home use either CUDA, CAL or OpenCL to interface with the GPU.

2.2 The CUDA Programming model

CUDA exposes the GPU as a generally programmable device, as opposed to a fixed function graphics pipeline where GPGPU is achieved by programmable shaders. In essence, CUDA requires the programmer to transfer data to the GPU, invoke a kernel upon that data and copy the result back to the host. Three critical optimisation strategies be adopted in order to make effective use of a GPU with CUDA: maximising parallel execution, optimising memory usage to achieve maximum memory bandwidth and optimising instruction usage to achieve maximum instruction throughput [23]. Optimisation within each thread and group of threads in the CUDA aims to maximise the available bandwidth and compute resources of the GPU and so maximises the compute performance of an algorithm.

2.2.1 The Compute Unified Device Architecture

The G80, introduced in 2006, was the first Compute Unified Device Architecture (CUDA) compliant GPU. This architecture was the first GPU to support C, as opposed to a shading language. The G80 combined vertex, geometry and pixel pipelines into a single unified processor, referred to as a *Streaming Multiprocessor* (SM) [60]. A scalable array of these SMs is at the core of the CUDA architecture. A high end GTX 280 has 30 SMs and a mainstream GT 220 has 6 SMs, but both prescribe to an identical programming model.

A streaming multiprocessor consists of a number of 32-bit *scalar processors* (SPs) and special function units, together with a multi-threaded instruction unit, shared memory, local registers and a constant cache. The GT200 architecture has a 64-bit precision unit in addition to these features, to enable double precision calculations in hardware. In G80 and GT200 series GPUs, these SMs are clustered in a *texture processing cluster* (TPC) where they shared texture unit

[4]Application list available at http://www.nvidia.com/object/cuda_app_tesla.html, accessed 2010-09-09
[5]http://fah-web.stanford.edu, accessed 2010/08/05

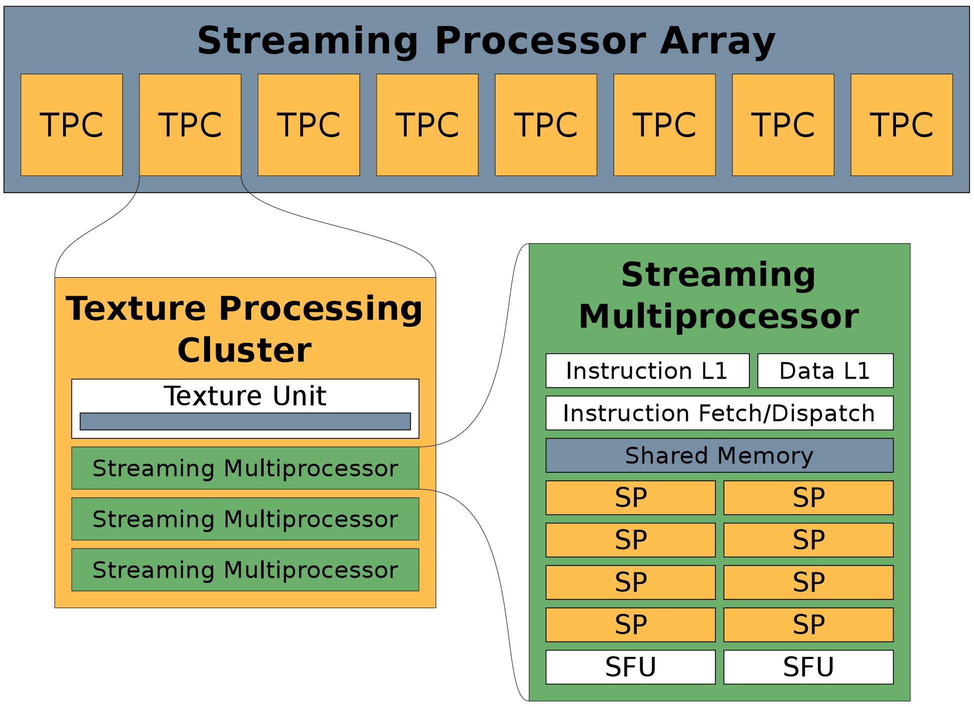


Figure 2.3: CUDA 1.0-1.3 Architectures
Streaming multiprocessors contain an array of scalar processors (SPs) together with instruction, data, local caches and special function units. SMs are grouped in a Texture Processing Cluster (TPC) with a shared texture cache and texture unit. A streaming processor array of TPCs ultimately determines the performance of the entire GPU, linearly scaling the performance of each TPC. The GT200 differs from the G80 GPU architecture in that it contains an additional SM per TPC. For different products the size of the streaming processor array is varied, so determining the number of cores: 240 core GTX280 (GT200) GPUs contain 10 TPCs, 128 core 8800GTX (G80) GPUs contain 8 TPCs. (Figure derived from Stone et al. [18])

and texture cache. G80 TPCs contain 2 SMs and GT200 TPCs contain 3 SMs, with the scalable array being assembled from texture units. Figure 2.3 illustrates the hierarchy of streaming multiprocessors, texture clusters and streaming processor array in the G80 GPU. Each G80 SM has 8 stream processors and 2 special function units. The stream processors perform floating point arithmetic in parallel with the special function units performing fast square roots, trigonometric, exponential, power functions and other mathematical functions [22].

The CUDA parallel programming model overcomes the challenge of scalable hardware with three key abstractions - a hierarchy of thread groups (thread blocks), shared memories, and barrier synchronization - all of which are exposed to the programmer as a minimal set of language extensions in C [22]. These abstractions provide fine-grained data parallelism and thread parallelism mechanisms. The thread groups impose a coarse subdivision of a problem into smaller independently solvable parts, within which, fine-grained thread parallelism and data sharing is used to find a solution. This data and task decomposition allows each sub-problem to be independently scheduled (Figure 2.4).

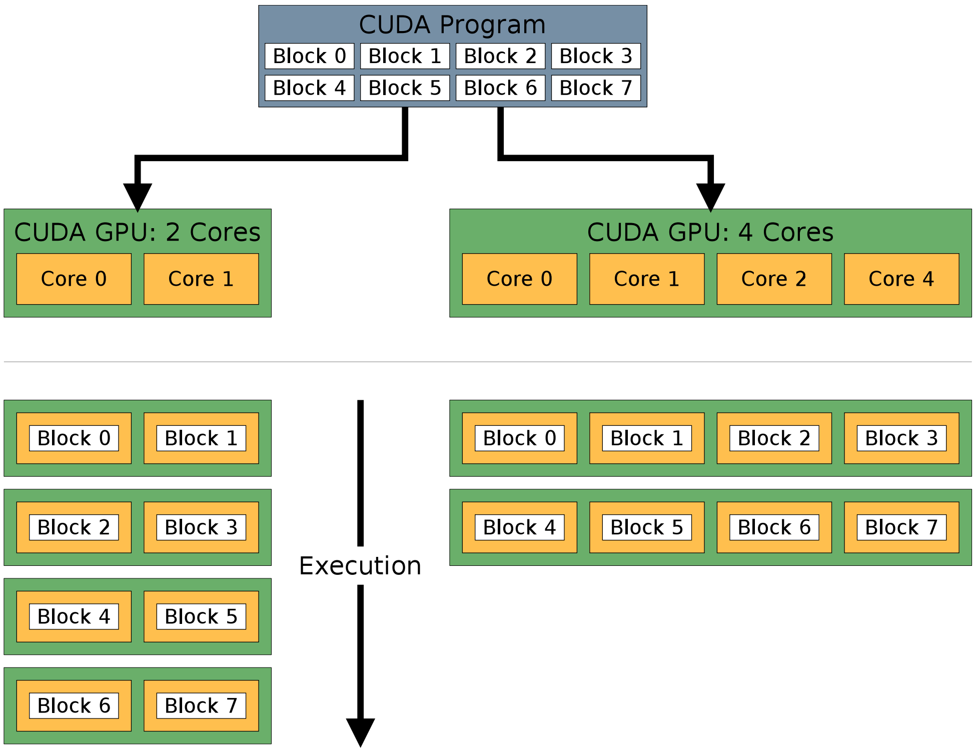


Figure 2.4: CUDA Hardware Scheduling
The data parallelism of CUDA programs results in identical program behaviour across any number of scalar architecture of streaming multiprocessors (SMs). Thread blocks are scheduled on the available hardware and thereby ensure that the same CUDA application can execute on any CUDA capable device without modification. (Figure derived from the CUDA Programming Guide [22])

CUDA requires no understanding of the GPU hardware in order to attain excellent performance because of the level of abstraction the API provides. However, knowledge of the hardware helps to optimise the performance of a CUDA program.

Multiprocessors create, manage and execute concurrent threads in hardware with no scheduling overhead. Threads executing on an SM can communicate via shared memory. Hundreds of threads running different programs can be managed by an SM by mapping each thread to one scalar processor core such that it executes independently with its own instruction address and register state. The multiprocessor SIMT unit creates, manages, schedules, and executes these threads in groups, called warps, of 32 parallel threads. The individual threads in a warp start with the same initial state and address, but are free to branch and execute independently.

More than one thread block can be assigned to an SM at a single time. Thread blocks are collections of threads processing spatially-local data and can have one, two or three dimensions.Thread blocks are arranged in a one- or two dimensional grid (Figure 2.5), thus allowing the data to be divided into schedulable work units on the GPU.

For the GT200 architecture, a CUDA grid can be arranged in any manner, as long as the

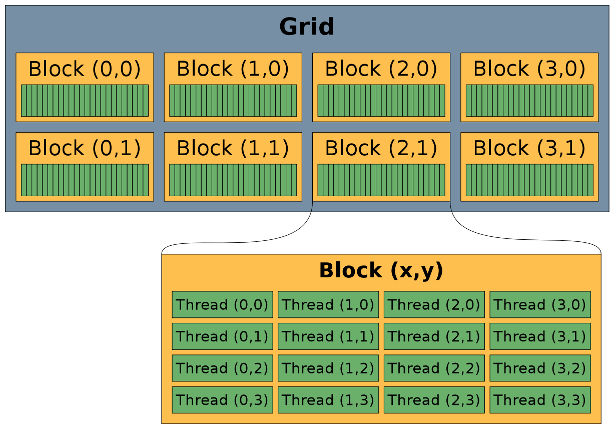


Figure 2.5: CUDA Thread Hierarchy
CUDA organises threads into groups called thread blocks which in turn are assigned to a grid of thread blocks. Blocks compose a 1 or 2 dimensional grid while threads compose 1, 2 or 3 dimensional blocks.

product of the dimensions does not exceed 2^{16}, i.e. grids of 65535 by 1 or 256 by 256 are both valid, but the former will address data linearly and the latter in 2 dimensions. Thread blocks are restricted to a maximum of only 512 threads per block, with maximum sizes of 512, 512 and 64 for its respective x,y and z dimensions. Thus, this architecture can schedule up to 33 554 432 threads. Unlike CPUs, which would "thrash" if this many threads were contending for the processor, the GPU is specifically designed for the massively multi-threaded usage case. These blocks are split into independent warps of 32 contiguous threads to be scheduled by the SIMT unit. Warps are not controlled by the programmer: their size is defined by the driver and CUDA runtime environment to make efficient use of the parallelism within each SM. When issuing instructions, the SIMT unit selects a warp and issues the same instruction to the active threads in that warp. Thus, warps are most efficient when all 32 threads follow the same path of execution, because divergence is handled by executing each conditional branch serially.

CUDA requires no explicit knowledge of the underlying hardware configuration to program a GPU. Kernels execute the same code for every thread in every block. The CUDA software stack, shown in Figure 2.6 provides the mechanism by which kernels are invoked and data is prepared on the GPU. At the bottom of the stack is hardware layer and on the top is the programmers GPU computing application. The layer in the middle can be CUDA C, OpenCL, Direct Compute or CUDAFortran, all of which interact with the GPU using the CUDA runtime. The programmer controls the flow of the CUDA application: usually a copy of data from the host to GPU memory, followed by a kernel invocation on the GPU. The kernel then processes the data in parallel using as many threads as possible, whereupon the data is transferred back to the host's memory.

The CUDA programming model assumes that the CUDA threads execute on a physically separate device that operates as a coprocessor to the host. Kernels execute on a GPU and the rest of the program executes on a CPU. The CUDA programming model also assumes that both

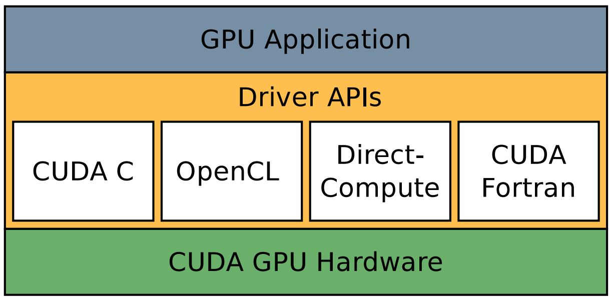


Figure 2.6: The CUDA Software Stack
CUDA hardware forms the base of the CUDA software stack. An interface, such as the CUDA C API, OpenCL, DirectCompute or CUDA Fortran provide a means by which to program the underlying hardware. Thus, provided that there is a mechanism to access one of these interfaces, any application can make use of the GPU. The underlying GPU is abstracted by the API, allowing the programmer to program at the thread block level with high level C or Fortran. (Figure derived from the CUDA Programming Guide [22])

the host and the device have separate memory, referred to as *host memory* and *device memory*, respectively.

Thread blocks and grids implicitly loop over all elements in a data set: serially, a 1D thread block would be a simple loop, a 2D thread block would be a nested loop etc. Within a thread block, threads can communicate via shared memory. Shared memory acts as an explicit low-latency cache near each processor, much like L1 cache on a conventional CPU. Furthermore, threads in the same block can be martialed by the intrinsic barrier synchronisation mechanisms provided in the CUDA API. Inter-thread-block communication is not possible, meaning that thread blocks execute independently and in no specific order. Thus, thread blocks can be tuned for occupancy or memory use. The maximum thread block size may be unusable because there are not enough resources on an SM to support the data requirements of this many threads, meaning that the block dimensions determine the grid dimensions in order to address all elements of the data set. The programmer focuses on ensuring that one thread block executes efficiently on one SM: scaling and scheduling is handled implicitly by the CUDA runtime environment. For example, the maximum number of threads a kernel may launch is 33 554 432 (512×65536), but a GTX280 only supports the 768 concurrent threads per SM on its 30 SMs, meaning that only 23040 threads are running at any one time. Similarly, a GTX260 with only 24 SMs can perform the same operation, only slower, since more thread blocks are queued per SM, as illustrated in Figure 2.4.

In practice, only a fraction of those threads are actively processing at any one time. A lightweight scheduling system can efficiently switch thread contexts/warps, ensuring that latency is hidden and the GPU is kept busy. Occupancy is defined as the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps [23] and is a measure of how well threads are able to hide the latency of the comparatively slow memory transactions on the GPU. The thread block size must be tuned to achieve an optimal balance between speed and occupancy for a kernel. While maximum occupancy is not a goal in itself, it is a valuable tuning metric, as occupancy relates to latency hiding.

2.2.2 CUDA Kernels

The CUDA kernel is the function executed in parallel by every CUDA thread on the GPU. The data operated on by a particular instance of the kernel is determined by the unique thread index and block index of that thread. Internally, a kernel is aware of its position in the hierarchy of threads and blocks, and so addresses its data based on this information.

Ideally, programmers use the basic GPU building blocks: map, reduction, scatter, gather, sort or search to implement their algorithms. The reason a programmer uses these parallel primitives is that they provide a means to decompose and efficiently map complex algorithms to the GPU [80].

2.2.3 CUDA Memory

Complementing the threading model, CUDA's memory model offers a variety of memory types. The CUDA threading model encapsulates data parallelism, while the memory model exposes, and minimises, the performance bottlenecks of data dependencies. Thus, use of suitable memory structures and techniques is critical for application performance. CUDA threads have access to a variety of memory spaces throughout their execution, (Table 2.1). Each thread can access its own private local memory during execution as well as a pool of memory shared between all the threads in its block, for as long as that block is running on an SM. At a global scope, all threads from all blocks have access to global, constant and texture memory. From the host, data can be written to global, texture and constant memory spaces for later use by kernels. These memory spaces are declared on the host and are persistent for the entire duration of the CUDA runtime associated with the host thread and are managed in the same way as host memory (using CUDA memory management functions analogous to C's *malloc*, *memcpy* and *free* functions).

Memory optimisation is the most performance-determining factor in CUDA [23]. Put concisely, the goal is to maximise the hardware utilisation by by maximising bandwidth utilisation by using as much fast memory (register, shared) and as little slow-access memory (global, constant, texture) as possible. Even slow-memory access is preferable to memory transfer across the PCI-E bus. GPUs possess in excess of 80GB/s bandwidth to DRAM and CPUs in the order of 10GB/s to 50GB/s bandwidth to RAM, depending on the architecture. Gen2 PCI Express buses can transfer at most 8GB/s between the host and device, necessitating that programmers minimize data transfer between the host and the device, even if that means running kernels on the GPU that do not demonstrate any speed-up compared with running them on the host CPU.

Table 2.1: ***CUDA Memory Spaces.*** *There are six different types of memory space available on the GPU. These range from small fast registers and shared memory to large volumes of high latency constant, global and texture memory. Different types of memory posses different scopes, writability, caching characteristics and lifetimes depending on their intended purpose.*

Type	**Location**	**Cached**	**Access**	**Scope**	**Lifetime**
Register	On Chip		r/w	Thread	Thread
Local	Off Chip	No	r/w	Thread	Thread
Shared	On Chip		r/w	Block	Block
Global	Off Chip	No	r/w	Global	Application
Constant	Off Chip	Yes	r	Global	Application
Texture	Off Chip	Yes	r	Global	Application

The primary function of cache is to reduce the latency of retrieving data from larger, and higher latency, system memory. CPUs are designed for general computation and, consequently, require large caches per core to maximise single thread performance. In contrast, GPUs are not intended for truly general computing and so the cache is designed to suit highly separable data, explicitly managing small subsets of this data in tiny local caches. The management of these explicit caches and selective use of singly cached variables in constant memory and the arrangement of memory to afford maximum performance is central to the CUDA memory model.

Each type of memory on the GPU serves a specific purpose (Table 2.1). Registers are fast and store local variables during the execution of the thread. Registers effectively have zero latency when used optimally. However, bank conflicts and read-after-write dependencies can increase the latency. For example, a read-after-write dependency, requires a thread to wait 24 cycles after writing to a register before it can be used again. Fortunately, multi-threading completely hides this latency if at least 6 warps are active on an SM, because each warp is issued an instruction once every 4 cycles. This occurs on any device when occupancy is greater than 18.75% [23] which is a motivation to increase kernel occupancy. SMs from G80, GT200 and Fermi GPUs contain 8192, 16384 and 32768 32-bit registers respectively which are shared between the number of threads in a block. Unfortunately, if a block contains more data in local variables than can be contained in registers, these values "spill" to global memory residing off-chip, resulting in a 2 orders of magnitude read/write latency over registers.

Shared memory is physically identical to registers in that it is partitioned in to 16 distinct 32-bit memory banks, enabling a half-warp of 16 threads to read or write 16 values concurrently in one instruction cycle. Like registers, shared memory has a 24 cycle wait when resolving read-after-write dependencies [23]. The G80 and GT200 architectures have 16384 bytes of shared memory per SM, but Fermi GPUs have a combination of L1 cache and shared memory configurable in a 16/48 or 48/16 split, depending on the applications needs [60]. The access latency of appropriately managed shared memory is approximately 100 times less than that of accessing the same data from global memory [23].

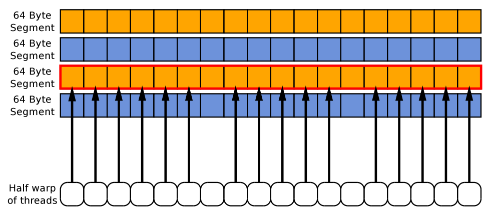
Global memory is the GPUs equivalent of system RAM. Physically located far from the GPU core, global memory provides the programmer with anywhere from 512MB to 6GB in which to store data required for simulations.

Streaming processors also have a constant cache, which is used to cache single read only variables from constant memory for reuse across a thread block. First read performance of constant and global memory is identical, incurring latency hundreds of clock cycles, but, once cached, constant memory provides register like performance if the cached value is broadcast to all threads in a warp.

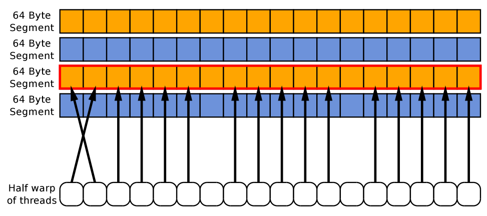
Finally, texture memory is cached in the texture processing units, not the SM, sharing the 24KB of texture memory between the SMs. An advantage of using texture memory over constant memory is that texture memory performs a spatially local cache of data near the accessed element, as opposed to a caching a single element, making it more like a CPU cache. Texture caches cache spatially in one, two and three dimensions. Therefore, if subsequent accesses are spatially local to the initial value, threads can benefit from the texture cache. This is particularly useful if the memory reads do not follow the access patterns required for good global or constant memory performance. Another important function of texture memory is the ability to perform linear interpolation in hardware. When accessed using normalised coordinates, a texture will automatically compute an interpolated value for the requested coordinates, facilitating the pre-computation and discrete storage of expensive functions that can be used as constants for computation on the GPU.

Possibly most important performance consideration in programming for the CUDA architecture is the coalescing of global memory accesses [22, 23]. If data is arranged and required according to specific access patterns, all threads in a warp can load or store a 64-bit word in global memory in as little as one coalesced transaction. Global memory is partitioned into 64 byte segments, thus a single segment can contain 16 floats. GPU hardware accesses these segments in transactions of 32, 64 or 128 bytes. Thus, to ensure optimal data transfer from global to shared memory or registers, a single 128 byte transaction can service every memory request for a warp. Memory transactions are performed at the half warp level, thus peak bandwidth is attained when the k^{th} thread in a half warp accesses the k^{th} word in a segment aligned to 16 times the size of the elements being accessed.

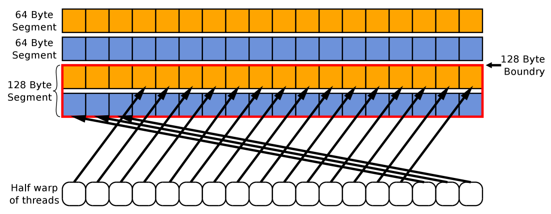
Figure 2.7 illustrates the how various arrangements of memory affect coalescing with Figure 2.7a showing the simple case of 16 threads accessing 16 floats aligned correctly. Note that it is not necessary for threads to access an element in order to maintain the coalesced behaviour. In compute capability 1.2 or later devices, a single transaction will occur for any permutation of



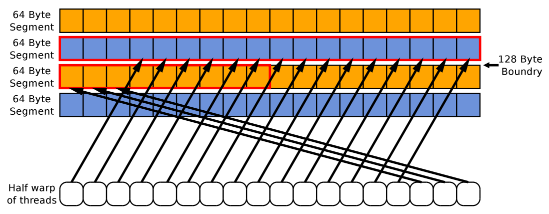
(a) Coalesced Access



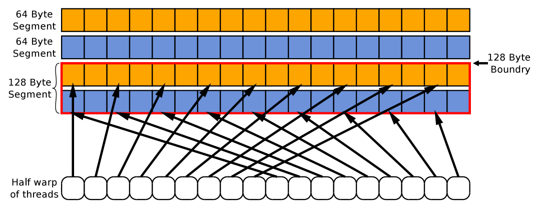
(b) Out of Order Access



(c) Sequential Misaligned Access



(d) Sequential Misaligned Access, Multiple Transactions



(e) Strided Access

Figure 2.7: Global Memory Coalescing
Coalesced global access are vital for achieving high effective memory bandwidth. Global memory is partitioned into 32, 64 and 128 byte addressable segments. Depending on the transaction, the GPU will attempt to use as few transactions as possible to service the request. Early CUDA architectures (≤ 1.1) required a 64 byte aligned serial access pattern (a) in order to perform a coalesced read or write, otherwise (b) to (e) all 16 transactions for a half warp would be performed as 16 serial transactions. Subsequent architectures relax coalescing conditions such that access patterns (b) to (e) are handled more favourably. Provided all addressed fall within the same 128-byte segment, even if they overlap a 64-byte boundary, a single transaction occurs. If a transaction crosses a 128-byte boundary (d), the transaction is decomposed into as few 32-,64- or 128-byte transactions as possible. (Figure inspired from Figures 3.3-3.6 and 3.8 of the CUDA Best Practices Guide 2.3 [23])

access pattern within the same segment (2.7b). However, compute compatible 1.1 devices would resolve this issue using 16 serial transactions, severely diminishing performance. The effect of some threads failing to participate only impacts *effective bandwidth*, the ratio of bytes requested over bytes transferred; kernel bandwidth is effected by the number of transactions and the size of these transactions.

Access patterns such as Figure 2.7c and 2.7d occur when threads access sequential data at an offset address than does not fall upon a 64KB boundary. Critically, if the threads in the half warp access elements within the same 128 byte segment, (Figure 2.7c), a single transaction occurs, wasting 50% of the available bandwidth. But, if these addresses cross a 128 byte boundary, the request is decomposed into 2 transactions to facilitate the transfer. Figure 2.7d illustrates a half warp's request for 16 4-byte elements crossing a 128-byte boundary, the request is serviced by a 64-byte followed by a 32-byte transaction.

Finally, if warps access elements in a strided manner, but all elements fall within the same boundary, a coalesced access such as the one in Figure 2.7e occurs for GT200 architectures. However, only 50% of the data transferred during each transaction is used, resulting in poor effective bandwidth. As the stride increases, effective bandwidth degenerates until the GPU has to perform one 32-byte transaction per element, resulting in serialisation of the request and over 16 times poorer performance. The relaxed coalescing criteria on compute capability 1.2 or higher devices means that any access that fits into 32 bytes for 8-bit words, 64 bytes for 16-bit words, or 128 bytes for 32- and 64-bit words is coalesced. Furthermore, provided programmers use arrays of the built in CUDA types and vector types such as *float, float2, float3, float4*, contiguous arrays of such types guarantee coalesced global memory access.

To achieve high memory bandwidth for concurrent accesses, shared memory is divided into 16 equally sized memory modules, called banks. Any memory load or store of 16 addresses that span 16 distinct memory banks can be serviced simultaneously, resulting in effective bandwidth 16 times that of a single memory module. A single is 4 bytes wide, meaning that no bank conflicts occur for contiguous arrays of types such as *int* and *float*. However, types such as *double2* or *float4* each require 16 bytes and thus occupy 4 banks per element. So a half warp of 16 of these types can only write 4 elements simultaneously to shared memory, resulting in a 4 way bank conflict. The repercussions of such conflicts is that the latency of read-after-write dependencies increases by the factor of the number bank conflicts. Thus, with no bank conflicts, 192 threads is sufficient to completely hide latency on an SM, but, with 4 way bank conflicts, 768 threads per SM are required to hide the resulting latency.

2.2.4 Limitations of the CUDA Architecture

An issue that has always hampered GPGPU is floating point accuracy and IEEE compliance. GPUs are designed to favour speed over accuracy, because their intended use is in rendering images, which are not particularly sensitive to these inaccuracies. The same does not hold for scientific simulations. This, coupled with the performance penalty of performing double precision calculations (8 times slower on a GT200, emulated on G80 hardware) has limited performance in general purpose applications. Floating point inaccuracy is also an issue in CPU implementations, but it is often ignored because its effects are less noticeable on an IEEE compliant CPU. Techniques exist to compensate for the lack of precision on both CPU and GPU. For example, Kahan summation to reduce truncation errors [81] or arbitrary-precision arithmetic. Should the

GPU require such methods to produce the correct result, its likely that the CPU algorithm will too. For single precision, CUDA implementations of special functions are generally accurate to 3 units of least precision (ULP), but certain functions, particularly power, gamma and Gauss error functions, range in ULP error from 4 to 11. Double precision implementations of the same functions sees smaller ULP errors in most cases [22].

Double precision was added to CUDA with the GT200 containing a double precision FMA unit in hardware. This precision comes at a cost 8$\times$ slower than single precision because threads cannot use parallelism of the single precision SPs for this arithmetic. Fermi improves this figure with only a 50% reduction relative to single precision performance [60].

2.2.5 Asynchronous Heterogeneous Computing

To further increase the parallelism of the host-device system, memory transfers and kernel execution can use either synchronous or asynchronous call semantics. In the simplest use case, a memory copy or kernel invocation will behave like a standard function call, passing program control from the host to the runtime and waiting for control to be returned upon completion of the operation. However, the CPU, memory bus and GPU can all function independently and concurrently. CUDA provides *streams*, asynchronous job queues, for this reason. Currently the GPU runtime supports 16 independent streams which are in themselves, in order queues of CUDA memory operations and kernel invocations.

By using streams, it is possible to further decompose problems in concurrent CPU and GPU tasks, executing both tasks in parallel. A relevant example of this is parallel tempering simulations. These simulations perform multiple instances of the same simulation with differing parameters concurrently, synchronising at a global level. Using a GPU asynchronously means that such a simulation can occupy all the cores of the CPU with global control, synchronisation and serial tasks ill suited to the GPU while the highly parallel parts of the simulations execute on the GPU. This usage model aims to utilise all of a systems resources concurrently, as opposed to one at a time in the synchronous case. *Fermi* architecture improves on this model allowing concurrent kernel execution on the GPU, G80 and GT200 GPUs scheduled kernels in a serial manner.

Streams provide a mechanism to divide algorithms into parallel tasks on both the GPU and CPU. Thus, algorithms can be efficiently divided into data parallel operations that run on the GPU, while concurrently, less data parallel portions of the algorithm are performed on the CPU. Thus, both CPU and GPU can be used simultaneously, maximising the resource utilisation of a process as opposed to using either the GPU or CPU in a mutually exclusive synchronous manner.

2.3 Summary: Optimisations in GPU Computing

Memory optimisations specific to the GPU are critical to effective GPU utilisation. Minimising the host-device bottleneck means that less time is wasted in transferring data between the host and device. nVIDIA even recommends using kernels that are of no performance benefit if they avoid data transfer. A more fine-grained level of memory optimisation is the GPU-DRAM bottleneck which, although wider and faster than the PCI-E bus, is still comparatively slow in providing the GPU with sufficient data to fully utilise the compute power of all the cores. Thus, maximising the effective bandwidth of an application and coalescing global memory is accesses is essential in minimising the degree to which a process is memory-bound. The finest level of granularity is minimising the use of global memory though explicit caching via shared memory. Replacing as many global accesses as possible with shared memory accesses, decreases the total latency of a thread's memory accesses.

Finally, at the warp level, it is more important to avoid thread divergence than optimise the instructions of in a thread. CUDA programming primarily relies on data-parallelism and single thread performance, the resultant high-level memory and thread optimisation generally yields more speed-up than per-thread instruction-level optimisation.

Thus, the kernel, memory and thread models give rise to a multi-dimensional parameter optimisation problem for a particular implementation. From a memory perspective, a choice of global, constant, texture or shared memory must be made in designing a kernel. Unfortunately, all of these resources, apart from global memory, are highly scarce, meaning that problems must be partitioned into smaller sub-problems that fit into the available memory space. This dictates the thread block size available to the programmer since shared memory, registers, local memory and texture caches are shared between all the threads in a block determining that parameter sets occupancy. In turn, altering the block size due to algorithmic reasons imposes a further optimisation parameter. At a higher level, the choice of asynchronous or synchronous calls and the number of streams on the GPU further increase the optimisation parameter set.

A successful CUDA implementation is mindful of all of these factors. In the generic case, programming for the average case is sensible and, via the templates in C++, a degree of dynamic runtime optimisation is possible. Furthermore, the loadable module support of the CUDA driver API allows for dynamic generation and loading of kernels if required [22]. But ultimately, hand-tuning a kernel and CUDA configuration based on the semantics of the data is likely to result is best performance, though at the cost of increased development time.

Conceptually, the GPU hardware characteristics can be phrased as a set of high-level optimisation strategies. However, algorithmic and method design are far more important than any other GPU optimisations [23, 82]. Discovering the manner in which serial code can be made parallel is the most important task of all; maximising the ratio of parallel to sequential code provides the most rewards in any HPC context, not specifically GPU computing.

2.3.1 Expected GPU Performance

GPU implementations typically achieve speed-ups of 10-100 times the CPU version [13]. nVIDIA markets the benefits arising from its GPU technology with the phrase, “reduce time-to-discovery” [6]. For example, a speed-up of 50× reduces a simulation from 1 year runtime to 1 week, decreasing the waiting time when simulating highly complex systems.

GPUs excel when applied to problems of high computational intensity, with a high ratio of calculations to memory accesses. The classical n-body simulation is well suited to the GPU because direct N-body methods involve an input vector of N positions and N velocities. The force of each element acting upon every other element is calculated and a double integral performed to attain the updated time-step velocities and positions. The high arithmetic intensity comes from each pairwise force calculation requiring 20 floating point operations per pair of input elements [15, 16]. This is in contrast to applications such as reduction, which performs only one arithmetic operation per pair of input elements, leading to a memory bound algorithm [82].

The nVIDIA reference N-body implementation by achieves 204 GigaFLOPS [15] on the then current G80 GPU. The GTX280 used in our work achieves 311 GFLOPS for the same parameters, which is similar to Belleman et al. [16] also implement the n-body problem using CUDA and a 4th order Hermite integrator, instead of the Verlet integration scheme used by Harris, achieving up to 230 GFLOPS on a 8800GTX (G80). Importantly, Belleman shows that a G80 GPU outperforms a GRAPE-6Af accelerator designed specifically for this application for a much higher price tag.

Friedrichs et al. use both a Lennard Jones and Coulomb potential to perform molecular dynamics on the GPU and achieve up to 212 GFLOPS performance for a 5078 atom simulation, a 735 times speed-up over a CPU [17]. Other molecular dynamics [18,25,83] and physics [16,26] simulations benefit from speed-ups of up to two orders of magnitude over a CPU implementation.

Unfortunately, the ability to extract these speed-ups requires explicit knowledge of the underlying architecture and programming model, tuning code and algorithms for maximum data parallelism, techniques often ignored, but of great benefit to CPU programs, especially in an era when hardware is becoming wider rather than faster.

[6]http://www.nvidia.com/object/gpu_tech_conf_research_summit.html

Chapter 3

Protein-Protein Docking Simulations

3.1 Introduction

The problem of protein-protein binding, is described as one of the 10 most sought-after solutions in bioinformatics [1]. Protein-protein or protein-ligand docking interactions play a central role in biochemistry, since the formation of complexes is integral to biological function [84,85]. While proteins-protein interactions are highly important, there are only a relatively small number of structures experimentally determined at an atomic resolution. The use of computational methods is therefore most important as tools to complement these techniques [12]. Predicting such interactions is as important as being able to predict the native structure that enables them to bind in this way.

Proteins are composed of chains of amino acids. This chain is linked by bonds between the carboxyl and amino groups of adjoining amino acids (Figure 3.1). Each amino acid of the protein chain is referred to as a *residue*, while the covalent bond between them is referred to as a *peptide* bond and the whole chain as a *polypeptide.* The various amino acids differ only in the compositions of their side chains. Of the approximately 300 types of amino acids found in nature, only 20 occur in proteins.

The peptide bonds form the backbone of a protein, exposing the side chains. The side chains give each each particular amino acid and particular protein its structure and function. Polypeptides (proteins) are strictly linear chains with no branching of the peptide linkages.

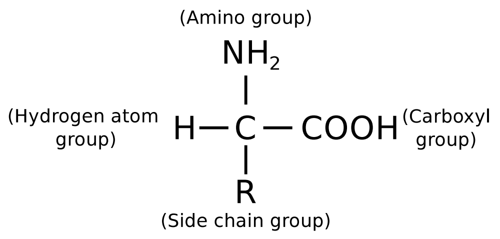
In this chapter, we review methods for protein-protein docking. This is followed by a review of algorithmic modifications which may be used in the implementation of the generic docking techniques. These algorithmic changes arise from the computational demands of the protein models and can therefore be considered as separate from the methods themselves.

The ultimate aim of a protein-protein docking simulation is to determine the molecular structure of a complex formed from two or more proteins in the absence of experimental data.

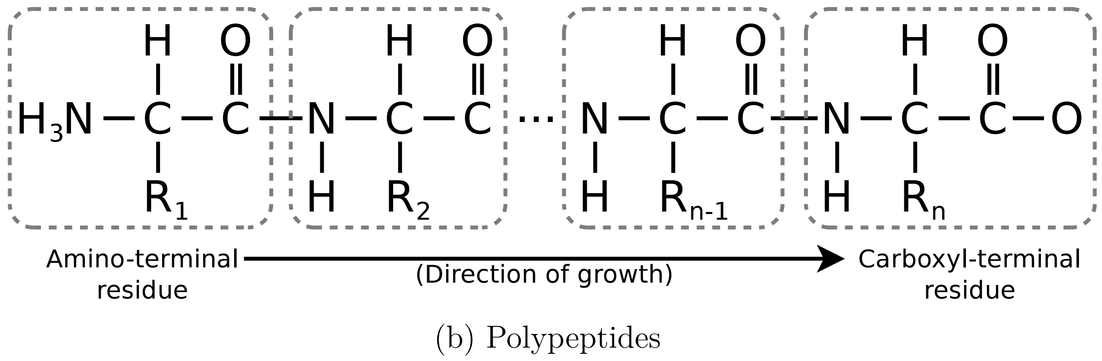
Consequently, docking methods have to generate configurations which contain a near-native docking configuration (the configuration similar to the naturally occurring structure) and have to be able to distinguish these configurations from other, less favourable, configurations.

Protein-protein complexes form due to the interactions of the residues in the participating proteins. Complex formation is driven by the same interactions that cause the proteins to fold, thus, the physical principles governing folding and docking are similar [86]. Complex formation can be viewed from either a physical or empirical perspective. Physically, the folding of docking process is a energy minimisation problem, while empirically, it is a structural motif (a three-dimensional structural element) matching problem. The empirical and physical interpretations of the problem give rise to two broad approaches to docking. Physical techniques use the electrostatic forces between proteins to perform simulations that minimise the energy of the system and empirical techniques perform docking via complementarity. A 3D representation of two proteins and their docked state is included in Figure 3.2, illustrating the implicit structural motif aspect to the docking problem.

The whether electrostatic or structural, the search space traversal posed by docking is analogous to traversing the folding funnel from protein folding [88]. The concept of folding funnels revolutionized the understanding of protein folding. Most importantly, the stipulation that protein folding progresses via multiple routes going downhill rather than via single folding tra-



(a) Amino Acids


$H_3N-C-C-N-C-C \cdots N-C-C-N-C-C-O$
H O H O H O H O
R_1 H R_2 H R_{n-1} H R_n
Amino-terminal residue
(Direction of growth)
Carboxyl-terminal residue

(b) Polypeptides

Figure 3.1: Amino Acids and Polypeptide Growth

Polypeptides are synthesised from amino acids. (a) Each amino acid contains one central carbon atom and 4 subgroups. (b) Polypeptides form from covalent peptide bonds between the carboxyl and amino groups of amino acids, growing from an initial sulphur containing amino acid in the amino terminal group.

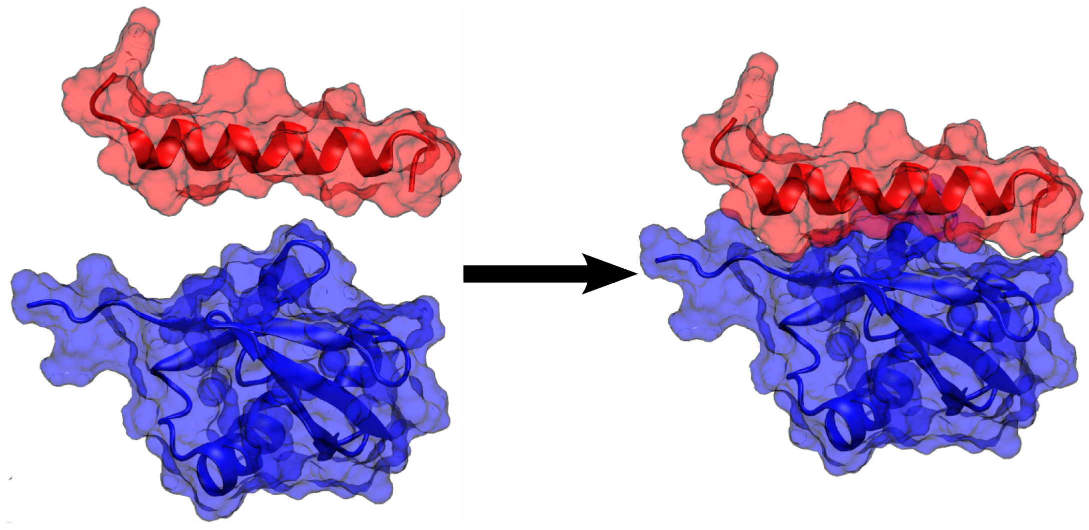


Figure 3.2: A Docking Schematic
Protein-protein docking is the process whereby proteins form a complex. This is either an energy minimisation problem or structural motif problem. Energetic or scoring functions applied to undocked components using a suitable search method provide the means to predict the docked structure. Docking specifically searches for these docked complexes, whereas binding simulations pertain to the manner in which the complex forms. [This image and subsequent molecular visualisation graphics are generated using VMD [87]]

jectory has shown a way out of the Levinthal paradox [88]. By using a funnel shape (Figure 3.3 to describe the protein folding energy landscape as a function of conformational space, protein folding is not a random search as it is driven to find the global minimum of the funnel. Depending on the folding model, a conformation can become trapped in the potential wells on the well's surface. What is also evident is that the higher these barriers, the longer it will take the process to reach the native conformation at the bottom of the funnel [88]. Tsai et al. state that the funnel of a protein complex can be expected to be rugged if the individual folding funnels of the participating proteins are rugged [88]. In docking rigid proteins, there is likely to be a single or few global minima due to the likelihood that the docking site will be very specific. However, flexible proteins have rugged funnel bottoms, because they can occupy a range of conformational isomers and with low energy barriers separating them [88].

The computational approaches to direct modelling of the physical interactions of proteins are classed as either binding or docking simulations. The distinction between these simulations is that docking simulations discover the configuration in which proteins form a complex and the equilibrium conditions surrounding this formation, whereas binding simulations model binding pathways and the kinetics of protein binding to study complex interactions [86]. Thus, binding and docking simulations provide complementary methods by which to model and understand protein interactions. Besides protein-protein interactions, proteins also interact with ligands, DNA and other bio-polymers. While these interactions are driven by the same physics and principles, the docking strategies and methods differ. In ligand-protein interactions, the ligand is much smaller than the protein and the binding site on the receptor (the protein in the in-

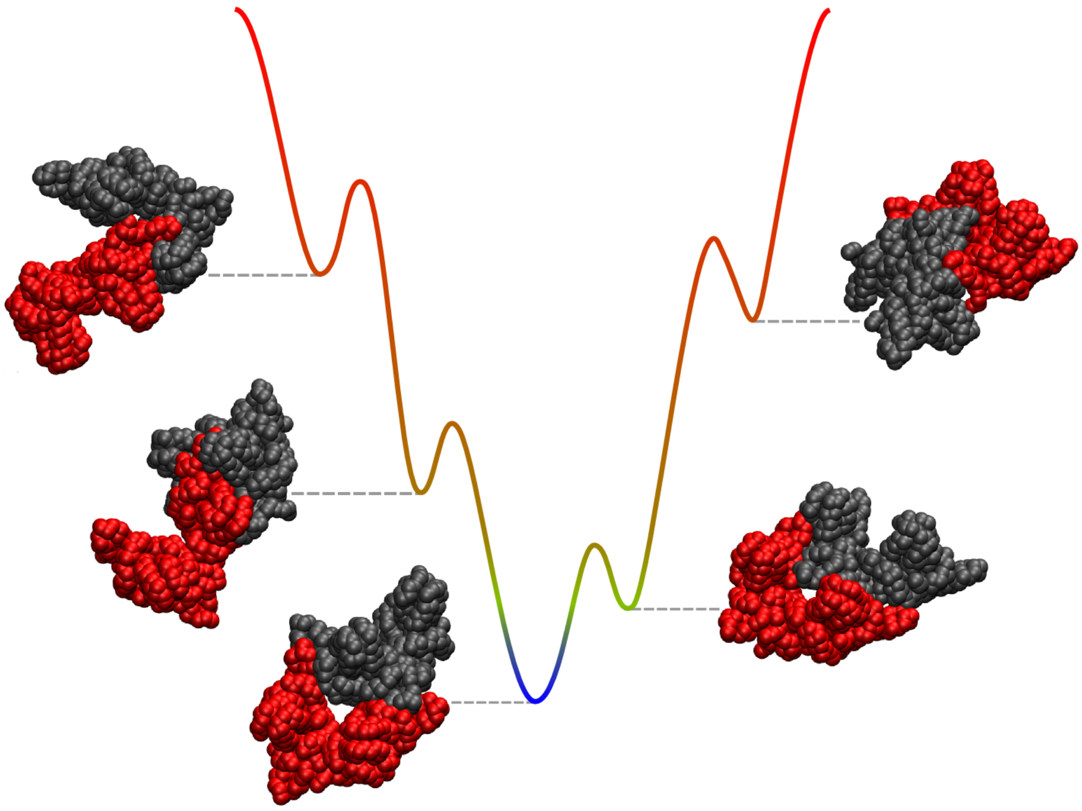


Figure 3.3: The Protein Docking Funnel
The docking problem is analogous to the folding problem in that both must locate a global minimum (bottom of the funnel) representing either the native state in the case of folding or the docked configuration. Local minima exist in the funnel, inducing local energy barriers which can trap the docking process. Depicted here are pieces of a viral capsid, the correctly docked configuration occupies the lowest energy state at the bottom of the funnel, with mis-docked configurations occupying potential wells of higher energies.

teraction) is either known or presumed: the purpose of such simulations is to determine the finer details of the interaction. Conversely, in DNA-protein docking, DNA is far more flexible and there is little in the way of structural recognition beyond the local DNA sequence [86]. Ultimately, protein-ligand and protein-protein simulations share the same principles, algorithms and procedures.

Docking simulations can be classified into rigid and flexible docking. For rigid docking, proteins are treated as rigid structures that interact without any internal change throughout the docking process. Thus, rigid body docking occupies a six dimensional conformational space consisting of relative translational vectors and rotational angles. Flexible docking includes the stretching, twisting and bending of the atomic bonds within each protein and consequently allows the protein to change shape during the docking process. This introduces an enormous number of degrees of freedom to the docking process and, in turn, is computationally infeasible from an all atom modelling perspective [89].

Flexibility can be introduced in several ways to docking. Implicitly, flexibility is intro-

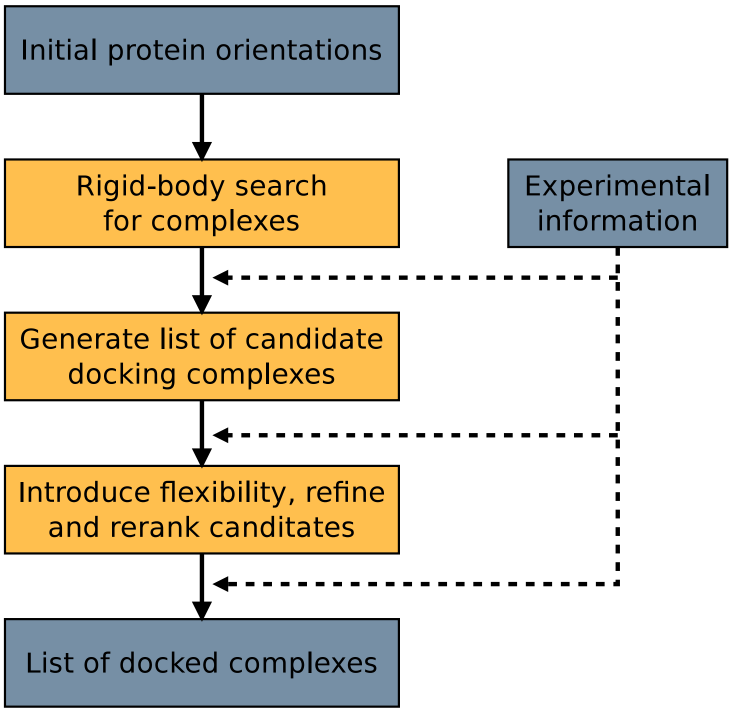

Initial protein orientations
Rigid-body search for complexes
Experimental information
Generate list of candidate docking complexes
Introduce flexibility, refine and rerank canditates
List of docked complexes

Figure 3.4: The Stages of Docking
Docking applications often adopt a multi-stage process, initially performing rigid body search for suitable docking candidates before refining the structures with flexible docking simulations. At each stage of the process, experimentally attained information can be used to guide the refinement and selection of the docking candidates.

duced by smoothing the protein surfaces or allowing some degree of interpenetration of residues (referred to as *soft docking*) or by performing multiple docking simulations from different conformations (*cross* or *ensemble docking*) [90]. Explicitly, flexibility can be modelled with side-chain and/or backbone flexibility. When proteins are treated as rigid bodies, docking is a self contained problem, but, as soon as flexibility of the protein backbone and links between secondary structures is incorporated into a docking model, the model simulates docking as well as the changes to tertiary structure, a topic that falls in the folding domain [90].

Accounting for protein flexibility is challenging due to the degrees of freedom in such systems. Consequently, most simulations treat proteins as rigid bodies or allow for flexibility of a fixed side chain structure [91]. In the cases where significant conformational changes occur during docking, flexibility is necessary, but, in cases where only small changes occur during docking, the trade off between the extra time required for flexible docking and its results is less distinct [91, 92]. Solernou et al. reports that a flexible backbone and side chain docking minimisation scheme based on the UNRES force field performs does little to improve the result attained from FTDock using rigid body docking [92].

Therefore, docking studies follow a multiple part process, depicted in Figure 3.4, performing a broad first-pass search on the global search space, before a more fine-grained search starting from the best candidate docking sites. For docking, this involves a rigid docking simulation followed by a flexible docking simulation to refine the docking structure. An example is rigid body docking using ZDOCK and FFT methods followed by optimisation of the structure by introducing flexibility with RDOCK [93].

Following the rigid body docking step, a stochastic method, such as a Monte-Carlo search, will produce a large number of candidate solutions. In the case of multiple samples, a clustering step is performed to extract representative structures for each set of similar poses. This is done using similarity metrics or screening by the free energy of the poses. The choice of screening method varies greatly by technique and ultimately depends on the purpose of the study. A blind docking simulation should indicate the theoretical docking site of the participants, whereas in screening studies the search may be for a specific binding site.

Ultimately, a question arising from any docking study regards the quality of the result. In bound docking studies this is comparatively easy, as many of these studies seek to develop a model and method based on model data, e.g. existing PDB complex entries. A metric such as RMSD (root mean square deviation) can evaluate the differences between the model (experimental) docking solution and the simulated structures. RMSD is the square root of the average sum of squares distances between the between the atoms of the experimental and simulated protein, making it a single precise measure for determining structural similarity.

However, in a blind docking study with no reference experimental structure, evaluation by RMSD is impossible. The CAPRI (Critical Assessment of Predicted Interactions) experiment was developed in response to this problem and provides blind docking challenges to evaluate the efficacy of docking simulations. CAPRI has shown that easy docking problems, ones with little backbone conformational changes, are generally well handled by the modelling community. However, simulations of even small conformational changes during a docking simulations are extremely challenging [90].

Whether rigid or flexible, two key parts to the docking procedure are an energy or scoring function that can correctly determine the difference between bound and unbound complexes and a search algorithm to find the orientation (or pose) of the molecules for evaluation by the scoring or energy function.

3.2 Scoring Functions and Energy Potentials

The nature of the scoring or energy function used for docking will determine the techniques available for docking. When approached geometrically, the scoring function serves as a measure of the geometric similarity between the interfaces of the participating proteins, while physical approaches evaluate quantities such the energy of the system caused by the electrostatic forces between the atoms in each protein.

3.2.1 Electrostatic Potentials

Protein docking can be likened to determining the minimum energy state in an intermolecular energy landscape [86]. This energy landscape resembles a rugged funnel [94] or potential well and the understanding of the topology of this funnel has had a significant impact on the understanding of both folding and docking as it accounts for the kinetic behaviours of both self-interacting protein in folding and protein-protein interactions in docking [94]. As a scoring function, the electrostatic potential of a system provides a means for traversing the docking funnel explicitly using a search algorithm or implicitly, as in the case of molecular dynamics, using the resultant force from each electrostatic iteration to accelerate each atom of a simulation in accordance with Newtonian physics.

Typically, energy functions are derived from the non-bonded forces between proteins and forces between the atomic bonds within a protein. Non-bonded forces are commonly composed of a long range electrostatic component derived from the integral of Coulomb's law, F_{ij}, caused by the point charges of every pair of atoms, q_i, distance r apart,

$$F_{ij} = k\frac{q_i q_j}{r^2}$$

and a short range force constructed from van der Waals force and Pauli repulsion. The short range force is usually modelled using the Lennard-Jones potential,

$$V(r) = 4\epsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right]$$

which approximates the effects of the van der Waals force using r^{-6} and Pauli repulsion due to overlapping electron orbitals using r^{-12} causing a sharp increase in potential when the orbitals overlap. [95].

Bonded forces include bond stretching, dihedral torsion, bond angles. All of these forces occur within the structure of a protein. Each covalent bond will have a minimum energy configuration, thus, these forces provide a means to quantify energy changes from flexing proteins during docking. Bond stretching and bond angle energy calculations model each bond like a spring, using Hooke's law: $\mathbf{F} = -k\mathbf{x}$. Similarly, dihedral energies measure the torque produced by twisting the covalent bonds between atoms and obey the angular form of Hooke's law, $\tau = -\kappa\theta$. Integration of the forces generated by the stretches, bending and twisting calculates the potential stored in each bond and hence determines its contribution to the energy of the system.

The final energy function sums all the aforementioned terms for every pair of atoms in a simulation resulting in an energy function with $O(N^2)$ computational complexity. Energy functions can be simple, containing only the Lennard-Jones potential, or full-scale standard force fields such as the OPLS, AMBER, CHARMM and ECEPP force fields [86]. The choice of which force field to use is determined by the modelling approach and, since there are thousands of atoms per protein, the time required to calculate the potentials becomes prohibitively large with increasing simulation size and number and complexity of the terms in the function. Simpler force fields leverage the collection of many samples for later refinement, whereas, more complex force fields leverage the generation of fewer, more refined samples in the first stage of docking, lessening the role of the refinement stage [86].

Various techniques are employed in electrostatic summation to improve the computational efficiency of the direct summation method. Direct summation is simply the sum of each pairwise electrostatic interaction; $O(n^2)$ for the n atoms in a system. Improvements on the Brute Force approach are to perform the a summation on a three dimensional lattice (m lattice points) in $O(mn)$ [18]. Computational improvements to such summation are distance based potentials which truncate the effect of a potential beyond a set distance to zero. This optimisation further decreases computational complexity on a lattice to $O(m+n)$ [18]. Multi-level summation also reduces the complexity of direct summation complexity to $O(m+n)$. Methods such as Barnes-Hut clustering and the fast multipole method (FMM), particle-particle particle mesh (P3M) and particle-mesh Ewald (PME) fall into this category of algorithms, using a hierarchy of nested lattices. Combination of cut-off summation for short range forces and multilevel summation for long range forces results in a more efficient summation [18].

3.2.2 Scoring Functions

Empirical scoring functions are an alternative approach to the physical potential functions. Empirical scoring functions often employ geometric registration, examining the complementarity of the surface motifs in participating proteins. This method has been prominent in protein-protein methodology since the first docking algorithm for the prediction of protein-protein interactions using structural elements [96]. Most commonly, registration methods employ the correlation technique of Katchalski-Katzir et al. [97] using a fast Fourier transformation (FFT) to maximise the surface overlap between molecules to rapidly search the docking space. First, molecules are loaded and represented using a discrete grid before a discrete Fourier transform (DFT) of each representation is performed and convolved. The inverse Fourier transform (IFT) of the convolution produces a map of translation vectors and scores which correspond to the complementarity of the translation [97]. To check all possible configuration, one of the participating proteins must be rotated about each axis by a discrete increment, Δ, in a systematic manner creating a search space of $\frac{360}{\Delta}^3$.For each step of the systematic search, a new DFT, convolution and IFT must be performed, each of algorithmic complexity of $O(N^3 \ln(N^3))$ for an N^3 grid [97] making the viability of such a search dependent on the size if N and Δ. Unfortunately, smaller increments and grid cells will produce higher resolution registration, making such an algorithm highly computationally demanding. Typically, a low resolution initial search will be performed, followed by high resolution localised searches on candidate configurations to improve tractability [98]. This method easily extends to electrostatic complementarity, where the each grid cell represents the sum of atomic charges within a set radius of that cell as opposed a binary representation of volume [98].

Chemical and surface complementarity can also be used to dock proteins. Chemical complementarity scores a configuration by the chemical composition of the surface using hydrophobicity and hydrogen bonding as scoring criteria [99]. Similarly, surface complementarity maximises the similarity of the surfaces using atomic surface contacts prior to hydrogen bond length optimisation [100].

Examples of rigid body methods developed for protein-protein docking are *FTDock* [98], *ZDOCK* [101], and *Soft dock* [102]. However, these methods are too computationally expensive for flexible body docking. When used in the protein-ligand docking scenario, the search space of the problem is significantly decreased with prior knowledge of the docking site.

3.3 Systematic Searches

Computationally, the search algorithm used for docking determines the tractability of such a search. Complementarity methods require systematic searches of conformational space, since it is not possible to quantitatively evaluate the viable and non-viable docking locations using a scoring function. In the global search, evaluating every possible solution requires that 6 degrees of translational and rotational freedom be explored in addition to all the internal degrees of freedom arising from the internal flexibility of the participating proteins. For even for the most simple system, this is highly computationally intense. For example, docking a ligand to a 1000Å^3 site using discrete translations of 0.5Å^3 and rotations of 10 degrees requires the exploration of a search space comprising 3.7×10^8 configurations [103]. If sampled at the rate 10 configurations per second it would require 432 days to complete this simulation. Unfortunately, protein-protein docking requires that the entire surface of a protein be gridded in this manner, meaning that a relatively small protein such as ubiquitin with a surface area of approximately 5000Å^2 (determined using VMD from pdb:1UBQ) has a search space of 1.9×10^{10} configurations. Flexibility of the participating proteins increases the search space even further, introducing 4 additional degrees of freedom to the system (bond angles, torsion and stretching forces). This results in an intractably large search space, even for very small proteins.

Systematic searches are thus viable only for sampling either coarse (low resolution protein-protein docking) or localised (protein-ligand docking) search spaces. An example of which is EUDOC for the identification of drug interaction sites [104]. EUDOC has recently been ported to run on IBM's Blue Gene/L for the purpose of screening large volumes of chemicals for drug development. This system has screened 23426 chemicals in 7 minutes using 4096 Blue Gene/L processors, a 34 times speed-up over the 242 minutes required on a commodity computing cluster of 396 Xeon processors [105]. This search performs an initial search of 10 degree rotational increments and 1Å translations, followed by a local optimisation on candidates using 5 degree samples and 0.25Å steps using the energy difference between the bound complex and unbound complex to score configurations. In order for such searches to be tractable, EUDOC requires knowledge of the binding site in order to perform its systematic search effectively within a small enough region. In a docking simulation for two small proteins of 100 residues, each with translational and rotation freedom, the search space of 3.7×10^8 expands to 3.2×10^{197} .

Thus, more directed methods are required for sampling the large search space posed by the docking problem.

3.4 Genetic Algorithms

A genetic algorithm (GA) mimics the process of evolution by manipulating a collection of chromosomes. In the case of molecular docking, a chromosome encodes the variables that describe the state of the proteins. Each chromosome encodes a possible solution to the docking problem and is assigned a fitness score based on a fitness function, e.g, electrostatic potential, while a genetic algorithm is the used to explore the confirmation space of the chromosomes [106].

In the GA, each state variable corresponds to a gene. Random pairs of chromosomes are mated using a process of crossover, in which new chromosomes inherit genes from either parent. In addition, some of the new chromosomes undergo random mutation in some genes. Once produced, each chromosome is evaluated using the fitness function and either retained or discarded, in so doing the algorithm allows bad solutions to die of while good solutions survive [107].

Genetic algorithms are particularly good at global search problems where the degrees of freedom in a system results in a combinatorial explosion [107]. Generic genetic algorithms [108,109] and evolutionary programming methods [110] have proved successful in drug design and docking. Packages such a AutoDock and Genetic Optimisation for Ligand Docking (GOLD) [106] are examples of GA implementations for flexible docking. AutoDock 3.0 onward implements a both genetic algorithm and Lamarckian genetic algorithm searches and shows that these algorithms reliably reproduce known crystallographically obtained structures [107]. Similarly, GOLD performs automated docking with ligand flexibility, and partial protein flexibility in the neighbourhood of the protein binding site [106]. Unfortunately, the aforementioned packages are usually associated with protein-ligand docking applications although AutoDock can perform protein-protein docking [107]. AutoDock has been released under the GNU General Public License and functionality such as MPI parallelisation with near linear scaling [111] and because AutoDock uses electrostatic scoring functions such as AMBER, it benefits from GPU acceleration (GPU AutoDock)[1] in the same way as molecular dynamics and Monte-Carlo implementations.

3.5 Molecular Dynamics and Monte-Carlo Simulation

Molecular dynamics (MD) and Monte-Carlo (MC) are two global search algorithms viable for protein-protein docking simulations. Both MD and MC are based upon statistical mechanics and are applicable across a broad spectrum of simulation besides protein-protein docking [112] The knowledge of the funnel means that a biased random search, such as Monte-Carlo (or a genetic algorithm) can be directed toward low energy conformations by a scoring function. Molecular dynamics also exploits this property, assuming the forces between the atoms in a protein will guide the complex into a minimum in the funnelled landscape.

Monte-Carlo (MC) searches are similar to GA searches in that the search moves are performed using rules based on understanding of the underlying physics that represent and only subsequently evaluated. MD, by contrast, inspects the electrostatics and updates the system accordingly. Where Monte-Carlo differs from genetic algorithms is that it is a biased random search on a single simulation, employing only mutation like moves.

3.5.1 Molecular Dynamics

Molecular dynamics methods simulate the time evolution of atomic positions and velocities by integrating Newton's second law of motion,

$$\mathbf{a} = \frac{\mathbf{F}}{m}$$

The microscopic trajectories of the atoms in MD simulations are averaged using the laws of statistical mechanics to describe the system macroscopically using quantities pertaining to the system such as diffusion coefficients, phase transition temperature, potential energy and kinetic energy [112]. In classical MD simulations, forces are generated by atom-atom interactions in terms of an empirical potential consisting of both non-bonded and bonded potentials [113]. For each discrete time step, an MD simulation will calculate the result forces acting upon each atom, integrating these forces to update the velocities and positions of each atom. The choice of potential and integrator has an affect on the accuracy and computational efficiency of an MD simulation.

In as late as 2007, Weihe et al. state that the most obvious way in which to perform docking would be to simulate the molecular dynamics to allow a complex to reach its native

[1]http://sourceforge.net/projects/gpuautodock/

state, but adds that the computational power necessary for such a simulation would make it intractable [114].

Historically, the long range non-bonded interactions were ignored for computational efficiency and truncated with cut-offs. This resulted in fewer computations than the n^2 comparisons required without a cut-off radius and marked computational improvements at the cost of substantial artefacts and inaccuracy in the systems simulated [113]. Advances in computation and algorithms have subsequently addressed these problems with the aforementioned multi-level summation, fast multi-pole method (FMM), particle-particle particle mesh (P3M) and particle-mesh Ewald (PME) methods [18, 113].

The first MD simulation studied the folding kinematics of the Bovine Pancreatic Trypsine Inhibitor and simulated all of 9.2 picoseconds of the 500 atom simulation [115]. Subsequent advances in supercomputing have seen the 2006 study of a complete Satellite Tobacco Mosaic virus of over 1 million atoms to 13 nanoseconds [116]. Notably, 256 Altix nodes at the National Center for Supercomputing applications (NCSA) were only able to simulate 1.1 ns/day in the case of the viral simulation. Another noteworthy MD simulation is that of folding the Villin Headpiece run to 500 microseconds. While not a docking application, it illustrates the computational resources required to construct hundreds of 10000 atom trajectories of time-scales comparable to experimental folding times [117].

Integration Methods

MD simulations typically require millions of time steps to produce only a few picoseconds of data. This coupled with the chaotic nature of the simulations results in similar initial conditions diverging exponentially fast with no correlation between them after only a few picoseconds [118]. This divergence is the result of MD being particularly sensitive to the inaccuracy of numerical integration. Unfortunately, all numerical algorithms suffer from round-off and truncation errors. The simplest of these integrators, Euler's method, while being computationally the most efficient, has an error that grows linearly, $O(\Delta t)$ with the time-step, Δt. Thus, Euler's method is never used in MD [112]. Conversely, Verlet integration is commonly used in MD due to its error growing quadratically with the time-step. An $O(\Delta t^2)$ error means that a 10 times decrease in time step results in a 100 times decrease in error [112]. This error restricts MD to prohibitively short simulation time scales.

MD is an exceedingly popular method for protein simulations, implemented in packages such as CHARMM [27], Amber [119], GROMACS [120], LAMMPS [121] and NAMD [122]. These packages often contain multiple force field implementations, e.g, GROMACS implements AMBER, CHARMM, Coarse Grained, GROMOS and OPLS force fields for MD. The parallel nature of MD has resulted in many high performance enhancements of these codes. Freddolino's Satellite Tobacco Mosaic virus simulation is a notable example, illustrating the clustering capability of the NAMD application [116].

GPU Implementations

The separable nature of the force calculations means that MD is highly amenable to GPU acceleration, offering up to two orders of magnitude improvement over CPU counterparts. Notable GPU implementations are NAMD [18], MDGPU [26], HOOMD [25], OpenMM [17] and ACEMD [63]. Stone et. al. implement ion placement and molecular dynamics simulations with NAMD on a GPU, achieving 36.4 billion atomic evaluations per second using direct coulomb summation, a 790 times improvement over the CPU. Non-bonded force calculation in their MD simulations experience a 8.9 times speed-up using one GPU and NAMD [18]. Meel et al. implement an N^2 MD method (MDGPU) experiencing a speed-up of up to 80 times that of their CPU implementation for simulations of more than 4000 atoms. Anderson et al. (HOOMD) demonstrate that GPU implementation of a pair-list short-range potentials algorithm analogous to LAMMPS running on a single GT200 GPU performs at the same level as a 36 processor core cluster [25]. Friedrichs et al. implement both short- and long-range potentials in their code, accelerating a brute force N^2 MD method by a factors ranging from 128 for a 544 atom simulation to 735 for 5078 atoms using CUDA against a AMBER solution on a CPU [17]. AMBER has also been ported to GPU. Explicit solvent simulations of 304, 2492 and 25095 atoms on a Tesla C2050 achieve 368.2, 49.9 and 1.04 nanoseconds per day respectively, far out performing 8 CPU processors. Explicit solvent models using the same hardware configurations achieve speed-ups of approximately 4 for 23558 (20.7 ns/day on a GPU) and 90906 (5.19 ns/day on a GPU) atoms [123]. An algorithmic improvement on direct MD is to use verlet lists, and cell-lists in cell based MD simulations. Verlet lists maintain a list of all particles within a given cut-off distance, reducing the number of pair-wise comparisons and distance calculations with a periodic $O(N^2)$ sweep as opposed to $O(N^2)$ pairwise calculations every iteration, resulting in the overall complexity of less than $O(N^2)$ for N particles [124]. Cell lists provide a similar mechanism to avoid all pairs computation. Generation of the neighbour list is performed by probing for particles in neighbouring cells. Assigning particles to cells is $O(N)$ operation for the N particles and probing them is $O(M)$ for M cells [124]. Choice of Verlet or cell lists is fixed by the size of a simulation, with Verlet being more efficient for small numbers of particles and cell lists becoming more efficient in larger systems [124]. NAMD, MDGPU HOOMD and ACEMD all use cell lists.

3.5.2 Monte-Carlo Algorithms

Monte-Carlo simulation of proteins involves mutating an element of the simulation. In rigid body docking this entails rotating or translating a protein. By also including mutations that alter a bond angles, bond lengths and torsions, MC can perform flexible docking. The mutations are random in both type and, optionally, quantity, exploring exactly the same search space as a systematic search. However, the the energy function allows for biased sampling, guiding the search space to local minima.

Metropolis Monte-Carlo [125,126] is a popular Markov chain Monte-Carlo method for physical simulation [127]. Simply, this algorithm proceeds by randomly generating a new configuration from the current configuration. The potential energy of the new configuration, U_{new}, is then calculated, indicating whether or not the new configuration is better (lower) than the current configuration, with potential U_{old}. In order to adequately sample the search space, Metropolis MC accepts moves which are of higher energy than the current configuration, and so reduces the tendency of simulations to become trapped local minima in the docking funnel. Repetition of this process ultimately locates a minimal solution indicating a docking site.

As with MD, the computationally demanding aspect of MC simulations is the evaluation of the energy function. Hence, MC can benefit from the same degree of speed-up on both GPU and CPU as the same force fields are used in both simulations.

In the case of simulations of constant size, volume and temperature (NVT or canonical ensemble) the behaviour of the proteins, adhere to to the Boltzmann distribution [127]. This law states that if the energy state associated with a system is ε, the probability of this state occurring is proportional to $\exp(-\varepsilon/k_bT)$ where T is the absolute temperature and k_B is the Boltzmann constant [127]. This simplified description arises from the more complete discrete form,

$$P_i = \frac{\exp(-E_i/k_BT)}{\sum_z \exp(-E_z/k_BT)}$$

which describes the probability of a state occurring P_i as a function of all possible states (z in number) of the system [127]. To adhere to this distribution, simulations must allow transition between all configurations in the distribution during the sampling process. The probability of this transition can be determined from occurrence probabilities of both the configurations meaning that the probability of a move from configuration i to configuration j is,

$$P(i \to j) = \frac{\exp(-U_j/k_bT)}{\exp(-U_i/k_bT)} = \frac{-(U_j - U_i)}{k_bT}$$

if $U_j > U_i$ [127], defining the Metropolis acceptance criterion as $min\{\frac{-(U_j-U_i)}{k_bT}, 1\}$. A uniform random number is used to generate a number s in the interval [0,1] such that MC simulations accept a mutation when $s < min\{\frac{-(U_j-U_i)}{k_bT}, 1\}$ [127].

A critical aspect of any stochastic modelling process is the availability of good random numbers. In Monte-Carlo simulations, the quality of the results deteriorates when the random number lacks a sufficiently large period [128]. However, this problem is addressed by quasi-random number generators, such as the Mersenne Twister with a period of $2^{19937} - 1$ [129].

3.5.3 Methods for Enhanced Sampling

The deterministic nature of molecular dynamics limits its its ability to traverse rugged docking funnels. Molecular dynamics simulations will tend to become trapped in local minima formed by local energy barriers in the docking funnel, making it particularly sensitive to the initial conditions [103]. Similarly, Monte-Carlo simulations can become trapped in potential wells when only a single temperature is used for simulation. An approach to improving conformational sampling is to increase the temperature of the system to provide more kinetic energy to the system such that trajectories can overcome the kinetic barriers imposed by the docking funnel. Both, simulated annealing and replica exchange methods can be used for this purpose.

The equilibrium properties of the canonical ensemble (constant volume, temperature and number of particles) are dependent on the metropolis acceptance criterion, which is itself dependent on the temperature of the simulation. Raising or lowering the temperature determines the magnitude of the potential barriers that a simulation can overcome in its local search, but also results in a change in the Boltzmann distribution, resulting more high energy configurations in higher temperature simulations and more low energy configurations in low temperature simulations [130].

Replica exchange is a method that improves the the search space visited by Monte-Carlo searches and improves the reduces the likelihood of molecular dynamics trajectories becoming trapped in potential wells that single temperature simulations encounter. Parallel tempering preserves the canonical nature of the simulation my performing R parallel simulations or *replicas*, maintaining constant number, temperature and volume in a given replica, and thus, preserving the statistical dynamics of each replica [130]. Computationally, RE is expensive because R simulations are performed as opposed to just one, compounding the already expensive potential energy evaluations. In replica exchange Monte-Carlo (REMC) this additional cost is amortized by the increase in search efficiency because replicas at low temperature are able to sample more regions that they would otherwise visit in a single temperature simulation [130]. Additionally, the independent nature of each simulation in RE means that it is amenable to parallel implementation with favourable scaling characteristics [131, 132].

In a typical replica exchange simulation, each of the R replicas are simulated at different temperatures, usually such that the ratio between adjacent temperatures is constant. Each replica's simulation will proceed for a requisite number of iterations before replicas are exchanged. If there are no conditions associated with an exchange, it is performed with probability,

$$p = min\{1, \frac{P(i \rightarrow j)}{P(j \rightarrow i)}\} = min\{1, \exp[(U_i - U_j)(\frac{1}{k_B T_i} - \frac{1}{k_B T_j})]\}$$

exchanging configurations i and j from the adjacent replicas using the Metropolis acceptance criterion [130] The purpose of the metropolis criterion to ensure that when are exchanged, the ensemble statistics of each replica remain unaffected. The optimal acceptance ratio for replica exchange is known to be 23% [130].

Replica exchange MC and MD are shown to be effective for applications such the prediction of protein-membrane docking [133–135], protein folding [136–144] and protein-protein docking [12, 145, 146].

Specific examples of Monte-Carlo applied to the docking problem range from using MC to perform the entire docking simulation as in the case of Kim et al. [12] and Gray et al. [146] to using Monte-Carlo to refine docking configuration by rigid-body moves and side-chain optimization attained from FFT-based ZDOCK samples [145]. Simulations by Kim et al. and Gray et al. are both able to discover docking sites using unbound initial conditions and refine docked protein structure using flexibility with a high degree of success [12, 146]. By contrast, Lorenzen et al. specifically aim to refine structures discovered through rigid body docking by backbone flexing and side-chain optimisation [145]. The method Gray et al. describe is used by the RosettaDock docking application [146].

Simulated annealing solves the frustration problem by increasing the temperature of the simulation to afford a greater conformational search space before cooling it to allow the simulation to attain a new equilibrium. Unfortunately, simulated annealing changes the conditions of the canonical ensemble and consequently the inferences that can be made regarding the statistical dynamics of that simulation. However, simulated annealing is computationally favourable compared to replica exchange because only one instance of the simulation is performed. Applications such as AutoDock also provide MC methods for docking using simulated annealing instead of replica exchange for this reason [107].

The majority of biological applications that employ replica exchange do so with molecular dynamics methods instead of Monte-Carlo. Parallel tempered Molecular Dynamics is not physically accurate because of the temperature changes affect the canonical ensemble, meaning that there is no reason to favour the use molecular dynamics for docking [130].

3.6 Dealing with Computational Complexity: Methods to Improve Tractability

All of the aforementioned computational methods suffer from one or another form of computational intractability is the simulation they are studying contains too many atoms, e.g, all atom MD of a viral capsid [116], the resolution of the lattice is too high, e.g, multi-level summation [18] or complementarity [98], or the degrees of freedom results in an exceedingly large search space, e.g. all methods. Second to the search-space/problem size issue is the complexity of the physical model. In the case of MD and MC simulations, it is the evaluation of the electrostatic potential that dominates simulation time. Naively, it can be assumed that the conformational changes can all be performed in $O(n)$ time complexity since they are essentially constant cost geometric deformations applied to n elements as opposed to the $O(n^2)$ force calculations in MD or the $O(N^3 \ln(N^3))$ multilevel summation algorithm on an N^3 grid.

While hardware advances in the form of GPUs and IBM's BlueGene are affording researchers more computational power, ultimately this will only increase the performance of a simulation in proportion to the available hardware. Only with algorithmic optimisations and advancement will the time-scales and size of simulations become more tractable.

There are several methods to improve the computation efficiency of the aforementioned methods. These can be classified as enhancements on the aforementioned MD and MC methods with little or no change to the algorithm itself, biased typed searches which alter the manner in which the methods sample the search space or coarse-graining, which simplifies the problem.

Enhancements of MD include multiple time step methods such as RESPA (reference system propagation algorithm) methods are popular. This method results in a speed-up of two to three times faster than conventional MD [147]. By the use of Langevin dynamics, the number of time-steps required for simulation can be decreased by maintaining equilibrium of bond lengths and angles. This introduces stochastic behaviour in place of the most degrees of freedom, replacing the bonded forces and motions with random noise [148]. Alternatively, simulated annealing speeds up searches if the temperature of the simulation is made exceeding high in order to rapidly explore the global search space, but at the cost of accuracy as most force fields are valid for 300 Kelvin. Apart from the increased search space per step, this method does not actually improve the computability of MD and the method over-estimates the entropic contribution to the systems free energy [147]. The next manner of optimisation is multiple-copy dynamics. Multiple copies essentially swarm to locate local minima, and as a whole will locate a global minimum [147]. REMD and REMC can also be categorised as algorithmic enhancements because performing R parallel simulations in this manner is more efficient than R distinct simulations, such an approach has been used in Folding@home [130, 147].

A bias can also be added to the search method so that it does not become trapped in potential wells. The first of these is landscape engineering. The local elevation method adds a penalty of some sort for any previously encountered configuration [147]. Unfortunately, the major drawback of this method is maintaining the memory of moves, a problem that the *Tabu* search also faces [103]. Alternatively, principle component analysis can be used to identify local energy barriers and modify the potentials such that the the search space is sampled more favourably [147].

Self-guided molecular dynamics (SGMD) are another option in biasing the behaviour of MD. Conceptually, the motion of a system can be divided into two components: random and systematic motions. Random motions occur locally due to the atomic interactions while systematic motions are caused by the force working along a free-energy gradient [149]. Similarly, self-guided Langevin dynamics (SGLD) generates a guiding force in the direction of the systematic motion, calculated as a local average of velocities. By introducing this into the equation of motion governing each atom, Self-guided dynamics enhance time in which a protein simulation will fold [150]. This approach is similar to that of Digitally filtered molecular dynamics (DFMD) which filters atomic motions using digital signal processing, enhancing low frequency (systematic) and suppressing high frequency (random) motion [151].

Finally, coarse graining can be used to reduce the expense of a simulation by reducing the number of individual participants. Any model that operates at a level of granularity above that of individual atoms can be considered coarse grained. The algorithms for simulating coarse grained models remain essentially unchanged from those above, introducing changes in quantities such as electrostatic potential to emulate those of aggregate particles as opposed to every atom. Coarse grain models include: elastic network models, Go-like models and bead models [10].

3.6.1 Coarse-Graining

While algorithmic changes will improve the performance of a method, they seldom decrease the complexity of the underlying problem. By contrast, coarse-graining and other reduced representation approaches use simplified systems to improve the tractability of a system.

Coarse-graining can be categorised by the degree and manner of approximation. Atoms can be represented by one to many beads, simplifying the number of participants in a simulation and thereby reduce its computational demand. Another approach is to simply the models spatial representation as in the case of lattice models, which represent continuous 3D space using a lattice. Finally, models such as Elastic network or Go models can be used to steer proteins toward an native state.

In terms of spacial representation a choice can be made to model proteins using a lattice model or a continuous space model. In lattice models, the atoms or residues are confined to a discrete cubic lattice of continuous space [152]. The major drawback of a lattice model, is that it is not suited to the study of real proteins, and in some cases, proteins folded with lattice models can show some overall geometric resemblance to real proteins. However, properly designed lattice models have been shown to accurately predict the structure of a folded protein, although the local geometry of the structure is inaccurate. The dynamics of some lattice models is also shown to be similar to reduced continuous models [152]. Lattice models are almost exclusively discussed in the context of folding and in the study of mechanisms using idealised protein models. Lattice models to offer a computational advantage in their ability to model a multi-resolution system, which leads to computationally favourable techniques such as multilevel summation on the lattice.

Elastic networks (ENMs) represent a system of amino acids using a network of beads connected by springs. ENMs are an extremely simple way to parametrize a system, provided that the equilibrium state of the system is known [10]. Recent studies apply ENMs in conjunction with REMD [153] and systematic searches [154] to model flexible protein-ligand docking. An ENM provides a means of refining a low resolution structure, providing a means to reduce the size of a systematic search space [154] or fit high resolution structures to low-resolution experimental data [155].

Go-like models are similar to elastic network models in that they bias protein's dynamic behaviour to form a native structure. Go models where designed specifically for folding studies. When the folding characteristics of a protein follow a weakly rugged funnel, Go models are effective, but in more rugged instances they fail to describe the intermediate states of the folding process [10]. Although not a common approach, Go models can be used in binding simulations, an example of which studies the binding two unfolded proteins in a homodimeric complex [156].

The bias in both elastic network models and Go models toward a reference structure results in them only being weakly transferable to dynamics studies. To transfer the coarse grain bead concept of Go and ENM to dynamics simulations, the parametrisation of amino acids emerged in a variety of single- to multiple-bead models in continuous space.

One of the first examples of coarse graining to combat the immense size and degrees of freedom in protein folding dynamics appeared in 1976 when Levitt and Warshel simplified the representation of a protein to a protein to a C_α chain with a single point representation of the side chain's centroid, producing unique C_α-side chain pairs for each amino acid type [4]. This simplified model assigned the same shape model to each amino acid, parametrising it by averaging the amino acid characteristics from eight different protein and decreased the number of interaction atoms by a factor of 15. Furthermore, only torsional moves are permitted through the C_α chain, reducing the number of degrees of freedom in the protein they studied by a factor of 4. Tests showed this model to rapidly reproduce the correct folding of a small protein molecule under certain conditions [4].

Refinement of the CG model for folding appears more critical than in the docking context. With folding, the orientation and packing qualities of the side chains in the folded structure must be optimised beyond merely the electrostatics of the coarse grain [9]. Studies investigating the inter-residue contact potentials [157, 158], allow parametrisation of the pairwise interactions between residue representations, e.g, modifying the Lennard-Jones potential

$$V(r) = 4\epsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^{6} \right]$$

such that it includes a scaling factor (ϵ) to account for the differences in interaction strengths between the residues [11, 12]. Extending these models to protein docking reaps the same computational benefits as folding, with studies using coarse-graining of systematic protein-ligand docking with flexible side chains and energy minimisation [159]. A elliptical representation of the side chain in this type of representation has proven successful due to the asymmetry and directionality that elliptical side chains and together with a Blue Gene implementation, resulting in a 120 fold speed-up for a single MD trajectory. The effect of the model is that the experimental folding time is sped up by a factor of 1000, resulting in simulations of nanoseconds representing reactions that occur in the millisecond time scale experimentally [160]

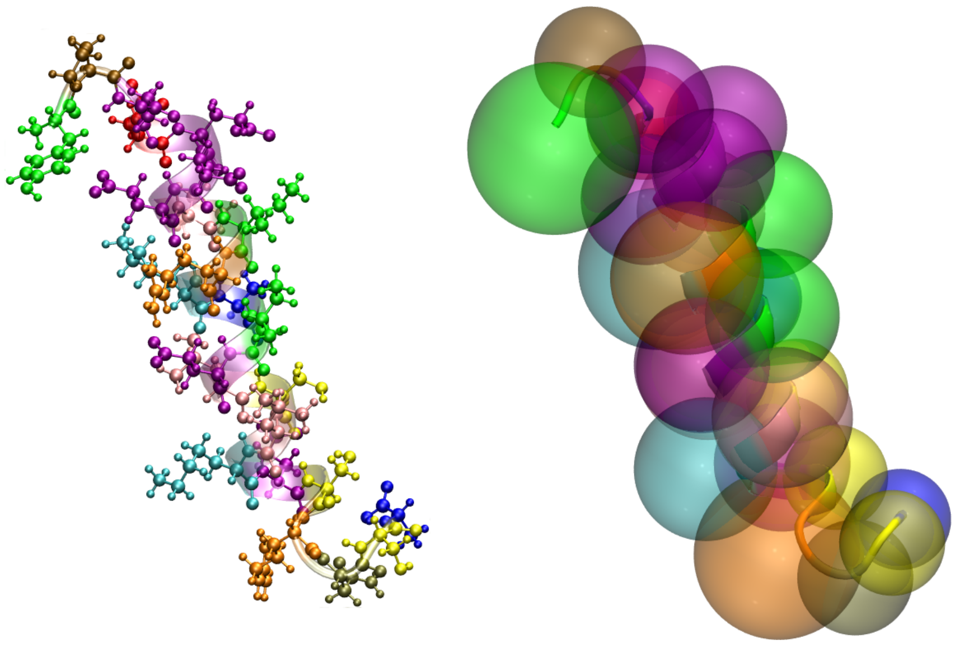


Figure 3.5: Vps27: All Atom vs Coarse Grained
(a) The all-atom representation of the Vsp27 helix. Atoms are coloured by amino acid residue overlaying the helical secondary structure. (b) The same molecule is transformed from an all atom representation to a single bead per residue, illustrating the approximation arising from the coarse-graining.

An even more coarse representation of an amino acid is to use a single bead per residue (Figure 3.5). These models are evolutions of the single bead Go models, with bias toward specific inter-bead potentials in a specific solvent. This bias arises from the equilibrium bond torsions, lengths and angles are calculated from folded proteins [10]. Kim and Hummer use Monte-Carlo docking simulations of folded proteins, representing each amino acid residue as a single bead. The model is able to correctly predict the binding interfaces of two proteins in approximately 90% of test cases and at least one binding/docking interface was produced correctly in the remaining cases, effectively simulating the thermodynamic and structural properties of complexes with relatively low binding affinities [12]. Larger simulations using the same type of coarse graining have been used to study the molecular dynamics of ligands binding, specifically an the HIV protease inhibitor [131].

Scaling up from the one and two beaded models are four and six bead models. Four bead models still represent the side chain with a single bead but retain the four atoms in the backbone chain of each amino acid [161] this and a similar model have been successfully applied to structure prediction using both MD and MC methods [10]. Models of five to six beads per amino acid have also been successful in folding small peptide sequences. Irbäck et al. developed a model that contains three types of amino acids and five or six atoms per amino to study the dynamics of secondary structure formation [162]. A similar model by Derremaux et al. has been applied using Monte-Carlo sampling for structure prediction [163].

Multi-resolution approaches using coarse graining have also been proposed, showing that representations can switch between all-atom to C_α as required without loss of information. This method allows for an all atom representation of the regions of interest, such as the binding site, while reducing detail in other areas, and in turn reducing the computational requirements. Ultimately, multilevel approaches like this will allow for smooth transition between all atom and coarse-grained models, preserving chemical details and end enhancing computational efficiency [164]. Chen et al. use a multi-resolution technique in the study of virus capsid dynamics [165].

An salient example using coarse graining for a first pass in docking before adding back the detail is RosettaDock. In this example, the side chains are represented with a single bead along with a complete backbone. Low resolution passes to identify docking sites with coarse-graining are then followed up with all atom simulations to refine the fit [166]. RosettaDock is part of the Rosetta suite of applications, designed to run on BOINC, with the Rosetta@home project using volunteer computing in the same way as Folding@home [167].

The aforementioned coarse-graining techniques can be used with Monte-Carlo dynamics, replica exchange Monte-Carlo, molecular dynamics, genetic algorithms and hybrid methods, improving their tractability by reducing search spaces by discretisation and the number of bodies in the simulation, implicitly reducing the degrees of freedom, by aggregating the atoms into beads. Coarse-grain docking shows that it is of use in both the initial and refinement stages of docking, examples of which are: rapid sampling of Monte-Carlo searching to locate native binding sites using CHARMM [12], performing Langevin dynamics in GROMACS to simulate protease inhibitor docking [131] or in the refinement of rigidly docked structures by coarse-grain side chain optimisation [92].

Combining the computational efficiency of coarse-graining and the performance of GPU computing is the subject of current research, producing results such as SOP-GPU, a coarse grained self organising polymer model based on Langevin dynamics to perform folding [168]. HOOMD-blue, the successor to HOOMD implements CG MD simulation on the GPU using the CGCMM model[2], this work was presented at nVIDIA's research summit in 2010. Currently, there is very little literature on the subject of coarse graining on the GPU in the field of docking, but a with GPU development of the major MD codes such as AMBER, CHARMM, DL_POLY, LAMMPS and AutoDock [61], it stands to reason that the coarse grain implementations in these codes will be used on the GPU.

[2]http://codeblue.umich.edu/hoomd-blue/

Chapter 4

Design

Our primary focus is on porting the coarse grained model and simulation method described by Kim and Hummer [12] (hereafter referred to as the model) from a sequential algorithm to a parallel algorithm. The key challenge is to design an implementation of the simulation that scales from one to many CPU cores and to many GPU cores.

In this chapter, we first describe our assumptions and any changes made to the model, before discussing the characteristics of our algorithm and ways of improving its performance by exploiting specific GPU hardware features. Application specific features will be discussed in Chapter 5.

4.1 Approach

We followed an iterative approach to designing and implementing our parallel algorithm enabling us to benchmark and profile of various configurations of the simulation, to ultimately converge on the optimal GPU usage model. Where possible, we aim to fully utilise all the cores of the CPU (or multiple CPUs). However, it is difficult to estimate the optimal CPU and GPU usage models. Thus, having the ability to use any configuration of GPU and CPU hardware is essential for tuning application parameters based upon profiling and benchmarking data. We therefore develop separate versions of the implementation, as follows.

Sequential CPU This is a direct implementation of the original model and method [12], programmed for the general case and hardware independent. This implementation allows the model and simulation to be checked for correctness against the original before any performance improvements are introduced. The sequential code also allows for initial profiling of the algorithm, indicating which parts of the simulation consume the most time and identifying performance bottlenecks.

Sequential GPU The evaluation of the interaction potential is performed on the GPU, with the rest of the simulation remaining unchanged. Validation of the GPU implementation is also performed to ensure errors do not persist as the complexity of the implementation grows. The development and configuration of an optimal memory usage and thread model is also performed with this implementation, focusing on minimising kernel execution time before the introduction of multi-threading, CUDA streams and multiple GPUs.

Multi-threaded CPU The introduction of threading allows for concurrent execution of parts of the simulation. Running multiple replicas concurrently has two theoretical advantages. Firstly, on a multi-core system, full CPU utilisation is only possible with multi-threading. Secondly, the sharing of a single GPU between multiple CPU cores can thus exploit the amortised cost of CUDA context switches, provided that both the CPU and GPU are kept sufficiently busy.

Asynchronous GPU The use of CUDA streams enables asynchronous computation on both the GPU and CPU for a single thread of execution. Thus, maximisation of a system's resources is attained when the GPU performs the interaction potential calculations, while the CPU performs operations of lower computational cost.

Multiple GPU The final implementation stage is the development of a version of the code that uses multiple GPU devices. Should the GPU be fully utilised at all times, it will become the performance limiting factor. In this case, having an implementation that scales to multiple GPUs is desirable.

The most expensive part of the Monte-Carlo algorithm is the $O(n^2)$ non-bonded interaction potential summation, necessitating its implementation on the GPU. Further improvements in performance can be attained by using asynchronous GPU calls. By tuning the number of threads and using asynchronous GPU memory and kernel operations, it is theoretically possible to configure an implementation to maximise utilisation of both the GPU and CPU at all times. An investigation into interleaving asynchronous memory operations and asynchronous compute operations using multiple threads will be performed to evaluate the optimal usage model.

Before presenting the design for implementation, a summary of the Kim and Hummer coarse-grain model is discussed, detailing the sources from which simulations will be performed.

4.2 The Coarse-Grain Simulation Model

There are a number of specific features which promise that the model and simulation method will be amenable to GPU computation and, because of coarse-graining, speed up an already computationally efficient search method.

The model uses Metropolis Monte-Carlo and Replica Exchange, both guided by the energetic interaction potential of each conformation. For tractability, coarse graining is employed at the residue level, decreasing the number of bodies by a factor of approximately 20.

The calculation of the electrostatic interaction potential in the model is similar in structure to that of Anderson et al. [25], Friedrichs et al. [17] and Meel et al. [26]. Thus, the decomposition of pairwise potentials to CUDA threads is solved for direct electrostatics calculating the force/potential for each body. The second part of electrostatic potential calculation is summation, achievable with the optimised CUDA reduction [82]. An alternative approach to summation would be to use the direct summation on a lattice [18]. Given the complexity of the electrostatic potential summation for the non-bonded forces, approximately $O(\frac{N^2}{2})$, and the linear complexity of the bonded potentials, division of work between the CPU and GPU would favour concurrent calculation of the non-bonded potentials on the GPU and bonded forces on the CPU.

Coarse graining introduces a contact potential lookup as a function of the types of amino acid. This results in an indirective instruction in the pairwise potential calculations. Both residues must be loaded and subsequently inspected before the contact potential value for the short range interaction is retrieved. Thus, a suitable memory model and its result performance must be assessed.

The Monte-Carlo moves on this model are not performance limiting; Monte-Carlo moves perform operations of linear complexity with respect to the number of residues: translation, rotation and crankshaft moves. The time required for such operations and the generation of random numbers (effectively handled by generators such as the GNU Scientific Library) is much less than the interaction potential calculation. Thus, calculation of the mutations on the CPU with non-bonded calculations on the GPU is sensible, considering replica exchange and flexible linkers.

Replica exchange introduces an additional level of separability to the simulations, resulting in linear scalability to multiple CPU cores and GPUs. Beyond a critical simulation size, it will suit asynchronous sharing of a GPU between replicas, calculating the non-bonded potentials while calculating the bonded potential and the Monte-Carlo mutations of other replicas while the GPU is busy.

In our implementation we treat proteins as rigid bodies, reducing the conformational search space of the simulation and maintaining a constant contribution to free energy for a single protein throughout the simulation. Interactions between proteins are specified at the residue level using coarse graining. Amino acid are represented by a single spherical bead, representing a protein as a chain of beads with each bead centred at the position of the C_α atom of its corresponding amino acid (Figure 4.2).

Kim and Hummer use Replica exchange Monte-Carlo to perform docking simulations. A electrostatic interaction potential is used to evaluate the Monte-Carlo and replica exchange moves [12]. Our design of this system assumes that the size and length of the simulations will be dynamic, thus, the number of replicas and the number of Monte-Carlo iterations will be specified at runtime.

Interaction Potentials

Pairwise interaction potentials are used to express the energy between residues [12]. The non-bonded part of this potential consists of short-range Lennard-Jones-type pairwise potentials and long-range electrostatic Debye-Hückel-type pairwise potentials. Bonded potentials evaluate the contribution of flexible backbone links between secondary structures, and consist of bond stretching, bending and torsion potentials. However, bonded potentials can be omitted from our implementation because we treat proteins as rigid bodies.

Non-bonded potentials are calculated between all the pairs of coarse-grained amino acid beads, with the interaction pair potential between residues i and j distance r apart, $\varphi_{ij}(r)$, consisting of the sum of the Lennard-Jones-type potential, $u_{ij}(r)$, and the long range Coulomb potential, $u^{el}_{ij}(r)$.

$$\varphi_{ij}(r) = u_{ij}(r) + u^{el}_{ij}(r) \tag{4.1}$$

Expanding this, Kim and Hummer define the short-range interactions to be

$$u_{ij}(r) = \begin{cases} 4|\varepsilon_{ij}|[(\sigma_{ij}/r)^{12} - (\sigma_{ij}/r)^{6}], & \text{if } \varepsilon_{ij} < 0 \\ 4\varepsilon_{ij}[(\sigma_{ij}/r)^{12} - (\sigma_{ij}/r)^{6}] + 2\varepsilon_{ij}, & \text{if } \varepsilon_{ij} > 0, r < r^0_{ij} \\ -4\varepsilon_{ij}[(\sigma_{ij}/r)^{12} - (\sigma_{ij}/r)^{6}], & \text{if } \varepsilon_{ij} > 0, r \geq r^0_{ij} \end{cases} \tag{4.2}$$

where σ_{ij} is the interaction radius between residue types corresponding to i and j.

The interaction is either attractive ($\varepsilon_{ij} < 0$) or repulsive ($\varepsilon_{ij} > 0$). σ_i, the interaction radius of a residue, is determined using the van der Waals diameter of that residue [12]. ε_{ij} can be further decomposed into $\lambda(\epsilon_{ij} - \epsilon_0)$. The contact potential ϵ_{ij} is adjusted by the scaling factor λ and offset parameter ϵ_0, ensure that the contact potential between residues contributes the correct value to the van der Waals component [12]. The contact potential ϵ_{ij} is calculated from known contact potentials between residues and are experimentally derived (Table A.2) [157, 169, 170], while the scaling factors are fitted empirically to simulation data [12].

The interaction radii for a pair of potentials i and j is

$$\sigma_{ij} = \frac{\sigma_i + \sigma_j}{2} \tag{4.3}$$

The long-range Coulomb potential, accounting for an implicit solvent using Debye-Hückel-type potential is

$$u^{\text{el}}_{ij} = \frac{q_i q_j exp(\frac{-r}{\xi})}{4\pi D r} \tag{4.4}$$

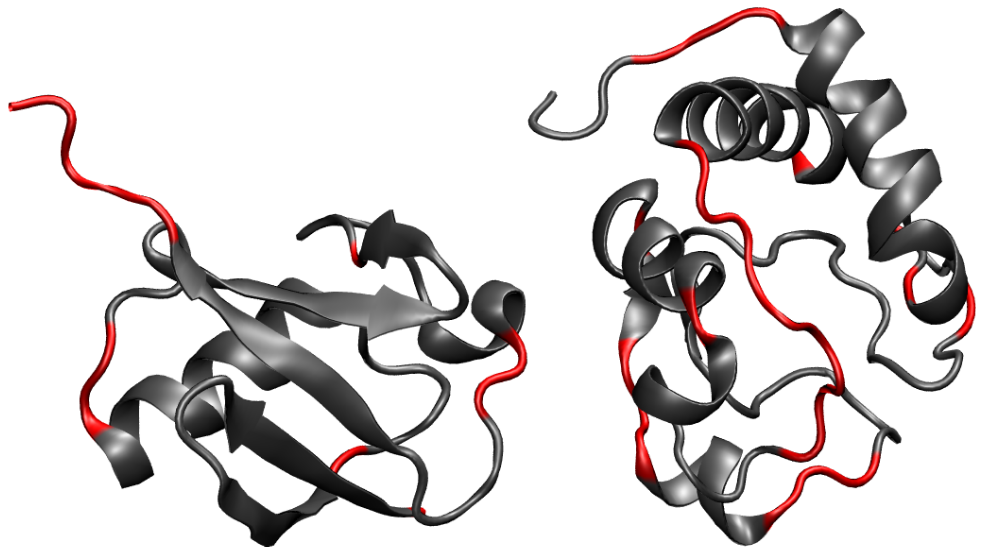


Figure 4.1: Flexible Linkers
Kim and Hummer include potentials for flexible linkers (red) undergo crankshaft rotations, rotating one residue about the axis of its neighbours while keeping the secondary and coarse tertiary structure of the protein unchanged.

The constants D and ξ refer to the dielectric constant of the solvent, in this case water, and the Debye screening constant respectively. In our implementation we will be using $D = 80$ and $\xi \simeq 10$Å as described in the original model. q_i and q_j refer to residue charges for pH7 [12]. Both interaction radius and charge are specific to each amino acid (Table A.1) with the van der Waals radius of each amino acid defining its bead's volume.

A weighting factor is introduced to account for solvent-accessible surface area (SASA) of each residue, modelling residues exposed to the solvent as having more influence on the interaction potential of the protein complex than residues contained beneath the molecular surface of the protein. In the simplest case $f_i = 1$, weighting the contribution between all residues interactions equally, (For other cases refer to Kim and Hummer [12]).

The total interaction potential confirmation,

$$U_{tot} = \sum_{ij} f_i f_j \varphi_{ij}(r_{ij}) \tag{4.5}$$

is the sum of all the distinct pairwise potentials. U_{tot} is the used the evaluation of Monte-carlo mutations. When U_{tot} is less than $2K_bT$ the complex formed between proteins is considered to be in a bound state.

Kim and Hummer allow for flexibility in their model by modelling the chains of residues between secondary protein structures as flexible links (red in Figure 4.1) and applying crankshaft moves. These moves rotate an amino acid bead about the axis formed between its neighbours. Thus, the secondary structure remains rigid (grey in Figure 4.1). Removing the crankshaft moves from the model results in rigid body protein domains. Keeping the structure of each

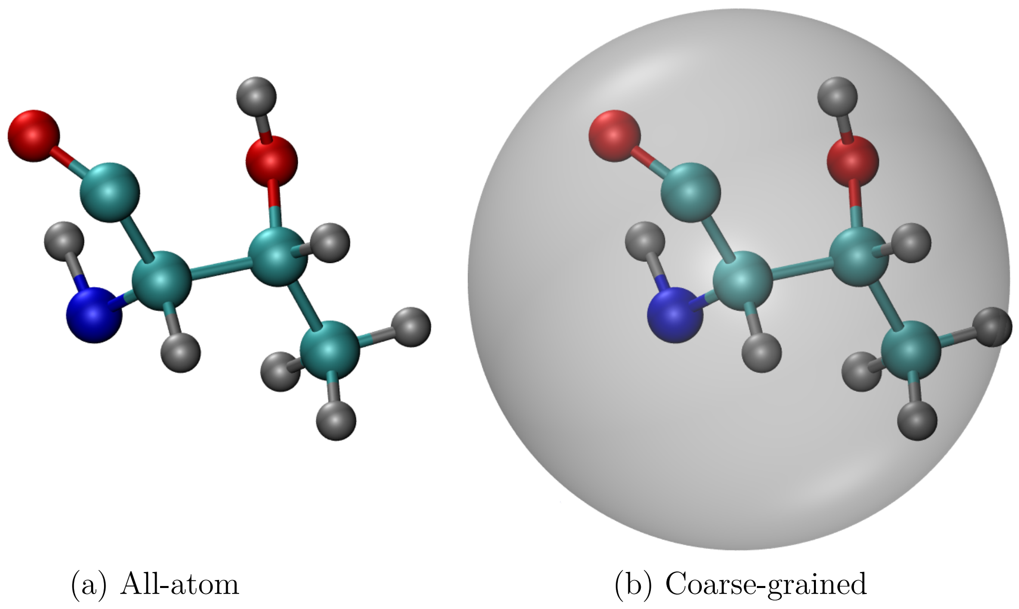


Figure 4.2: Coarse-grain Residue Representation. *(a) an all-atom ball and stick representation of the amino acid Threonine. Coarse-graining reduces the complexity of the representation to a single spherical bead, centred at the position of the alpha carbon (C_α) atom (b).*

protein rigid causes the energy contribution of the bonds, the angles and the torsions between residues to remain constant. Consequently, they do not affect the Monte-Carlo and the Boltzmann acceptance criteria (Figure 4.5, lines 4 and 8, respectively) because both criteria depend on the change in potential between iterations (ΔE) and not absolute potential.

Another computational benefit of rigid body domains is that the solvent accessible surface area (SASA) of each residue is constant. SASA is the fraction of the residue that is exposed to space outside of the molecule out of its total surface area. Kim and Hummer have six different weighting functions for the SASA, each of these can be substituted into f_i and f_j from Equation 4.5, provided the SASA of each residue is pre-calculated. We use the the simplest parameters setting both f_i and f_j equal to 1.

Our simulations are designed to accept Protein Database (PDB) descriptions of proteins in the PDB file format. In our case, a coarse-grained model is constructed from the atomic PDB data as follows. Each coarse-grain bead represents an amino acid residue using a sphere centred at the location of the alpha carbon (C_α) atom of each amino acid in the protein. The coarse grained residues are parametrised with specific values (see Table A.1) according to the type of amino acid they represent. Figure 4.2 shows an illustration of the conversion from all-atom amino acid representation to coarse-grain residue representation. Proteins are subsequently constructed as chains of coarse-grain amino acid residues from all-atom structures (Figure 4.3). Implementing a PDB file reader for the purpose of this project is trivial, as only the `ATOM` entries of the alpha carbons are relevant. C_α entries are invariably of the form:

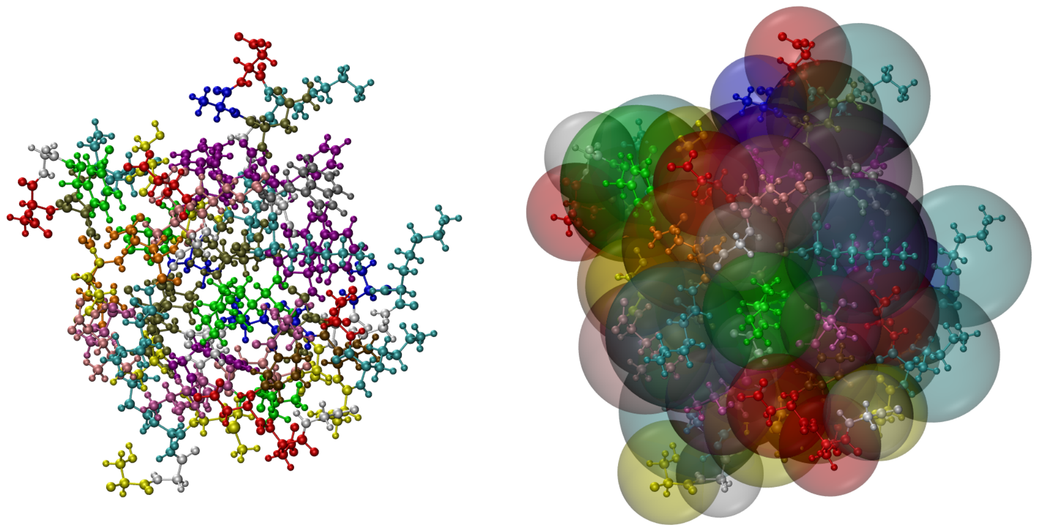

(a) All-atom (b) Coarse-grained

Figure 4.3: All-atom and Coarse-grain Representations of CspA *Major Cold Shock Protein of Escherichia coli (CspA) is a 69 residue long protein, consisting of 1004 atoms in total (513 non-hydrogen atoms). Coarse-graining reduces the all-atom structure (a) from a 1004 atom structure to a 69 grained structure. (b) coarse-grain beads representing each amino acid residue overlay the original structure, illustrating the differences between the two structures.*

```
ATOM   1597  CA  GLU B  49      26.452  16.593  19.165  1.00 41.35     C
```
where `CA` denotes C_α.

4.2.1 Simulation Outputs

A statistical docking simulations such as this, can output statistical information regarding the replicas in the simulation and the docked poses discovered during the simulation.

The statistical nature of Monte-Carlo docking requires our simulation to report metrics relating the the state of each replica. Our simulations will output the acceptance ratios for each replicas Monte-Carlo simulation and the replica exchanges. The fraction bound, the ratio of bound samples out of the total number of samples will also be recorded. This metric ultimately allows us to determine the dissociation constant for the complex [12] and the associated binding strength of the complex.

The bound samples need to be recorded for analysis at a later stage. This analysis will use existing tools to determine relevance of each docked structure. Clustering isolates candidate structures by population; a good model and successful will result in these structures being representative of the structures occurring in nature.

4.3 Algorithm Design

Efficient exploitation of the hardware on which an algorithm is to be run dictates how the algorithm is designed in order to scale favourably and maximise its ability to use its hardware resources. A scalable solution will be able to run on any feasible combination of multiple CPU cores and GPU devices efficiently without major modification. We aim to produce a single solution capable performing simulations with or without a GPU using multiple CPU cores. This is complemented with ability to use one or many GPUs, synchronously or asynchronously as required.

4.3.1 The Replica Exchange Algorithm

A generic version of the replica exchange, or parallel tempering, algorithm [171] for Monte Carlo (REMC) simulations (Figure 4.4) provides a starting point for designing a parallel algorithm. This algorithm contains an inherent degree of parallelism; each replica involves a Monte Carlo simulation which can be run independently and in arbitrary order. Consequently, the Monte Carlo searches performed (Figure 4.4, line 4) in this algorithm can be run concurrently in an "embarrassingly parallel" fashion. By performing the Monte-Carlo searches (4.4, lines 3-5) as collection of CPU threads, a multi-threaded version of the REMC algorithm is created to exploit the parallelism of the CPU that allows for a scalable implementation that can run on multiple cores using a shared memory model.

Others have used a similar approach to distributing replica exchange across a cluster, mapping one replica to each cluster node, due to the low communication requirements between replicas [132]. This design also provides a mechanism to distribute computation across multiple GPUs. Data can be transferred to the relevant device or machine before beginning the Monte Carlo search. Once all Monte Carlo searches are complete, the replicas are synchronised and wait while performing the exchange portion of the algorithm, before being allowed to continue with the next iteration of the simulation.

4.3.2 Monte-Carlo Searches

The most computationally expensive part of this docking application are the interaction potential calculations in each Monte-Carlo simulation due to their $O(N^2)$ complexity. Therefore, most effort will be focused on speeding up a single instance of the Monte-Carlo part of the simulation (Figure 4.5) and attempt to perform concurrent instances of the same algorithm on CPU multiple cores.

The steps of the Monte Carlo loop can be summarised as *save*, *mutate* and *evaluate* procedures, followed by either acceptance of the mutation or rejection and restoration of the previous

```
**Data**: $C$: The set of all replicas
$offset \leftarrow 0$
**while** $!ExitCondition$ **do**
    **foreach** $c_i \in C$ **do**
        $MonteCarloSearch(c_i, \phi)$
    **end**
    $i \leftarrow offset + 1$
    **while** $i + 1 \leq M$ **do**
        $j \leftarrow i + 1$
        $\Delta \leftarrow (\beta_j - \beta_i)(E(c_i) - E(c_j))$
        **if** $\Delta \leq 0$ **then**
            $swaplabels(c_i, c_j)$
        **else if** $U(0,1) \leq e^{-\Delta}$ **then**
            $swaplabels(c_i, c_j)$
         $i \leftarrow i + 2$
    **end**
    $offset \leftarrow 1 - offset$
**end**
```

Figure 4.4: Sequential Replica Exchange Monte Carlo. *A generic listing of the REMC algorithm. $E(c_i)$ is function to calculate the total potential energy of the system. $U(x,y)$ is a real uniform random number (x,y). M is the number of replicas.*

```
**for** $i = 1$ *to* $\phi$ **do**
    $c' \leftarrow c$
    $m \leftarrow U(0, N)$
    $t \leftarrow U(0, 1)$
    $c' \leftarrow Mutate(c', m, t)$
    $\Delta E \leftarrow E(c') - E(c)$
    **if** $\Delta E \leq 0$ **then**
        $c \leftarrow c'$
    **else if** $U(0,1) < e^{\frac{-\Delta E}{T}}$ **then**
        $c \leftarrow c'$
    **end**
**end**
```

Figure 4.5: Sequential Monte Carlo Search *Each Monte-Carlo search performs a random mutation, sampled uniformly ($U()$) to determine the mutation type (t) and molecule (m). The interaction potential (E) of each replica is then used to accept or reject the mutation.*

state. Of these steps, *mutate* and *evaluate* are the most computationally intensive.

Mutation steps involve either a translation or rotation of a specific molecule in the simulation. The molecule and mutation type are both selected using uniform random number generators. Translations are simply the addition of one 3-dimensional vector to N other 3-dimensional vectors. Rotations are more computationally intensive, as each vector must be rotated using a rotational matrix or quaternion. Regardless of the method, all mutation operations are $O(n)$ in time, where n is the number of mutated residues in N residues.

Evaluating a mutation is the most expensive operation, with time complexity $O(N^2)$. For large simulations, it is anticipated that $n << N$. Consequently, if a mutation is rejected, the cost of copying and restoring the previous state of the replica should be less than evaluation of the new potential. The evaluation of the interaction potential ($E(c')$) should account for the majority of computation time in any mutation scenario. In the least computationally intensive case, a molecule is translated and the translation is accepted because it results in a lower energy state ($\Delta E \leq 0$); in the most computationally intensive case, a molecule is rotated and the rotation is rejected ($\Delta E \leq 0$ and $U(0,1) < e^{\frac{-\Delta E}{T}}$). In both cases, the rotation and translation operations are of lower time complexity than the interaction potential calculation.

4.4 GPU Design

It is important to optimise the the division of work between a CPU and GPU to ensure the minimum number of idle bottlenecks occur. While a GPU may offer orders of magnitude more computing power than a CPU, it is important to not allow the CPU to wait while the GPU performs calculations or transfers data. While a GPU is busy, the CPU can prepare data for the next interaction potential calculation. This reduces the overall time spent waiting for CPU to GPU memory transfers and increased the overall throughput of a simulation. This strategy requires the use of asynchronous computation and at least two concurrent Monte-Carlo simulations.

The GPU initially performs the calculation of a replica's potential after a mutation. This requires a replica's data be copied to GPU memory or updated in GPU memory before every iteration (Figure 4.6). For a simulation using N residues, the transfer time of those residues is dependent on the data structures used for the replica and number of residues. By performing concurrent Monte-Carlo searches, both GPU and CPU can be kept busy. While one thread is performing a mutation, another is copying data to or from the GPU, while a third thread waits while the GPU evaluates a replica's potential. This allows both CPU and GPU to perform work at all times instead of one blocking the other. Technically, one can only perform GPU memory transfers and kernel invocations concurrently when using CUDA's stream mechanism [22] as each thread will own a CUDA context which has exclusive use of the GPU while it is active.

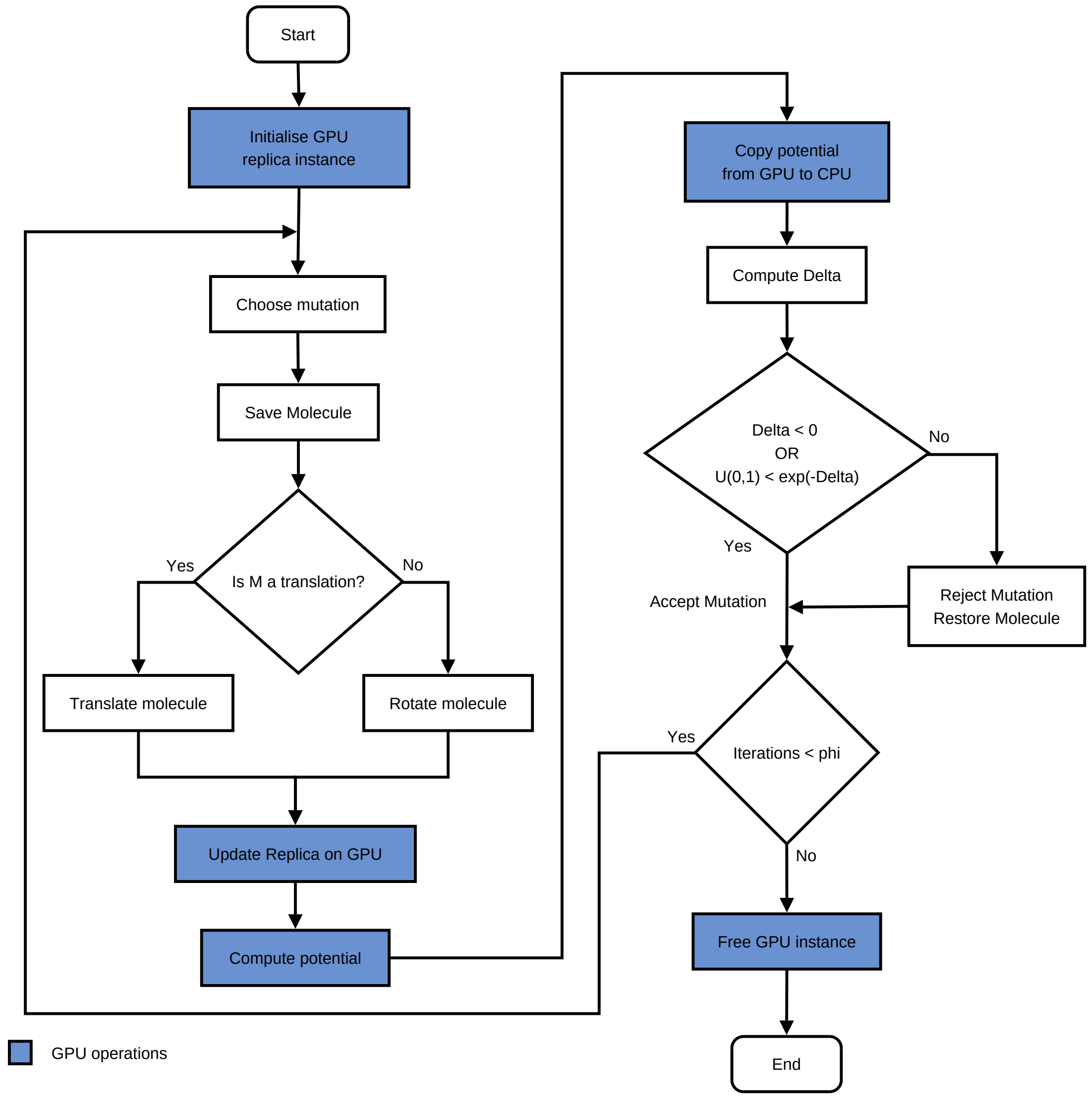


Figure 4.6: GPU Monte Carlo Search. *The sequential Monte-Carlo simulation loop, including GPU operations (in blue).*

4.4.1 Problem Mapping

Implementing an efficient GPU application will rely on correctly mapping the potential calculation to GPU hardware. We design our "kernel" to evaluate each pairwise in the same manner as Friedrichs et al. [17], using a tiling approach. Remembering that Friedrichs et al. perform molecular dynamics, which reduces the n^2 interactions to n velocity and position vectors, we must reduce all n^2 interactions to a single value.

The CUDA grid of thread blocks is used to address residues. For this a 2-dimensional grid of 1-dimensional thread blocks should prove optimal. If every pairwise interaction is viewed as a cell of a matrix (Figure 4.8), then thread blocks perform the accumulation of the interaction

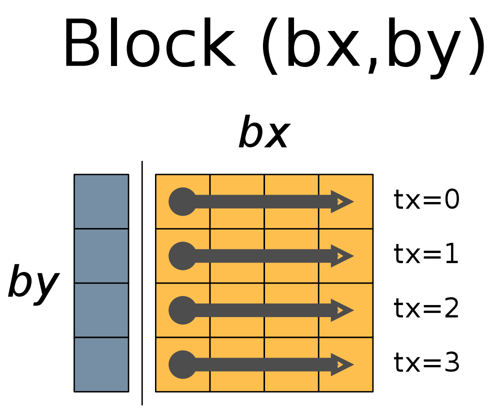


Figure 4.7: Thread Block Mapping
In a thread block, Block(bx,by), each thread is assigned a residue in the bx subset of residues according to its thread index, tx. This thread, in lock step with the other threads, loops though the subset by and computes the pairwise potential for its element on each element of the by subset. The results of each thread are again reduced to a single partial sum using a parallel reduction.

potential of spatially local residues. In each thread block, threads can efficiency cache a contiguous block of residues (the sub-row partitioned by bx) in shared memory on the GPU such that the latency of fetching each residue is reduced. Each thread block can then iterate through the cache, accumulating the pairwise interaction potentials between its residue (Figure 4.7). Potentials are mirrored diagonally and need not be performed twice [17]. Using this method a GPU kernel can process potentials between 33 554 432 (65536 × 512) residues.

Part of the pair potential, $u_{ij}(r)$ (equation 4.2), requires a random access look up. Since residues can be one of 20 different types, there are 210 unique interaction coefficients ε_{ij} determining this short-range component of the potential. The CUDA programming guide states that reading a randomly accessed value such as this from GRAM will take in the order of 400-600 clock cycles. This cost can be amortised, provided there is a sufficient number of instructions being performed while the value is retrieved. However, alternate storage, such as constant or texture memory on the GPU, should be used to improve the performance of these look-up. Implementations of each type of memory look-up will be performed to evaluate their impact on performance.

4.4.2 Multiple GPUs

The model described uses independent threading and will scale to multiple GPU devices as well as multiple nodes using one or more GPUs. Each Monte-Carlo thread can have one GPU associated to it. In the case of a single GPU, multiple threads can share that GPU. In the case of multiple GPUs, threads will be divided into groups and each group will share a GPU. As the number of GPUs in a simulation increases, the cost of transferring data also increases as the GPU devices must all share the same PCIE bus. This configuration model should also scale for

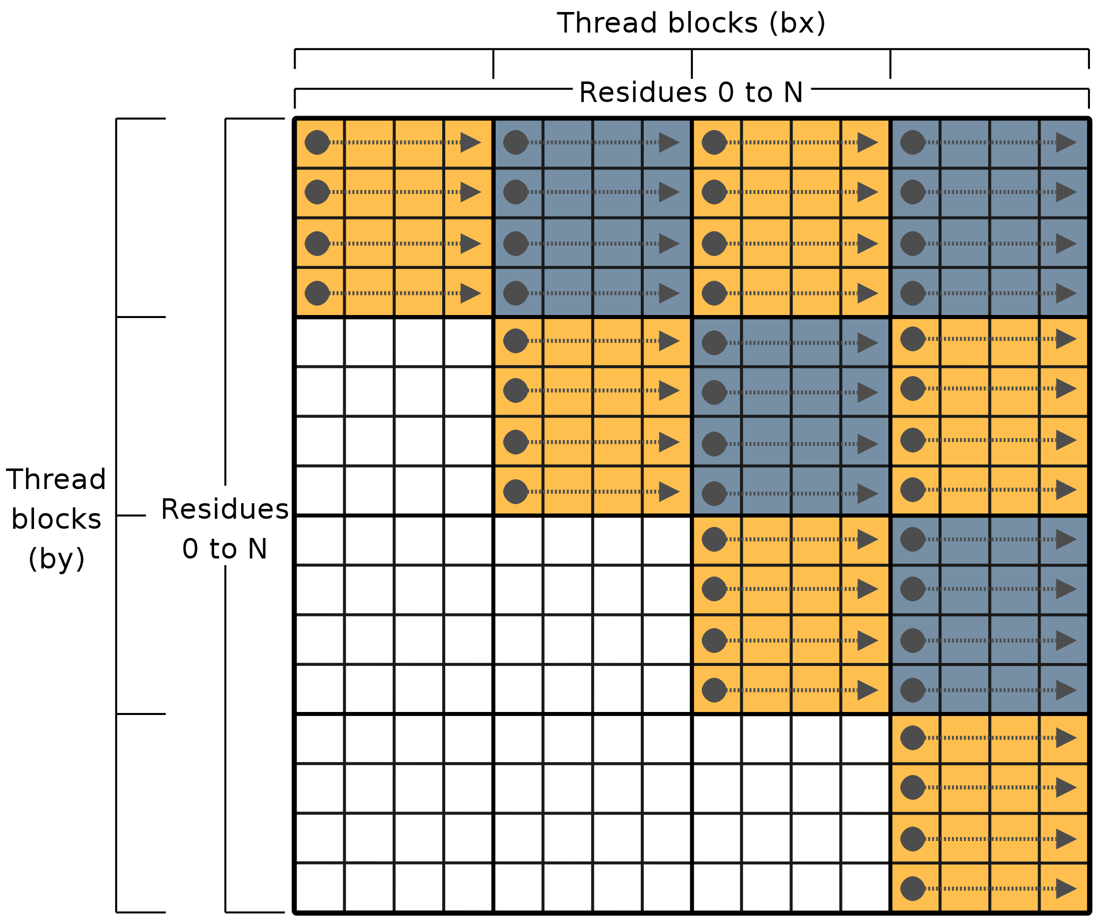


Figure 4.8: GPU Tiled Kernel. *Each thread calculates the potentials between one residue and all other residues in the same block. Thread blocks below the major diagonal will be terminated early, as they mirror the calculations performed by thread block above the diagonal, while blocks on the diagonal merely half their partial interaction potential to correctly account for the double calculation of certain pairs.*

multiple nodes of single or multiple GPUs due to the differing levels of parallelism implemented at replica exchange level using threads or Monte Carlo level using GPUs.

The major problem with this design is that GPUs should not be shared amongst CPU threads. Each time a command is issued to the GPU, the CUDA runtime context associated with that thread must be swapped onto the GPU, a costly operation if performed often. Thus, GPU sharing amongst threads is only a viable model if kernel execution times dwarf the cost of context switches.

A better alternative is to use asynchronous GPU calls. Replicas can be grouped and assigned a stream. This mean that, even on a single GPU and CPU core, a single runtime context is used and, GPU and CPU computation is overlapped. But, CUDA GPUs without asynchronous functionality [22] perhaps stand to benefit from the shared GPU usage case, even if it does add the additional context switching cost to computation. Both cases will be implemented and compared.

4.5 Design Summary

Many performance related factors influence a high performance implementation of the Kim and Hummer model. Hardware configurations range from CPU only computation configurations to multiple CPU and GPU computation configurations. Consequently, algorithms and methods need to be designed to scale for both problem size and hardware configuration. Tunable GPU kernels and multi-threading facilitates an implementation capable of exploiting the hardware on which it will execute. Together with benchmarking and profiling, an implementation based on the designs in this chapter will provide an insight into the utility of using GPUs to accelerate simulations using this model.

Chapter 5

Implementation

A detailed discussion of the methods used and decisions taken to implement the design discussed in the previous chapter follows. The implementation also provides practical insight into determining additional variables and factors influencing performance. Documentation of such factors and the techniques employed to improve them are included in the relevant sections.

We begin with a basic CPU only implementation, discussing the use of data structures and methods preparing data for simulation. Use of the appropriate random number generators is important for a good quality Monte-Carlo simulation, we discuss, briefly, the use of the Mersenne Twister and the benefits of using this random number generator for our simulations. A short description of our Monte-Carlo simulations and interaction potential calculation follow before a more detailed discussion about our particular implementation of the replica exchange algorithm.

Following this, detailed explanations of CPU and GPU implementations are included to show how the multi-threaded and GPU versions of code differ and accelerate the original algorithm and method. First, details regarding the steps taken to transform our sequential CPU code into multi-threaded CPU code by the use of Posix threads to scale to multiple cores are discussed. Following this a lengthy explanation of our GPU implementation follows. We build an optimised kernel, adhering to the CUDA Best Practice guidelines. We also discuss the use of specific hardware features on the GPU to attempt to find a solution to the problem of random contact potential look-ups, pertinent to the van der Waals component of the interaction potential between every pair of residues and the use of GPU special functions to maximise the computational throughput of our kernels.

Finally, the use of asynchronous GPU computing and the use of multiple GPUs is discussed, detailing the final level of optimisation we pursue.

Sundry details regarding our management of simulation statistics such as fraction bound and simulation sampling is included at the end of the chapter.

All of the code for the sections that follow is written in C and C++ for a Linux 32 bit operating system using an NVIDIA GTX280, CUDA 1.3 capable GPU device. Versions ranging from 2.0 to 2.3 of the CUDA tool kit are used for development. The GNU Scientific Library, the C++ posix thread library and the C++ Standard Template Library are also used for implementation.

5.1 Sequential CPU Implementation

Development of an implementation began with the design of a class hierarchy encapsulating aspects of the model to be implemented. Although the coarse-graining of a model simplifies its computational demands, it increases the meta-data required in the simulation. Interactions between each distinct pair of residues is unique and requires a lookup table for its van der Waals interaction potential. In this case it would be an inefficient use of memory to encapsulate the entire table in each residue as it would require at least 80 bytes (twenty 32-bit floats) more per residue than storing them in a table or map. The implications of such random access behaviour pose interesting questions regarding implementation on a GPU. Different strategies of implementation are employed to attempt to achieve speed-ups that vastly reduce simulation times.

5.1.1 Data Structures

A hierarchy of objects encapsulating the model's structure is detailed here. The smallest element in the simulations is the amino acid bead or residue. The following data is associated with each residue:

- **Type**: What amino acid it represents.
- **Position**: Where in 3D space it is situated.
- **Charge**: Its net electrostatic charge.
- **Interaction Radius**: Its van der Waals interaction radius.
- **Contact Potential**: How it interacts with other residues depending on their type.

Protein molecules are represented as a contiguous chain of these residues. For performance reasons, implementing an object that encapsulates all the data described above is undesirable. Were all the data for a residue to be stored in a residue object, it would require 20 floating point entries for its interaction potential values. Removing this array of values and replacing it with a lookup-table mechanism results in a residue object that contains a type (4 bytes), a position (12 bytes), an interaction radius (4 bytes) and a charge (4 bytes). In total, an object is 24 bytes large in contrast to a 104-byte object when the 20 contact potentials are included. This means that a model's memory footprint when using a lookup table is reduced to approximately a quarter of the original size. This design results in more cache hits due to the higher spatial locality of residue data in memory. By representing the residues as compactly, less memory bandwidth is required to transfer them from RAM to CPU. Consequently time taken by data transfer is reduced, lowering dependence on memory bandwidth which is beneficial for performance.

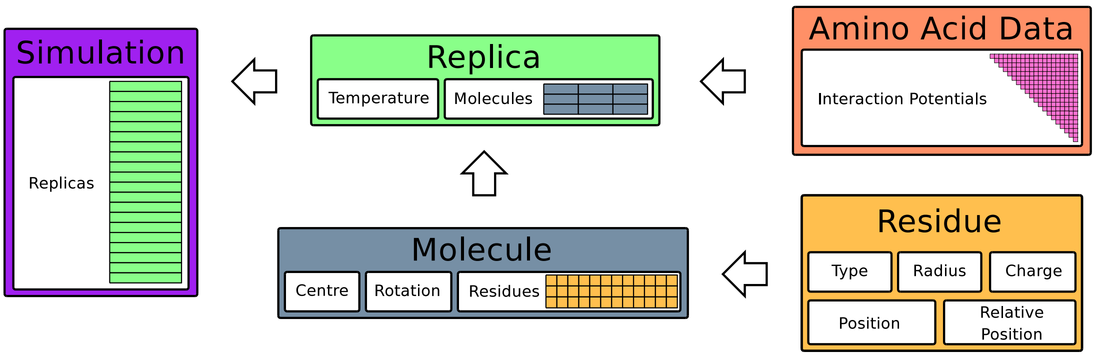


Figure 5.1: CPU Object Hierarchy
Residues are aggregated into arrays in molecule objects, which are in turn aggregated in Replica objects to represent a single conformation of the simulation. The contact potentials between residues are stored in a separate data structure (Amino Acid Data) to avoid repetition and allow for a compact representation each residue.

Residue objects are aggregated together into a contiguous array belonging to a *Molecule* object. Storing both the absolute position of each residue and its relative position (with respect to the molecule centre) simplifies computation at various stages of execution. Because molecules are treated as rigid bodies, the position of each residue relative to a molecule's centre is constant and can be calculated when initialising the molecule. Were only one position to be stored, although space saving, it would cost CPU cycles during simulations.

If only the absolute position is stored, the center of each molecule would need to be calculated for each rotation, subtracted from the absolute position, stored and finally added to the new position once the rotation has been performed. Altogether this is an $O(2N)$ operation, where N is the molecules length in residues.

If only the relative positions are stored, translations would be more efficient as performing a translation would only require modifying the center variable of each molecule. However, to calculate the interaction potential between molecules, one of two implementations can be used. The relative position and the center must be added together to calculate the correct interaction potential. This can be done before any calculations are performed and then reversed after all the calculations ($O(2N)$ for all N residues in the simulation), or, in the worst case, the absolute position is calculated at each pair-wise interaction ($O(N^2)$).

To summarise, storing both relative and absolute positions eliminates the need for intermediate calculations.

Replica objects aggregate *Molecule* objects along with the replica's temperature, encapsulating all the data required to perform Monte-Carlo simulations. Figure 5.1 illustrates a simplified representation of the data structures used for CPU simulations.

A single instance of all contact energies can be attached to the simulation in the form of the *Amino Acid Data* object. This object is a 2 dimensional array of values from Table A.2. Each replica contains a pointer to a single instance of the lookup table, eliminating the need for multiple instances of the same data. Altogether there are 210 unique pair interactions. A 20 by 20 array represents this data. By storing the matrix of lookups mirrored about its major axis, pair values can be retrieved using M[y][x] or M[x][y], which is useful because one would have to otherwise sort the inputs and access the values using M[*min*(x,y)][*max*(x,y)]. This would also introduce computing overhead, since *min* and *max* operations are not free.

5.1.2 Monte-Carlo and Random Numbers

Monte-Carlo simulations rely heavily on good-quality random numbers [128]. In this case, a random number is consumed by each of the following tasks:

- Choosing which molecule to mutate.
- Determining the type of mutation.
- Generating the axis about which to perform the mutation.
- Determining if a mutation can be accepted because it fits the Boltzmann distribution.
- Performing replica exchange.

Hellekalek's article *Don't Trust Parallel Monte Carlo!* [128] lists the inherent properties a simulation requires from its random numbers. In the case of parallel Monte-Carlo algorithms, the deterministic algorithms that generate random numbers will produce correlated numbers if enough random numbers are consumed. If this correlation is such that it interferes with the simulations random nature, the results that the simulation produces may be useless [128].

The Mersenne Twister is specifically designed with Monte-Carlo simulations in mind and provides a random number generator with a period of $2^{19937} - 1$ [129]. Considering that our implementation of the model consumes approximately 5.5 (1 for molecule section, 1 for mutation type, 3 for generating the mutation axis and 0 or 1 to conform to the Boltzmann distribution if the interaction potential for the new configuration of residues is greater than the old configuration) random numbers per Monte-Carlo iteration, a 20 replica simulation will consume approximately 2^{40} random numbers after 10^{10} Monte-Carlo steps. Consequently, the use of a single instance of the Mersenne Twister will be sufficient for simulations of the same duration as those performed by Kim and Hummer to gather data. Consuming only 2^{40} random numbers for simulations of 10^{10} Monte-Carlo steps means that if simulations were run for far longer durations that those stated, one instance of the Mersenne Twister would still be sufficient.

In addition to the large period of the Mersenne Twister (MT), it is able to generate numbers quickly [129] and an implementation is included in the GNU Scientific Library. Assuming that GSL implements the algorithm correctly, using such a recognised implementation eliminates unnecessary re-implementation with potentially incorrect code. The SIMD-oriented Fast Mersenne Twister (SFMT) is also available for SSE2 enabled architectures. SFMT improves on the MT's speed by using the 128 bit calculations afforded to it by SSE, resulting in a speed-up of approximately 4 times that of the original 32-bit MT implementation. SFMT also increases MT's period from $2^{19937} - 1$ to $2^{216091} - 1$ [172]. SFMT is not used as it is not included in a standard library such as GSL.

Monte-Carlo Mutations

Performing a rotation or translation is the same for both CPU and GPU implementations. Generating the axis about which to perform the rotation is implemented by generating a 3 dimensional vector using a dedicated instance of the Mersenne Twister. Random numbers are generated using GSL's *get_rng_uniform* function which returns numbers in the range $[0;1)$. A random vector is generated with all its components in the range $[0;1)$. It then has 0.5 subtracted from each of its components and it is normalised. This ensures that it is random in any direction and not only the first Cartesian octant. The vector is then used as a rotational axis to generate a rotational matrix or a quaternion, representing a clockwise rotation of 0.2 radians [12]. The rotation is then applied to the position of each residue relative to the centroid of the rotating molecule. Both quaternion rotation and matrix rotation are implemented due to the numerical errors that became apparent during validation (see Section 6.1.2).

Translations are dealt with similarly: scaling the random vector to length 0.5Å and adding it to the position of each residue of the relevant molecule.

Mutations of this sort can be made into parallel operations. The use of SSE would probably speed-up the above method by between 2 and 4 times. However, both rotation and translation are $O(N)$ operations, meaning their effect on simulation time is much less significant compared to calculating the interaction potential, which, as discussed below, is of $O(N^2)$ complexity.

5.1.3 Evaluating Global Potential Energy

Monte-Carlo moves are evaluated using the interaction potential:

$$U_{tot} = \sum_{ij} f_i f_j \varphi_{ij}(r_{ij})$$

where pairwise potential, $\varphi_{ij}(r)$, expands to

$$\varphi_{ij}(r) = u_{ij}(r) + u_{ij}^{el}(r)$$

representing the Coulomb and short range interactions. Implementing U_{tot} involves looping though each molecule and performing a pairwise comparison between its residues and residues belonging to other molecules, generating distinct pair of interacting residues.

```
   Data: M[m]: Molecule $m$ of a replica instance with $N$ molecules. M[m][r] denotes residue
         $r$ of molecule $m$.
   Data: LJ(i,j): Lookup table for short range residue interactions.
 1 $U \leftarrow 0$
 2 for $m_i = 1$ to $N$ do
 3     for $m_j = m_i + 1$ to $N$ do
 4         for $r_i = 1$ to $M[m_i].residues$ do
 5             for $r_j = 1$ to $M[m_j].residues$ do
 6                 $r \leftarrow \| M[m_i][r_i].position - M[m_j][r_j].position \|$
 7                 $e_{ij} \leftarrow \lambda(LJ(i,j) - e_0)$
 8                 $\sigma_{ij} \leftarrow (M[m_i][r_i].radius + M[m_i][r_i].radius)/2$
 9                 $u_{ij} \leftarrow -4 \cdot e_{ij} \cdot {\sigma_{ij}}^6({\sigma_{ij}}^6 - 1)$
10                 if $e_{ij} > 0$ and $r < \sqrt[6]{2} \cdot \sigma_{ij}$ then
11                     $u_{ij} \leftarrow -u_{ij} + 2e_{ij}$
12                 $u_{ij}^{el} \leftarrow M[m_i][r_i].charge \cdot M[m_i][r_i].charge \cdot \frac{exp(-r/\xi)}{4\pi Dr}$
13                 $U = U + u_{ij} + u_{ij}^{el}$
14             end
15         end
16     end
17 end
18 return $U$
```

Figure 5.2: CPU Implementation of Interaction Potential
Implementation of Equations 4.1 to 4.5 results in four nested loops to iterate over all the residues in a simulation. The divergence in Equation 4.2 can be reduced to only 2 calculations as shown in lines 9-11, since its first and third parts are equivalent.

This also provides a starting point from which to begin optimisation. Computing the interaction potential is order $O(\frac{N(N-1)}{2})$ complexity since, each unique pair of residues contributes to the interaction potential. The aggregation of residues into molecules further reduces the complexity as residues belonging to a molecule are only compared to residues belonging to other molecules. In the UIM1/Ub reference case, the two molecules of size 24 and 76 mean that only 1824 comparisons actually occur, as against the theoretical worst case of 4950 ($\frac{N^2}{2} - \frac{N}{2}$ where $N = 100$).

For larger systems containing more molecules, the number of comparisons will tend to $\frac{N^2}{2}$ as the number of molecules increases. Simulating macro-molecular crowding resembles such larger systems, requiring a scalable implementation of the code in Figure 5.2.

5.1.4 Replica Exchange

Replicas are initialised from a single instance of a *replica* object which is then duplicated as needed. The *Simulation* object in Figure 5.1 is initialised as a dynamic array of *Replica* objects copied from the initial replica. Values for initialising replicas, such as number, temperature and molecules are taken from the input file for a simulation run and copied into the initial instance of a replica as they are common to all replicas. Finally, the temperature of each replica is set to the appropriate value.

For replica exchange, the range of temperatures for the replicas is a geometric progression, $\beta^*_0; \beta^*_1 ... \beta^*_{i-1}; \beta^*_i$ [130], and $\beta^*_i \equiv \frac{T}{T_i}$ where T is room temperature. This progression is calculated using the maximum temperature and minimum temperature input values (T_{max} and T_{min}) making $T_i = T_{min} r^i$ where $r = \left(\frac{T_{max}}{T_{min}}\right)^{\frac{1}{N-1}}$. Setting $T_{max} = 500K$ and $T_{min} = 250K$, reproduces the temperature progression as used by Kim and Hummer in their simulations [12].

Each temperature in the simulation has an associated fraction bound and acceptance ratio. These values are encapsulated in the replica to which they apply to at a specific time in the simulation. However, this causes a minor problem when performing replica exchange. It is most efficient to perform replica exchange in place, as it minimises the amount of memory copies required for exchanges.

This is done by exchanging only the temperature, fraction bound and acceptance ratio data of replicas, avoiding an unnecessary exchange of all the data in replicas being exchanged. However, the replica exchange operation requires an ordering of replicas by temperature and not array position, introducing a potential problem that can be alleviated by either sorting by temperature before each replica exchange or by maintaining an ordered map from temperature to a replica's location. The latter option is implemented and illustrated in Figure 5.3. The map $TR(x)$ is used to maintain an ordering of replicas by temperature while keeping the replicas in their original location in memory. The most important reason for not sorting replicas is to simplify multi-threading and using multiple GPU devices. In the multiple GPU case, replica data swaps require data to be swapped between GPU memories, an operation that is undesirable because all molecule data will need to be swapped between the two participating replicas. If only temperature is exchanged, zero bytes relating to the structure of the replicas is modified meaning that the GPU will not need any re-initialisation or updates after a replica exchange.

The aforementioned sections complete our sequential CPU implementation with the core functionality of this implementation is listed in Figure 5.4. The implemented code differs

```
Data: $R$: Replicas
Data: $T$: Temperatures, ordered lowest to highest
Data: $TR(x)$: Map from $T_x$ to its corresponding replica.
offset ← 0
while !Exit Condition do
    foreach $R_i \in R$ do
        MonteCarloSearch($R_i, \phi$)
    end
    $i \leftarrow offset + 1$
    while $i + 1 \leq T.size$ do
        $j \leftarrow i + 1$
        $a = TR(T_i)$
        $b = TR(T_j)$
        $\Delta \leftarrow (\beta_a - \beta_b)(E(R_a) - E(R_b))$
        if $U(0,1) < min(1, e^{\Delta})$ then
            Exchange($R_a, R_b$)
            swap($TR(R_a.temperature)$,$TR(R_b.temperature)$)
        $i \leftarrow i + 2$
    end
    $offset \leftarrow 1 - offset$
end
```

Figure 5.3: Replica Exchange Implementation
Swapping the data between R_i and R_j would prove inefficient for very large replicas as it scales linearly with replica size. However, the primary benefit of maintaining the TR map becomes significant when replicas reside in different memory locations, e.g. R_1 to R_{10} on one GPU and R_{11} to R_{20} on another GPU.

marginally when compared to the pseudo-code listed in design Figures 4.4 and 4.5 in the previous chapter.

Differences between the design and implementation are largely due to minimising memory operations in both replica exchange and Monte-Carlo mutations. In the original design, before a mutation, a copy is made of the entire replica, the copied replica was then mutated and the original replaced if the mutation was accepted. This operation can be improved in two ways. Firstly, by copying only the molecule being mutated instead of the entire conformation. This results in fewer memory operations as the overall size of the saved structure is smaller. Secondly, depending on the acceptance ratio of a simulation, if there are more accepts than rejects, it is better to mutate the original molecule and restore it if it is rejected, resulting in fewer memory operations over time.

The sequential CPU implementation aims to directly implement the method described by Kim and Hummer [12] as simply as possible to verify that the implementation of the model is correct. This code forms a reliable starting point for multi-threaded and GPU accelerated code, discussed in the sections that follow.

```
Data: $R(t)$: The set of $N$ replicas at step $t$
Data: $M_k[x]$: Molecule $x$, of replica $k$
$offset \leftarrow 0$
while $steps{+}{+} < REsteps$ do
    for $k = 1$ to $N$ do
        for $s = 1$ to $\phi$ do
            $u \leftarrow U_{\aleph}(1, N)$
            $m \leftarrow M_k(u)$
            $Mutate(M_k[u])$
            $\Delta E \leftarrow E(R_k(s)) - E(R_k(s-1))$
            if $\Delta E > 0$ and $U(0,1) > e^{\frac{-\Delta E}{RT}}$ then
                $M_k[u] \leftarrow m$
        end
    end
    $i \leftarrow offset + 1$
    while $i \leq M - 1$ do
        $j \leftarrow i + 1$
        $\Delta \leftarrow (\beta_i - \beta_j)(E(R_i) - E(R_j))$
        if $U(0,1) < min\{1, e^{\Delta}\}$ then
            $Exchange(R_i, R_j)$
        $i \leftarrow i + 2$
    end
    $offset \leftarrow 1 - offset$
end
```

Figure 5.4: Sequential CPU Implementation
The complete implemented Replica Exchange Monte-Carlo algorithm for this model differs in a few places from the design in chapter 4. $U_{\aleph}(1, N)$ and $U(0, 1)$ are uniformly distributed random numbers, on natural and real domains respectively. The Mutate function performs MC rotation and translation operations, while the Exchange function performs part of the replica exchange.

5.2 Multi-core CPU Implementation

Replica Exchange Monte-Carlo scales well on multi-core architectures primarily due to the Monte Carlo parts of the algorithm, each of which map onto a thread of execution or compute node in a cluster [131, 132]. This is possible because this section of the algorithm is a collection of independent Markov processes, each requiring no communication or synchronisation, while the Monte-Carlo simulations between replica exchanges are performed. In the case of our algorithm, the ratio of Monte-Carlo moves to replica exchanges is high (5000:1), resulting in large sections of sequential embarrassingly parallel, computationally demanding code. If Amdahl's law is applied to a configuration of 20 replicas, each taking 5000 MC steps between exchanges and assuming both a Monte-Carlo step (T_{MC}) and a replica exchange (T_{RE}) take 1 time unit each (T_{MC} is actually far greater than T_{RE}, but this doesnâĂŹt change the result) executing across N threads, the theoretical execution time and resultant speed-up for this number of replicas would be N, were there sufficient processors:

$$\begin{aligned}
T_{seq} &= T_{RE} + 20 \cdot T_{MC} &&= 1 + 20 \cdot 5000 = 100000 \\
T_{par} &= T_{RE} + \frac{1}{N} \cdot 20 \cdot T_{MC} &&= 1 + \frac{100000}{N} \\
\text{Speed-up} &= \frac{T_{seq}}{T_{\|}} &&\approx N
\end{aligned}$$

As the ratio of compute time between the Monte-Carlo portion of code and the Replica exchange increases, speed-up becomes linear. Were the simulation to execute on multiple nodes, communication time would negatively affect the speed-up if the amount of time required for synchronisation and data transfer was significantly large. Remapping of the replica exchange algorithm (as discussed in section 5.1.4) from temperature ordered data to replica locality ordered data will significantly reduce such communication time. For replica exchange, the only time when replicas mush synchronise and communicate, only 6 floating point variables are swapped between replicas. These variables relate to temperature and counters for accumulated bound state and acceptance ratio.

Our initial threading model, involves launching the application (main thread) and this thread is then responsible for Replica Exchange. Between exchanges, N threads are created to perform the Monte-Carlo simulations. These threads are then joined and replica exchange performed. This process is repeated until the simulation has performed enough Monte-Carlo steps, as is illustrated in Figure 5.5.

However, due to the hardware, simply allocating 1 replica to 1 thread is highly inefficient because this is a computationally expensive problem. Threads will be competing for CPU time unless the number of CPU cores is equal to or greater than the number of threads. The simulation will begin to thrash because the operating system's scheduler will attempt to give all threads an equal share of CPU time. Hence, threads are swapped in and out rapidly, to the detriment of overall efficiency. To avoid thrashing, there needs to be sufficiently few threads running to ensure that as many CPU cycles as possible are spent computing rather than swapping threads.

Assigning a set of replicas to each thread achieves this. Conceptually, this layout is identical to that in Figure 5.5, except that there is now an N to 1 mapping of replicas to threads. The addition of an outer loop in each thread to process all its associated replicas instead of only one replica is the only change to the implementation. To assign N replicas to T threads, T sets of $\lceil \frac{N}{T} \rceil$ contiguous replicas are assigned to a thread until all replicas have been assigned. Should T not divide N the final thread is merely assigned $N - \lceil \frac{N}{T} \rceil \cdot (T-1)$ replicas. The idea of in-place replica exchange now becomes beneficial. If replicas are initialised and remain with their data in a permanent place in memory, or on a specific compute node in a cluster, the division of labour between threads becomes trivial as the replica to thread mapping needs to be performed only once at the beginning of the simulation and any context sensitive data, such as GPU memory, remains associated with the thread that created it. Thus, it eliminates the need to copy data between threads after replica exchange.

The mapping of N replicas to T threads parametrises the implementation such that it is configurable for multiple architectures. Initialising $T = 1$ enables the runtime configuration of the implementation to run sequentially, which is optimal for running a simulation on a single-core CPU. Because these simulations are largely compute bound, the optimal number of threads to cores is one-to-one.

Were GPUs not used, the above implementation would suffice. However, GPU acceleration necessitates that the simple model described thus far be changed due to a bug that causes a memory leak between the NVIDIA device driver and the CUDA runtime. This bug arise due to the way CUDA runtimes are initialised.[1]

Before a GPU is used it needs to be initialised using the call CUDA API call, *cuInit*. What this does is attach an instance of the CUDA runtime to each thread from which CUDA functions are called. Every time a new thread is launched the first CUDA call leaks a small amount of memory. Over time this amounts to a large amount of memory, causing the application to run out of memory and crash. There are two was of overcoming this impasse; wait for NVIDIA to fix the bug, or workaround it.

Eliminating this bug via a workaround is fairly straight forward because it only occurs if threads are continually created. The easiest method to eliminate it is to use a single set of threads for the entire simulation. The problem of parallel replica exchange then becomes a type of producer-consumer problem. The Monte-Carlo simulations can be viewed as producers,

[1]Reported on CUDA forum and said to be resolved internally on 22 Sept 2008, updating the tool kit at this time did not resolve our problem.

producing the next versions of the replicas, which the replica exchange then consumes.

Producer-consumer problems typically have two processes depending on each other, where the consumer needs the producer to prepare its data before it can proceed. Essentially, while the producer is producing, the consumer is waiting for items to be produced before it can consume them. Replica exchange bears similarities to this because it requires Monte-Carlo to be finished before it can exchange replicas which then allows Monte-Carlo to continue.

The action of joining a thread is by default a blocking call if there is no expiration time associated with the join. The synchronisation at the end of all Monte-Carlo simulations is performed by using a join that will wait indefinitely, resulting in all threads synchronising before any replica exchange is performed. However, as soon as the Monte-Carlo threads become reusable, some degree of explicit inter-thread communication is required as there is no implicit synchronisation caused by joining the threads.

To incorporate the workaround, the initial program thread, which performs replica exchange (on the left in Figure 5.6), creates and assigns replicas, before creating as many threads as required. These threads then run for the duration of the simulation, only ever initialising a CUDA runtime once, thereby eliminating the accumulation of leaked memory. The replica exchange thread, after creating all the worker threads has to wait for a signal from a worker thread. Once created worker threads perform Monte-Carlo simulations and enter a waiting state once complete. The last thread to complete its Monte-Carlo simulations is tasked with signalling the parent thread to perform replica exchange. After replica exchange completes, all the threads are signalled to continue with their next Monte-Carlo simulations. This loop continues until enough steps have been taken, at which point the threads are joined with the main thread and the simulation exits.

What is immediately apparent is that there needs to be a mechanism to determine if all threads are waiting before performing replica exchange. This is done by using a waiting thread counter variable, which is incremented as threads complete their Monte-Carlo steps. Using a mutex to lock and unlock the variable ensures that the count remains accurate and that when it indicates that $T-1$ threads are waiting, the T^{th} thread entering a waiting state, signals that the replica exchange to start. The signal passes the same mutex used to increment the counter, thereby making it impossible for any worker threads to continue past the waiting state they are in, which is dependent on the mutex that the replica exchange holds. Once complete, the replica exchange broadcasts to all the threads that they may continue.

Posix threads implement waiting such that when the wait condition is fulfilled, the signalling thread atomically passes the mutex token associated with the wait condition to the waiting thread, should it be holing that mutex. In the broadcast case, where multiple threads are signalled, they compete for the mutex. Only one thread can secure the mutex after the broadcast, therefore, the mutex is released immediately by the Monte-Carlo thread to allow the other Monte-Carlo threads to continue.

To avoid race conditions, which would ultimately cause all Monte-Carlo threads and the replica exchange thread to reach their waiting states simultaneously and the simulation to become deadlocked, the following locking mechanism is used. It employs a single mutex and two wait conditions. Having only one wait condition suffices if there is only one Monte-Carlo thread, but the possibility of N threads necessitates the use of two.

Figure 5.7 shows, in black, the paths which may not be executed concurrently. If for any reason, all threads are in states ③ and ④ simultaneously, the system deadlocks. Deadlock is completely avoided by using a mutex, shared between the replica exchange thread and the Monte-Carlo thread. Before initialising all the Monte-Carlo threads, the main thread, responsible for replica exchange holds the mutex. After initialising all worker threads, the main thread waits for the first iteration of the Monte-Carlo simulations to complete (state ④). The posix documentation recommends that the thread calling the wait should own the mutex associated with that wait to ensure predictable behaviour[2]. Because of this, sharing a single mutex amongst all threads prevents the Monte-Carlo threads from signalling (state ②) unless the main thread is waiting, and conversely, the main thread will only own the mutex if the Monte-Carlo threads are in a waiting state (state ③). This works because the mutex is released atomically when a thread enters a waiting state, eliminating the possibility of a race condition to own the mutex. To allow multiple Monte-Carlo threads to run concurrently, the mutex is immediately released after resumption (state ③), this allows the other threads to resume on the same condition/mutex pair. When a thread is finished its Monte-Carlo iterations, it contends for a lock on the mutex and, after attaining the lock, increments a counter. When the final thread increments the counter and determines that all threads are waiting, it signals the main thread and then waits. The main thread, subsequently, performs replica exchange and resets the counter of waiting threads to zero before broadcasting to the Monte-Carlo threads that they may continue. The main thread then waits, releasing the mutex such that the Monte-Carlo threads may continue. This process repeats until sufficient Monte-Carlo steps are performed. It is always possible to determine when each thread must terminate without any inter-thread communication because each thread counts how many iterations have been performed.

The threading model and the use of CUDA are in many ways decoupled. The number and layout of threads does not affect the ability of an application to use CUDA. However, it is reasonable to assume, because each thread by default has its own associated CUDA runtime that there will be scheduling overhead in swapping between concurrent GPU contexts. Performance predictions are made even more difficult by using asynchronous GPU calls. These factors are

[2]The Single Unix Specification, Version 2, The Open Group, 1997

benchmarked and compared in Chapter 7. The efficiency of multi-threading will also be discussed at a later stage. It stands to reason that, because calculating interaction potential is a compute bound problem, optimal CPU usage will occur when each Monte-Carlo thread has its own CPU core. However, if work is outsourced to the GPU this may no longer apply. Benchmarking and profiling of the system described in this section will be performed to ascertain how it would theoretically benefit from GPU acceleration, followed by actual tests of acceleration due to the GPU implementation.

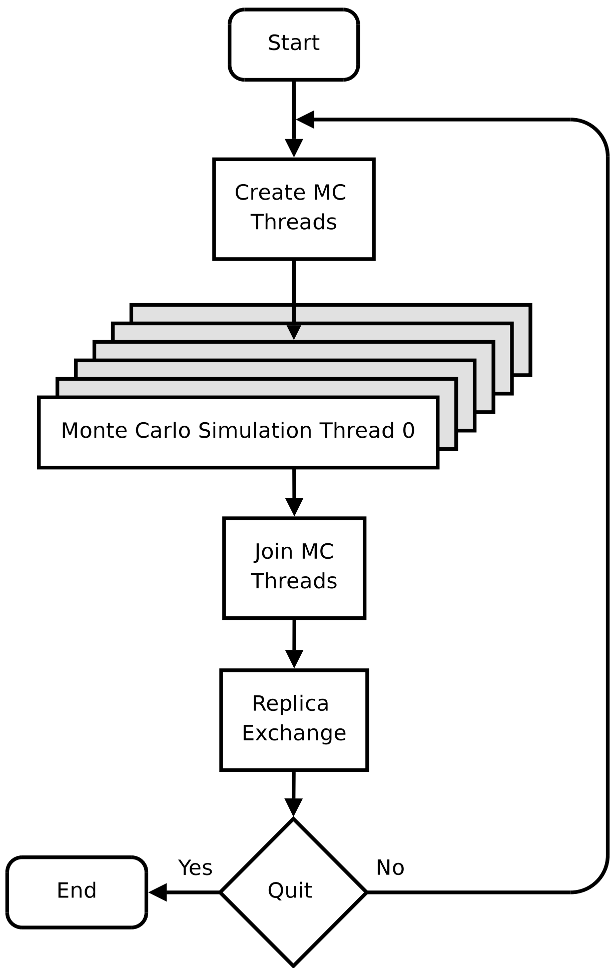


Figure 5.5: Multi-threaded Replica Exchange
Replicas are assigned to threads and these threads are created when Monte-Carlo simulations need to be performed. After the required number of steps, the threads are joined and replica exchange is performed. This cycle is repeated until the required total number of Monte-Carlo steps has been taken. This implementation is sufficient when no GPU, or only one GPU, is used for acceleration. The cost of launching and joining threads between every replica exchange iteration is amortised, provided the Monte-Carlo simulations are long enough.

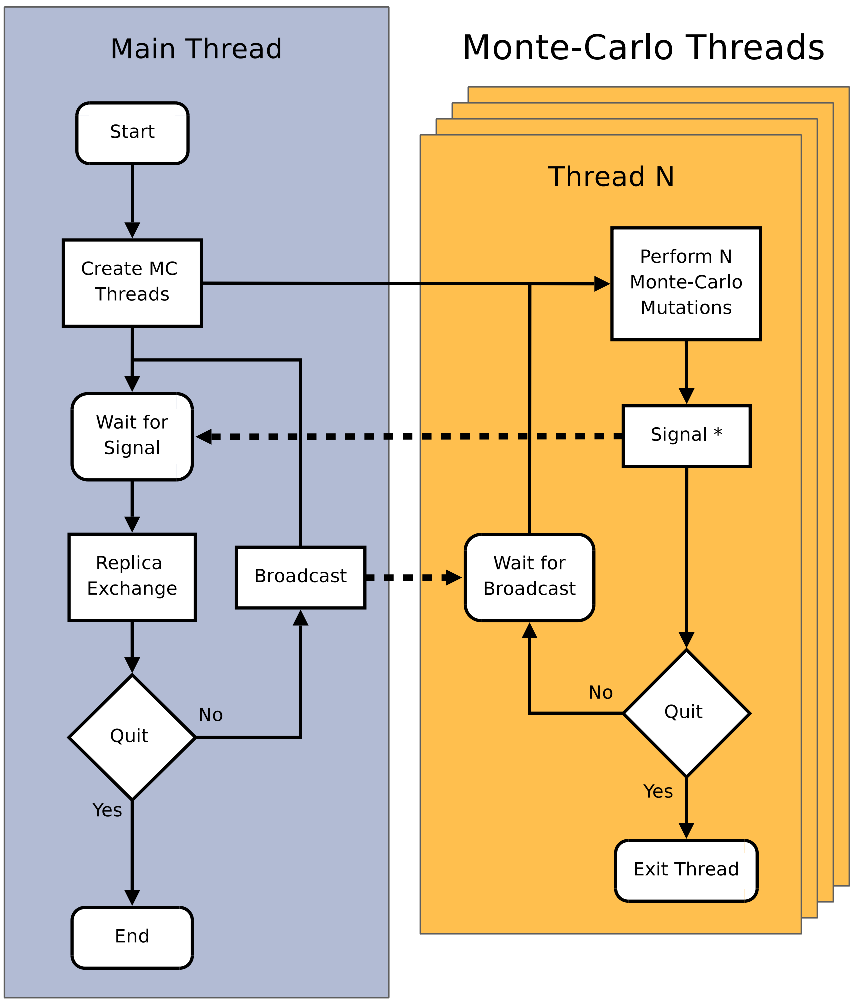


Figure 5.6: Multi-threaded Replica Exchange with Thread Reuse
The producer-consumer model used for multi-threading improves on the simple model illustrated in Figure 5.5 by reusing threads for the entire duration of the simulation. This implementation was necessary at the time of development due to a memory leak occuring on the first use of a CUDA call in a new thread context. With the previous model, a small amount of memory was irretrievably lost each time a thread was launched. This model requires the use of semaphores and signals for synchronisation, techniques not previously required due to the simplicity of the initial implementation.

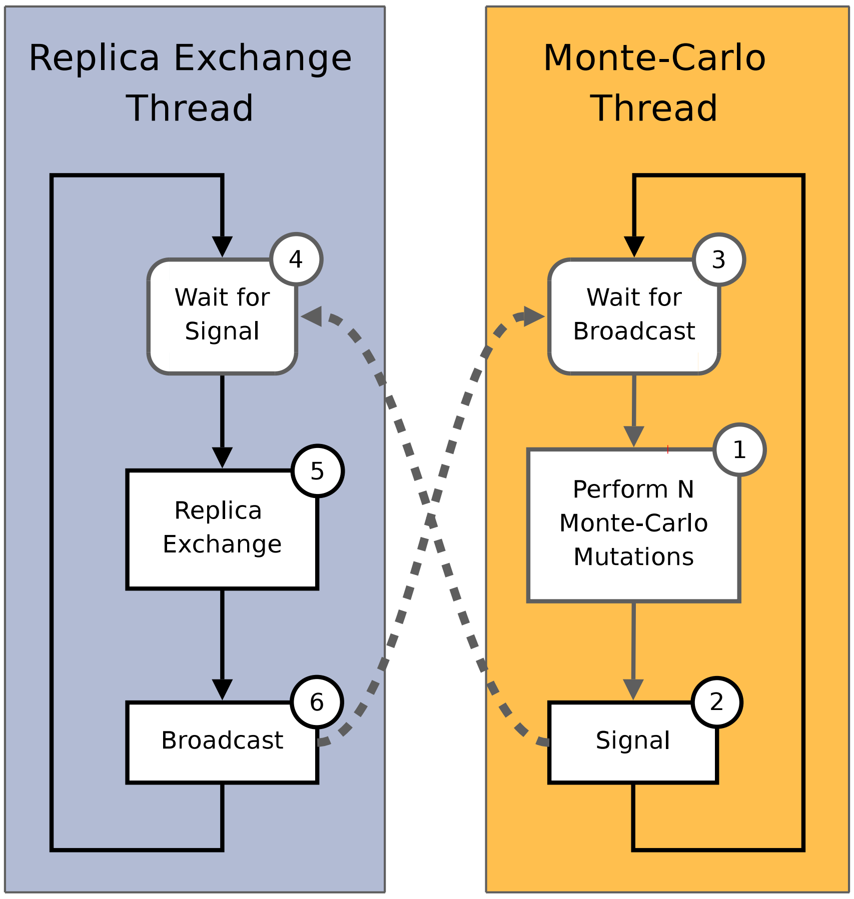


Figure 5.7: Mutexes and Replica Exchange
The thread reuse model requires that threads invoke replica exchange once all have completed their Monte-Carlo simulations. In this diagram, paths depicted in black can only execute if they own the simulation's mutex. This mutex is both owned and released at step 4 for replica exchange, ensuring that replica exchange only occurs once all Monte-Carlo threads are in a waiting state because this would otherwise corrupt the simulation data. By owning the mutex from state 2 to state 3, the Monte-Carlo thread ensures that the Replica exchange cannot continue until after it is waiting. State 1 does not require ownership of a mutex as the Monte-Carlo simulations in different threads can operate concurrently as no writeable data is shared between them.

5.3 GPU Implementation

The GPU in our implementation is responsible for calculating the interaction potential between all the molecules in a replica following the design from Chapter 4. Graphically, Figure 4.6 shows the key areas of implementation associated with the GPU.

For the GPU to complete its task of calculating the interaction potential between all molecules in a replica, the following tasks, represented by the blue blocks in Figure 4.6, are performed:

1. Initialise the CUDA runtime and initialise GPU data (initialise GPU replica Instance).
2. Transfer residue data to the GPU after a mutation (Update Replica on GPU).
3. Calculate the interaction potential using a CUDA kernel (Compute potential).
4. Copy the interaction potential from the GPU to the host (Copy potential from GPU to CPU).
5. Free GPU resources once complete (Free GPU instance).

The implementation details regarding these tasks are discussed in the sections that follow.

GPU acceleration was implemented in two iterations. The first iteration of GPU code merely affirmed that the application could execute successfully on a GPU. This implementation used a naïve version of the algorithm in a CUDA kernel, encoding the functionality of the points listed above without any significant optimisations.

Improvements in the second iteration of GPU development, adhering to the performance optimisation guidelines in the CUDA Programming Guide [22] and CUDA Best Practices Guide [23], increased speed-up significantly when compared to the initial GPU kernel. The introduction of performance enhancing features introduce a fair degree of complexity to the GPU solution and these features are thus discussed in later sections.

5.3.1 Performance Optimisation Guidelines

The NVIDIA CUDA Best Practices Guide states that performance optimisation revolves around three basic strategies [23]:

1. Maximise parallel execution
2. Optimise memory usage to achieve maximum memory bandwidth
3. Optimise instruction usage to achieve maximum instruction throughput

The first strategy, maximising parallel execution, involves structuring our algorithm such that as much of it can be executed in parallel as possible. Once restructured, the kernel parameters need to be tuned to achieve peak performance for a single kernel. Then, at a higher level, concurrent execution using streaming and concurrency between host and device execution need to be employed to achieve maximum performance. To summarise, implementation of the GPU code needs to maximise the degree of parallel execution, otherwise the 240 cores of the GTX280 will not be used effectively.

Optimising memory usage is best done by minimising the amount of data transferred between the host and GPU device. This is because the memory bandwidth between host (RAM) and device (GRAM) is slow relative to the bus speed between the CPU and RAM and the GPU and GRAM. This project uses an Intel Core 2 CPU. This CPU family is capable of 1.3 GT/s on a 64 bit memory bus resulting in a modest peak memory bandwidth of 10.6 GB/s. Newer CPUs, such as the Intel i7 can perform 6.4 GT/s, with a theoretical 51 GB/s of peak memory bandwidth. However, due to the current speed of memory the actual performance of a stock i7 and DDR3-1600 combination is 32 GB/s. GPU's further improve on this figure due to a bus width of 512 bits, meaning that an NVIDIA GTX280 (GT200) has a theoretical maximum memory bandwidth of 141 GB/s. In contrast to this, the PCIE bus connecting GRAM to RAM has a peak transfer rate of only 8 GB/s per second. From these figures, it is apparent that, from a memory bandwidth perspective, even if a GPU accelerated simulation significantly outperforms a CPU simulation, transferring data between GRAM and RAM on the PCIE bus will be a bottleneck. Structuring the implementation such that this bottleneck is hidden by other operations is vital.

The final strategy, optimising instruction usage is a trade off between speed and accuracy [22, 23]. CUDA implements a full range of mathematical functions, such as *exp*, *pow*, *sqrt*, etc. However, only add, divide, multiply and sqrt are IEEE compliant. The remaining commonly used functions are accurate to 1 unit of least precision (ULP) or in the case of trigonometric functions, 2 ULP [22]. These functions are further optimised on the GPU. Termed intrinsic functions, they are implemented in hardware to execute in fewer clock cycles than the standard functions at the cost of far greater ULP errors. This optimisation also involves choosing appropriate instances when the use of intrinsic functions is beneficial. There is no reason to use an intrinsic function if it results in a performance gain disproportionate to its loss of accuracy.

5.3.2 GPU Initialisation and Resources

The first step in using a GPU is initialising the device. Using CUDA for GPU acceleration requires adherence to the appropriate programming model to fully utilise the GPU and CUDA runtime environment.

Each process or thread must to be associated with its own instance of the CUDA runtime.

CUDA does allow for saving and loading of CUDA contexts, but like a posix thread, CUDA cannot access data dynamically declared outside its scope as it has no knowledge of the data. Consequently, each thread must call the CUDA initialisation function *cuInit* to instantiate an instance of the CUDA runtime for future use. All subsequent calls to CUDA from the thread are part of this context. Section 5.2, describing the multi-threaded aspects of the CPU implementation, showed that it is possible to run the code with any number of threads. Thus, a CUDA runtime is initialised in each thread before any CUDA API calls. The scoping of CUDA runtimes provides a convenient way of encapsulating data. Each thread is responsible for initialising its own CUDA runtime and managing the replicas associated with it.

When running a simulation, threads are created as stated in Section 5.2. After initialising a CUDA runtime, the lookup table of contact potentials, required to calculate the van der Waals component of the interaction potential, is initialised on the GPU. This is followed by the initialisation of GPU memory for all the residue data required for calculating the interaction potential. All memory is reserved for the entire duration of a thread's life. At the end of the simulation the device memory for the lookup table and the residues are freed.

The declaration of variables that use GPU resources from inside a Monte-Carlo thread scope also solves the multiple GPU problem and issues of distributing simulations across compute nodes in a cluster. In the case of multiple GPUs, the selection of the GPU is performed once at the beginning of the thread and never needs to be managed. In the case of a cluster of nodes, each with a GPU, each thread can run on a compute node in the cluster. Configuration of the hardware is thereby decoupled from the simulation because adherence to the one thread per CUDA context rule is enforced.

5.3.3 GPU Design and Data Structure

The mapping of a problem from its logical structure or object-orientated design to a problem suited to the GPU architecture is of key importance when using a GPU. Inspection of our algorithm indicates several apparent performance bottlenecks. The summation of all pair potentials is an $O(N^2)$ problem and by nature looking up the van der Waals interaction component of the potential is unpredictable, implying that it will be inefficient. The speed of the interconnect between GPU and host is also a bottleneck because of the need to update the GPU representation at each step of the Monte-Carlo simulation.

The data structures for the GPU can have a significant effect on GPU performance because of the first two points mentioned under the performance guidelines, namely: maximise parallel execution and optimise memory usage to achieve maximum memory bandwidth.

The data structures implemented for the CPU cannot be used directly on a GPU. The data sets passed to the GPU are fundamentally arrays of type *float, double, int, long, short, char* or their unsigned equivalents. CUDA also implements vector types derived from these basic types such as *float2, float3, float4, uint2, uint3, uint4*, etc. These vector types are merely *structs* of the basic types, where components are accessed by x,y,z and w variables. The advantage of using the vector types over basic types is that their byte alignment is explicit. This means that once allocated on the GPU, they are byte aligned optimally for parallel memory fetches. While it is possible to define new types on a GPU it is simpler to use the built-in types because this avoids issues with memory alignment and its effect on performance.

As mentioned in Section 5.1.1, residues in the system encapsulate the following data: position (x,y,z), type, charge and radius. The position is a set of 3 single precision floating point values representing x,y and z components. Type is an integer used as an index of the residue's amino acid type. Charge and Radius are also single precision floats.

The manner in which the CPU calculates interaction potential is bound predominantly by the number of cores. This necessitates the use of four nested loops, as illustrated in Figure 5.2, to sum the potential between all pairs. The 4 nested loops can be re-factored into two nested loops by removing the Molecule as a form of partition. The CPU loops over every molecule and compares it to all the other molecules and in turn the molecule pairs loop over all their residues. However, if the notion of a Molecule is removed and only residues remain, it becomes 2 loops, which, for each residue, loop through all the other residues and calculate the pairwise potentials.

While this re-factoring does not change the number of calculations required to perform the interaction potential calculation, but due to the thread level parallelism exposed by the GPU, this algorithm structure maps each interaction pair calculation to a GPU core, calculating each pairwise interaction in parallel without outer loops for each molecule.

However, removing the molecule abstraction causes a problem. The interaction potential is calculated between pairs of residues from different molecules. Therefore an additional variable identifying to which molecule a residue belongs is required. Thus each GPU residue representation encapsulates:

- Position (3 floats)
- Molecule (1 integer)
- Amino acid type (1 integer)
- Charge (1 float)
- Radius (1 float)

This structure is integrated into the existing *Replica* class. The following representation is adopted as it minimises the number of fetch requests to GRAM required to calculate the interaction potential for a single pair. Two arrays contain the residue data on the GPU, namely, *deviceResiduePosition* of type *float4* and *deviceResidueMeta* of type *float3*.

deviceResiduePosition[i] maps to the following components of the i^{th} residue:

- deviceResiduePosition[i].x → x component of position
- deviceResiduePosition[i].y → y component of position
- deviceResiduePosition[i].z → z component of position
- deviceResiduePosition[i].w → molecule to which the residue belongs

deviceResidueMeta[i] maps to the following components of the i^{th} residue:

- deviceResidueMeta[i].x → amino acid type
- deviceResidueMeta[i].y → electrostatic charge
- deviceResidueMeta[i].z → van der Waals radius

As previously mentioned, these arrays are initialised once at the start of the simulation. Device memory is reserved until it is explicitly freed or the thread to which it was allocated exits. Were this data initialised and transferred for every Monte-Carlo iteration it would be far more costly than merely updating it after each iteration. Furthermore, when only the molecule that has been mutated in the Monte-Carlo move is updated, we observed transfer to GPU to be an order of magnitude faster than initialising and copying the entire replica to the GPU each time. This in turn reduces the transfer time between host and GPU, decreasing the time it takes between mutating a molecule and completing its transfer to the GPU.

Packing data in this compact representation (of only 7 floats) also helps decrease the bandwidth required by the GPU. If one has to fetch data for 32 residues from global GPU memory, it is decomposed into 2 fetches, one of 32×4 bytes and one of 32×3 bytes in two contiguous blocks. It is crucial that the blocks are contiguous since this means that the fetch is more efficient than if it were interleaved with other data or bytes that act as byte alignment padding. [23].

Some variables, such as molecule index and amino acid type, are not floating point numbers, Although they are stored as such. Thus, separate arrays for molecule identification and amino acid type are not required. IEEE 754 compliance ensures that the conversion to and from float will not affect the values stored in these variables [173]. In the cases of molecule identification and amino acid type identification, integers in the range [0;19] and [0;N] are used to label residues.

GPU kernels are only able to access pointers to GPU memory within the current context. Thus, pointers to *float4* and *float3* arrays on the GPU are stored in the *Replica* object to which they belong and passed to GPU kernels on invocation. The position of each molecule in the array is stored in the *Replica* as this removes the need to recalculate it every time a molecule is updated and transferred to the GPU.

Identical device and host copies of the data are maintained such that no initialisation is required when packing and preparing data for transfer to the GPU. When GPU streams are used, this method of storing data allows for asynchronous host to GPU memory transfers because the data can be page locked (as discussed further in Section 5.3.6).

Data Padding

GPU kernels also impose restrictions on data for efficiency reasons. Kernels are launched as grids of thread blocks. This concept is explained more fully in later sections since it is important in implementing GPU kernels. With regard to data structures, a GPU kernel using grids of thread blocks can only launch grids with the same number of threads.

Were data to be of a size not divisible by the number of threads in the block, additional code for handling edge cases would have to be included in a kernel. For example, consider a kernel configured as 3 blocks of 64 threads to compute the sum of 187 elements. Elements are accessed using the block number, b, multiplied by the block width, 64, added to the thread number, t: $\mathrm{A}[b \times 64 + t]$. The first two blocks may proceed without any edge case handling as all values produced by $b \times 64 + t$ are valid elements of the array. However, the final block will attempt to access elements 187 to 191, which do not exist. This will cause a segmentation fault unless control code is added to check that the addresses are valid.

Introduction of code to check for the end of an array is detrimental for two reasons:

1. All threads are identical, requiring all threads to test if their element is in the array bounds. This results in additional computation.
2. It introduces a branch condition. SIMD architectures, like the GPU, cannot concurrently execute divergent branches of code, one half of the branch must first execute while the second half waits and vice versa. [22].

NVIDIA also recommends that the number of threads in a block be a multiple of 64 for optimal scheduling of threads on the GPU [22].

Padding the data on the GPU addresses the above-mentioned issues. Elements are added to the end of the GPU arrays such that the array length is a multiple of 64. This removes the need for code to check for out of bounds elements and avoids branching.

Figure 5.8 illustrates the effect of padding on the number of pairs produced while calculating the interaction potential. Although padding does introduce more computation, calculating 64 pair interactions versus 49 pair interactions as per the example in Figure 5.8, it does not cost

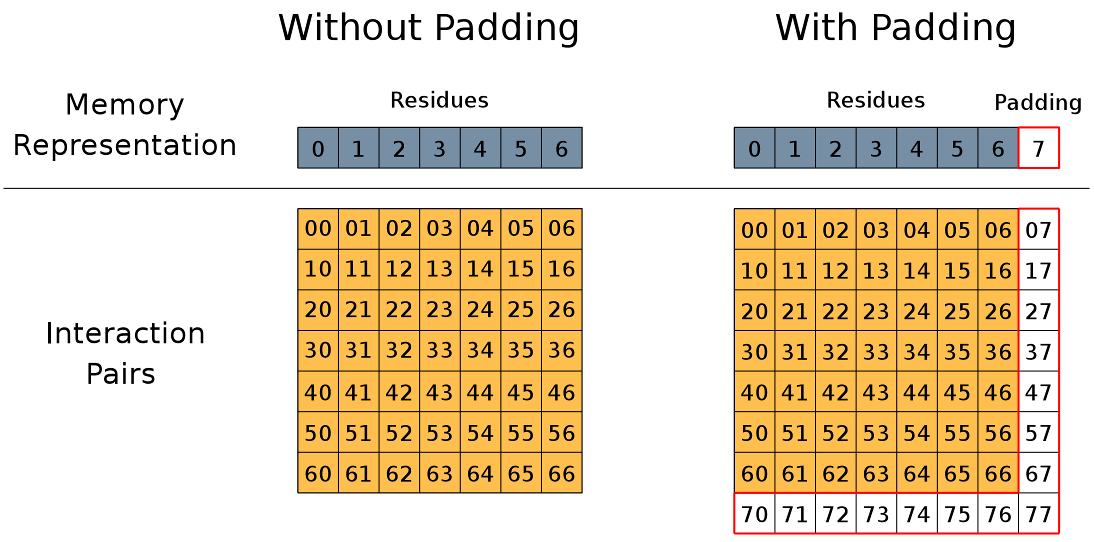


Figure 5.8: Parallel Execution and Padding *On the left, each interaction pair between the residues maps to a square matrix of size N^2. Once padded, elements are introduced to make the data size a power of two for efficiency and so that a parallel sum may be performed on the data. Elements 0,0 to 6,6 in the block contribute to the interaction potential while elements produced by padding, x,7 and 7,x, contribute zero. Because all pairs are generated simultaneously, padding elements remove the need to handle edge cases, which in turn makes the computation more efficient.*

more time because the computation is performed in lock-step with other threads of the same block.

This implementation identifies padding residues by setting *deviceResiduePosition[i].w*, the molecule identifier, to -1.

Parallel Reductions

On the CPU, a contribution of a residue pair to the overall interaction potential is accumulated as the algorithm proceeds, but if each pair interaction is computed in parallel, N^2 intermediate values need to be summed to determine the overall interaction potential. It is highly inefficient for N^2 concurrent threads to accumulate the overall sum as they proceed, because it will require code for frequent synchronisation. Were all threads attempting to concurrently read, modify and write to a single accumulation variable, unpredictable results would occur. All threads would read the same value, because they execute the same instruction at the same time in lock step, but write back different values. Atomic instructions prevent this from happening by ensuring that the read-modify-write operation is atomic, but would result in the operation becoming sequential because they lock the variable they modify for each thread in turn [22]. Consequently, the thread level parallelism afforded by the GPUs threading model is lost.

Assuming that each pairwise potential has already been calculated by the GPU, calculating the total interaction potential involves a summation of all the elements. In sequential form this is merely:

```
Data: M[x]: an array of values
for $i = 1$ to $N$ do
    $sum = sum + M[i]$
end
```

Adding together the elements of a matrix with N elements is $O(N)$ if done in this fashion. However, this can be done in $O(log_2 N)$ using a parallel sum reduction.

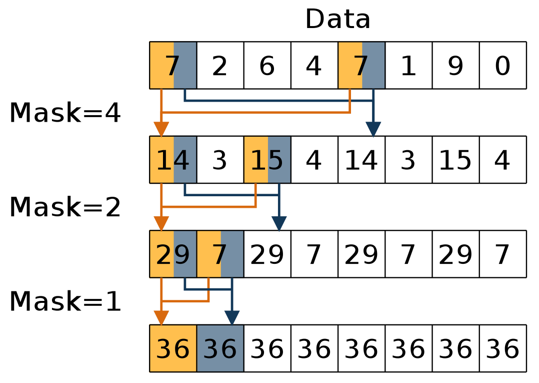
Parallel sum reductions can only sum datasets which are a power of two in size. Assuming N threads, each indexing element x of an array of length N, a parallel sum reduction works in the following way:

```
Data: M[x]: an array of values, indexed by x, the current thread index.
$mask = N/2$
while $mask > 0$ do
    $tmp = M[x] + M[x \bigotimes mask]$
    syncthreads()
    $M[x] = tmp$
    syncthreads()
    $mask = mask/2$
end
```

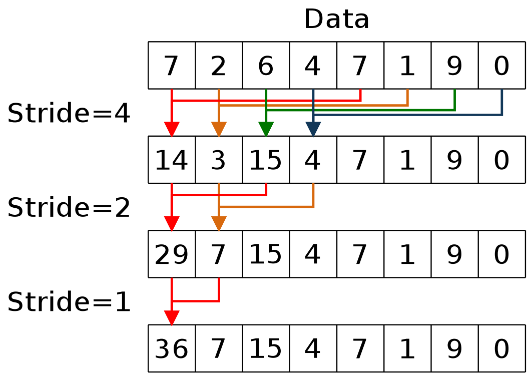
Threads operate in pairs on the values in the array. Before a pair can write back the results of addition, it must synchronise to ensure that no thread writes back before the other thread in the pair reads. Both threads then write back and synchronise again such that iteration is complete for all threads before beginning the next iteration for a different masking value.

The XOR ($\bigotimes$) of the array index, x and $mask$ results in two threads accessing distinct pairs of elements for each iteration. A thread always operates on a pair of elements at positions tx and $tx \bigotimes mask$. Because N is a power of two, the distance between the pair always halves until every elements contain a complete sum. A parallel sum reduction of eight elements is illustrated in Figure 5.9a. The figure shows that threads work in complementary pairs. Thread 0 and thread 4 operate on the same pair of elements in the first iteration because $4 \bigotimes 4 = 0$ and $0 \bigotimes 4 = 4$. In the next iteration threads 0 and 2 interact, and finally thread 0 and thread 1 to complete the sum.

Although 24 additions occur during the parallel sum, $log_2 8$ times more than a sequential sum, these additions are performed in parallel, resulting in $O(log_2 n)$ time versus the $O(N)$ time of the sequential sum. To make the reduction fast, efficient and minimise bank conflicts on the GPU, reduction is performed on elements in a shared memory on the GPU. Bank conflicts are



(a) Parallel Reduction



(b) CUDA Reduction

Figure 5.9: CUDA and CPU Reductions *(a) An array of eight elements requires only 3 iterations performed by eight threads to execute a parallel sum reduction on the values in an array. Thread* 0 *and its pair, thread* $0 \otimes$ *mask perform reduction in* $O(\log_2 n)$ *time, but* $O(n \log_2 n)$ *work. After the parallel sum reduction, each element of the array contains the sum of the values in the original array. (b) CUDA reduction implements the same algorithm, but in a work efficient* $O(\log_2 n)$*. The strided implementation also results in only the first element containing the reduced value, unlike using bitmasks (a), which place the reduced value in all elements.*

minimised by using a stride, as opposed to a bit mask. Work efficiency is achieved using n threads to reduce $2n$ elements for every iteration of the algorithm and half as many threads are used in each subsequent iteration. This optimisation results in fewer active warps per iteration if the number of CUDA threads per block is a multiple of 32 [82], but also results in only the first element containing the reduction result.

This type of reduction also minimises the amount of thread synchronisation required. Once only 32 threads are involved, one warp, no synchronisation is required because warp operations are synchronous with respect to each thread in that warp. Hence, 64 values can be reduced with no synchronisation, 128 with a single synchronisation, 256 with two synchronisations, etc... [82].

If the GPU is to accelerate simulations with many thousands of residues, the use of a parallel sum on the GPU to add the interaction pair results together is crucial due to the number of residue pairs. The specific use of the parallel reduction will be discussed with the kernel with which it is used, as this affects its implementation so some degree.

5.3.4 GPU Kernels

The GPU kernel must perform the task of calculating the interaction potential of a replica. This involves porting the CPU algorithm, listed in Figure 5.2, from a sequential algorithm to a data parallel algorithm. Initially, a naïve approach to the problem and was used as a proof of concept, mapping each pairwise calculation to a thread. While functional, the maximum speed-up achieved by this kernel peaked at approximately 60 times that of the basic CPU implementation. The final implementation of the kernel uses the tiled MD methods of Friedrichs et al. [17] and Meel et al. [26] with a parallel reduction to produce a partial sum array of interaction potentials on the GPU.

Specifically, a kernel must accumulate each interaction potential pair between a residue and all other residues, executing the code in Figure 5.10 each time for every distinct pair of residues in a replica before summing them to calculate the overall interaction potential.

For the NVIDIA GTX280 a maximum of 65536 thread blocks may be launched for a single kernel. A grid defining these thread blocks may be configured in any way the programmer wishes by setting the *gridDim* variable of type *dim3(x,y,z)* equal to *dim3*([1:65536],[1:65536],1) provided the product of the dimensions does not exceed 65536. Each thread block may contain up to 512 threads. An instance of *dim3*([1:512],[1:512],[1:64]) defines *blockDim* with the product of the dimensions not allowed to exceed 512.

GPU kernels can only read and write to memory in GPU memory. This means all data must already be on the GPU before a kernel can execute. The data structures discussed in Section 5.3.3 are passed to the kernels as pointers to GPU memory, such that each kernel has the following arguments:

float4 *ResiduePosition Array of residue positions and molecule associations packed by the host into global memory.

float3 *ResidueMeta Array of meta-data for each residue consisting of amino acid type,

```
Data: M[m]: Molecule $m$ of a replica instance with $N$ molecules. M[m][r] denotes residue
      $r$ of molecule $m$.
Data: LJ(i,j): Lookup table for short range residue interactions.
$r \leftarrow eps+ \parallel M[m_i][r_i].position - M[m_j][r_j].position \parallel$
$e_{ij} \leftarrow \lambda(LJ(i,j) - e_0)$
$\sigma_{ij} = (M[m_i][r_i].radius + M[m_i][r_i].radius)/2$
$u_{ij} \leftarrow -4 \cdot e_{ij} \cdot {\sigma_{ij}}^6({\sigma_{ij}}^6 - 1)$
if $e_{ij} > 0$ and $r < \sqrt[6]{2} \cdot \sigma_{ij}$ then
    $u_{ij} \leftarrow -u_{ij} + 2e_{ij}$
$u^{el}_{ij} \leftarrow M[m_i][r_i].charge \cdot M[m_i][r_i].charge \cdot e^{-r/\xi}/r$
$U_{ij} = u_{ij} + u^{el}_{ij}$
```

Figure 5.10: The Interaction Potential Kernel
This code fragment is most basic unit of computation in the model. This code must execute for each distinct pair of residues. U_{ij} is the interaction potential for a pair of residues and must be summed for all pairs to produce the total interaction potential of a replica.

charge and van der Waals radius.

float *LJPotentials The table of contact energies for calculating the van der Waals pairwise interaction.

float *Result An array for storing the result of each thread block.

CUDA provides built in variables to retrieve information regarding the thread block and thread currently executing. These variable are used to index elements of the input data. Each of these built-in variables is a three-dimensional vector containing x, y and z dimensions of the structure or index they represent. Using combinations of these built in variables allows one to access the elements of the input data in an appropriate manner.

Using a thread to perform multiple pairwise potential calculations, as opposed to one comparison per thread, means that each thread does more work and the kernel requires fewer threads to calculate the total interaction potential. This also implies that there is a higher ratio of work to scheduling and fewer thread blocks per kernel are required. This technique allows thread blocks to cache data in shared memory explicitly, reducing the latency of each threads residue data lookups.

A single thread associates itself with a particular residue (Figure 5.11), R[$bx \times blockDim + tx$], according to its block index, *bx*, thread block size, *blockDim*, and its thread index, *tx*. It then calculates pairwise potentials between this residue and *blockDim* other residues, R[$by \times blockDim$] to R[$by \times blockDim + blockDim$], accumulating the results in shared memory. Shared memory results are then summed the optimised CUDA reduction and the global array of partial sums is updated with a thread block's pairwise potential sum.

A block begins with each thread reading its *bx* referenced residue into local memory on the symmetric multiprocessor. This makes all future accesses to this data as fast as accessing a reg-

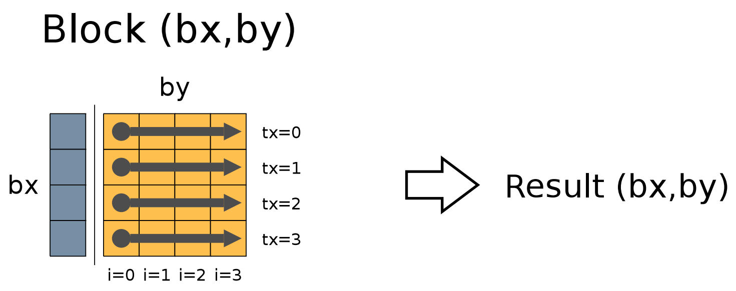


Figure 5.11: A Tiled Thread Block
In a given thread block, Block(bx,by), each thread is assigned a residue in the bx subset of residues according to its thread index, tx. This thread, in lock step with the other threads, loops though the subset by and computes the pairwise potential for its element on each element of the by subset. The results of each thread are again reduced to a single partial sum using a parallel reduction and stored in the appropriate position in the Results array in global memory on the GPU.

ister. Local memory also stores all the variables required to perform calculations with thread, tx, such as accumulating the pairwise potentials. Each thread in the block then reads, in lock-step, the same by referenced residue from memory in each iteration until all pairwise potential calculations for that block are complete.

Copying residues for a thread block from global memory into shared memory at the beginning of that thread block also decreases the bandwidth required by a kernel when looping though residues. A thread block must loop though all residues from index $by \times blockDim$ to $by \times blockDim + blockDim$. Global memory reads suffer a latency of 400 and 600 clock cycles per read [22] so if each residue is read from global memory for each iteration of the loop, *blockDim* global reads of 7 bytes would be performed, each suffering significant latency. But using shared memory reduces this to one global memory read of $blockDim \times 7$ bytes, as depicted in Figure 5.12, achieving optimal memory usage pattern [22]. This is also the approach taken by Friedrichs et. al. [17] and Anderson et. al. [25] to perform molecular dynamics simulations and Bellemen et. al. [16] to perform N-body simulations. Figure 5.13 illustrates the pattern in which the residues are stored in shared memory. Bank conflicts do not occur because all threads in a block read the same residue data in shared memory in lock step.

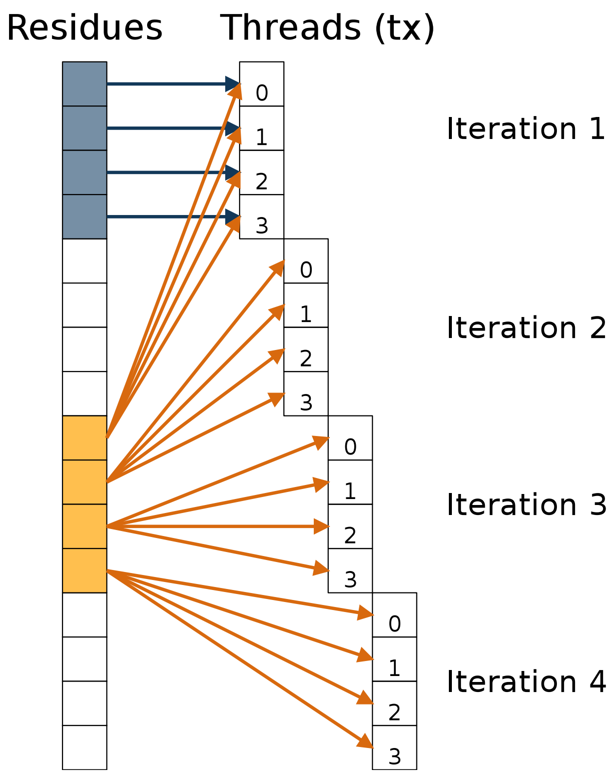


Figure 5.12: Thread Access Patterns
Threads iterate though the set of residues in shared memory. Because all threads access the same data in shared memory in lock step, this results in blockDim accesses, serving the blockDim^2 requests due to the broadcast capability of shared memory.

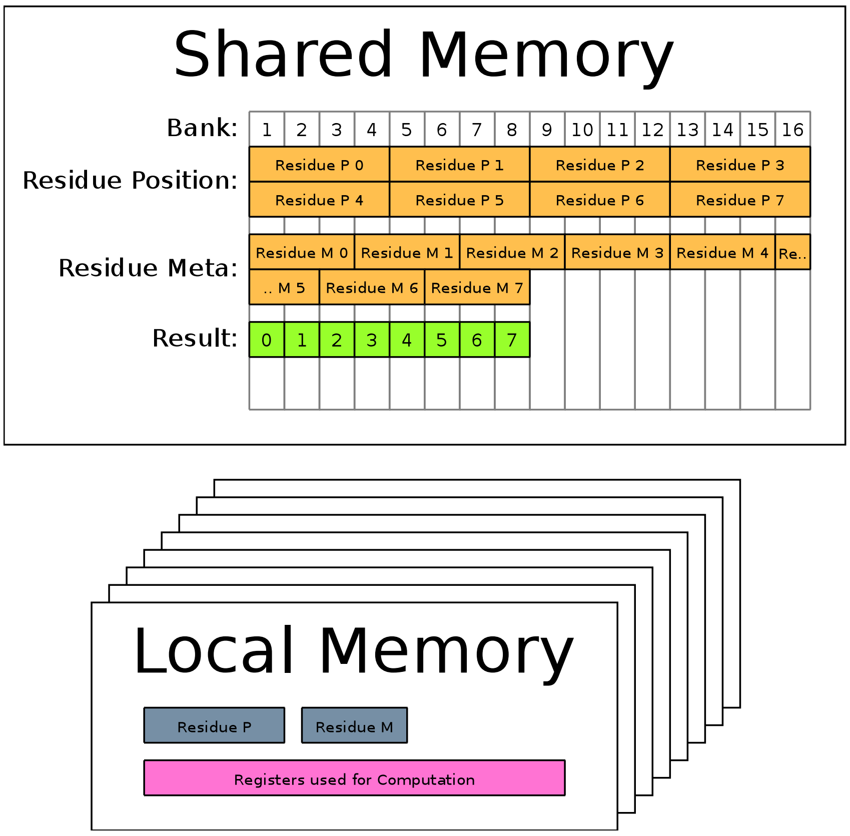


Figure 5.13: Memory Allocation per Thread Block *Each thread in the a thread block stores a copy of its residue and its interaction potential accumulator in local memory. Shared memory contains data from residuePosition and residueMeta that each thread accesses in lock-step. There are no bank conflicts when reading residue data because all threads access the same bank simultaneously. Results for each thread are stored in shared memory, with each thread assigned a different bank of shared memory for its result up to 16 threads before threads begin to share banks, resulting at in 2 way bank conflicts for a warp.*

Intermediate results for each thread block are stored in shared memory. An array, *Results*, is declared of size *gridDim* and each thread uses it to accumulate the pairwise potentials it calculates. These values are stored according to thread index. Because threads are indexed 0 to $gridDim - 1$, the 32-bit floating point values in *Result* are assigned to contiguous banks in shared memory (Figure 5.13). Once again, this avoids bank conflicts because no two elements share a bank. If there are more than 16 threads per block this is still the case because shared memory transactions are schedualed in half-warps of 16 threads each [22].

Finally, once the threads in a block have iterated though the residues in shared memory, the contribution to the overall interaction potential for that block is calculated using a parallel reduction of the elements in shared memory. Upon completion a kernels partial sums are summed on either the GPU or host to calculate the overall interaction potential.

The thread and block indexing method results in this kernel being able to address up to

$\sqrt{65536}$ distinct blocks of residues where each block contains up to 512 residues. In total, this allows the kernel to access up 131072 residues. Multiple kernels will only be needed to calculate the interaction potential between more than 131072 residues, adding an additional 2 outer loops to the algorithmic representation in Figure 5.2.

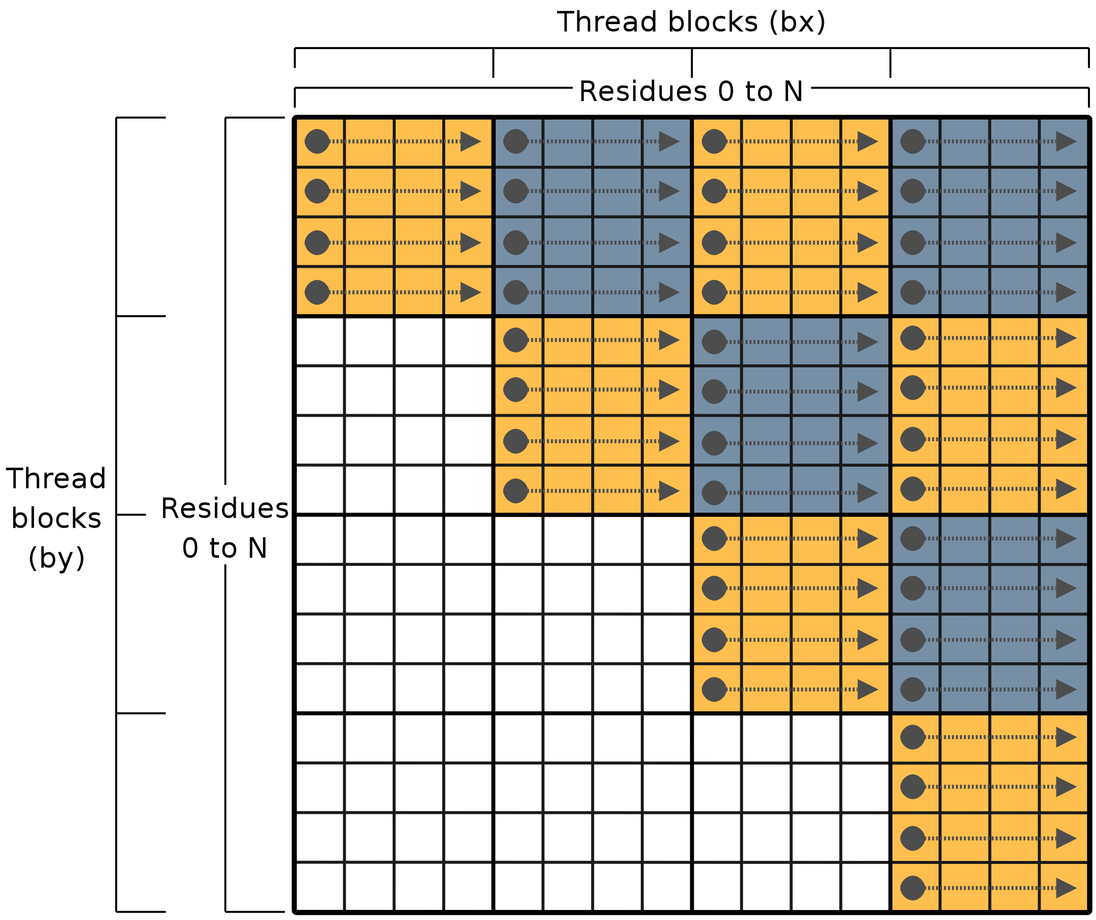


Figure 5.14: A Shared Memory Kernel Grid *Each thread calculates the potentials between one residue and all other residues in the same block's shared memory. Reusing threads to calculate the potential from multiple pairwise interactions between residues stored temporarily in shared memory exploits the speed of shared memory over the high latency of global memory.*

Thus far, discussion has focused on the manner in which kernels subdivide and manage simulation data. This only satisfies the part of the optimisation guidelines, discussed in section 5.3.1. The final guideline, optimising instruction usage to achieve maximum instruction throughput, will be discussed in the following sections. In optimising instruction usage, the algorithmic restrictions of the model determine the degree of optimisation.

5.3.5 Algorithmic Restrictions

For the kernel, the two components of the interaction potential between a pair of residues must be calculated, these components being; the van der Waals potential:

$$u_{ij}(r) = \begin{cases} 4|\varepsilon_{ij}|[(\sigma_{ij}/r)^{12} - (\sigma_{ij}/r)^6], & \text{if } \varepsilon_{ij} < 0 \\ 4\varepsilon_{ij}[(\sigma_{ij}/r)^{12} - (\sigma_{ij}/r)^6] + 2\varepsilon_{ij}, & \text{if } \varepsilon_{ij} > 0, r < r_{ij}^0 \\ -4\varepsilon_{ij}[(\sigma_{ij}/r)^{12} - (\sigma_{ij}/r)^6], & \text{if } \varepsilon_{ij} > 0, r \geq r_{ij}^0 \end{cases}$$

and the electrostatic potential:

$$u_{ij}^{\text{el}} = \frac{q_i q_j exp(\frac{-r}{\xi})}{4\pi D r}$$

Both equations are dependent on the Euclidean distance, r, between the two interacting residues. Calculating r stands to benefit from the GPU special function unit as $r = \sqrt{(p_1 - p_2) \cdot (p_1 - p_2)}$. Since all values are stored as 32-bit floats, the GPU *sqrtf* function is used. There is always the possibility that r could be zero. To avoid checking this condition in order to prevent division by zero from occurring $\epsilon = 10^{-38}$, is added to r before any calculations dependent on r are performed. This value of ϵ is chosen because nearest value to zero that a 32-bit floating point number can represent is 2^{-126}, approximately 10^{-38} [173]. Expected values for r are greater than 1 because the amino acid course grains have radii of approximately 5Å. Consequently, the addition of 10^{-38} will be truncated to the original value for values in the expected range for r. If r were zero, division by ϵ occurs and the subsequent introduction of NaNs, is avoided.

Calculation of the van der Waals potential requires that branching be introduced into whatever kernel is used. In its original form there are three branches, but closer inspection reveals that there is a common component to the equation and that it can be simplified to two branches. Firstly, the contact potential interaction between the residues, ε_{ij} (discussed further in section5.3.5), can be stored in a register after being calculated. The three part equation can then be calculated in two steps with one branch instruction, where only one of the branches has any instructions. This is done by first calculating:

$$u_{ij} = -4\varepsilon_{ij}[(\sigma_{ij}/r)^{12} - (\sigma_{ij}/r)^6]$$

Then, updating the value of u_{ij}:

$$u_{ij} = -u_{ij} + 2\varepsilon_{ij}, \text{ if } \varepsilon_{ij} > 0 \text{ and } r < r_{ij}^0$$

The first and third parts of the van der Waals component of the potential are identical because if $\varepsilon_{ij} < 0$ then $4|\varepsilon_{ij}|$ is equivalent to $-4\varepsilon_{ij}$, meaning that the first and third parts of the formula can be combined, becoming: $-4\varepsilon_{ij}[(\sigma_{ij}/r)^{12} - (\sigma_{ij}/r)^6]$. Since ε is stored in a register, the second part of the calculation the involving the addition of $2\varepsilon_{ij}$, can be calculated without a lookup.

The reduction in the number of branches and the elimination of all instructions for one half of the branch favours the GPU [22].

The electrostatic part of the potential:

$$u_{ij}^{\text{el}} = \frac{q_i q_j exp(\frac{-r}{\xi})}{4\pi D r}$$

suits implementation on the GPU as it requires 3 variables, q_i, q_j and r together with constants ξ, π and D: The charge of one of the residues, q_i, is resident in local memory and q_j is in shared memory. r has meanwhile been calculated and its value is in a register. Finally, fetching q_j is efficient because it is a single floating point value in shared memory, accessed by all the threads in a block at the same time. Consequently, this function maps well to the prescribed GPU programming models and access patterns [22].

The use of the GPU's intrinsic functions, ___*powf* and ___*expf* to perform the calculation of $(\sigma_{ij}/r)^6$ and $exp(\frac{-r}{\xi})$ further decrease the computation time required to calculate the potential between a pair. While intrinsic functions are less accurate than their non-intrinsic counterparts [22], the fact that some input values for the simulations have, at most, 4 significant figures of precision seems to make the simulation largely insensitive to the inaccuracy of the intrinsic functions, causing the GPU results to be identical irrespective of whether intrinsic functions or non-intrinsic functions are used. However, use of the intrinsic functions results in significantly better speed-up.

The Contact Potential Look Up Table

The optimisation of the arithmetic operations used for a pairwise potential calculation and the manner in which residues are accessed to reduce latency addresses the majority of the issues pertinent to an N-body simulation. What makes the model we implement different from a regular N-body problem is that the interaction between residues is dependent on pre-calculated values independent of the data encapsulated in each residue. The van der Waals component of each pairwise potential uses these predetermined values to calculate ε_{ij} which determines whether or not residues attract or repel each other and the relative strength of the interaction. These values, ϵ_{ij}, are be read from a table (attached as Table A.2) and calculate:

$$\varepsilon_{ij} = \lambda(\epsilon_{ij} - \epsilon_0)$$

with scaling and offset parameters λ and ϵ_0 from from Kim and Hummer [12]. ε_{ij} is then used to determine the van der Waals potential between two residues, $u_{ij}(r)$ (Equation 4.2).

Fetching ϵ_{ij} requires a lookup in memory that is random because each pair of residue types has a unique value, meaning that there are 210 possible values of ϵ_{ij} for any one pairwise interaction. The random nature of this operation is likely to degrade performance in a kernel because the memory access pattern will not map optimally to any well ordered memory access pattern is optimised in hardware. Additionally, random access operations on a GPU are undesirable, because of the GPU's SIMD like architecture. In a worst case scenario for these simulations, if there are 20 residues, all of a different type and in a random order, there is no mapping to SIMD. For all threads to read memory in parallel, the memory address of ϵ_{ij} needs to be a function of thread index. Consequently, memory reads of ϵ_{ij} become serialised because the randomness of their location in memory.

To minimise the complexity of implementing the lookup function, the GPU implementation of the table containing ϵ_{ij} is an linear array of 400 32-bit floating point numbers. The 2D lookups are mapped to a 1D array such that $\epsilon_{ij} = A[row + 20 \cdot column]$. As with the CPU, data is duplicated so that a lookup $A[x][y]$ is equivalent to $A[y][x]$ making the ordering of the residues is unimportant. Depending on now padding elements are used in the calculation of interaction potential, the array may need to be larger. One strategy for padding elements is to set the amino acid type identifier of the padding residues to 20, effectively creating an new type of amino acid with zero contact potentials and zero charge. This means that the table would be 21 by 21 elements linearly arranged such that if x or y is equal to 20 it returns zero. The alternative option for implementation padding residues is to test whether they are padding elements and then to skip any computation using an empty divergent branch.

Texture memory ultimately suits the usage model described above and the impact of choosing it as the memory container for the lookup table is reflected in our speed-up results. The choice of texture memory comes from one of four memory types on the GPU, each of which has its advantages and disadvantages. The four options for memory are:

Global Memory Global memory has the highest latency of all the GPU memory and is not cached. Accesses to global memory is only advantageous if many values are fetched at the same time.

Shared Memory Shared memory is on the GPU die and can be accessed very quickly. However, there is only 16K of shared memory and this needs to be used by the GPU kernels for storing residue data and results. Using 1600 bytes of shared memory will reduce the memory available for kernel images, thereby reducing the thread level parallelism if that memory is required due to the parametrisation of the kernel.

Constant Memory Constant memory is stored in the same place as global memory, but cached on first use, for the duration of a thread block. There is 64K of constant memory. Caching means that the first read from constant memory costs the same as a read from global memory, but subsequent reads of the same value cost the same as a read from local cache which is as fast as shared memory.

Texture Memory Texture memory is cached. However, texture memory differs from constant, shared and global memory in that it is not as sensitive to random access patterns. Its caching of data near values already retrieved also means that it can afford operations that do not conform to good access patterns. [22]

Global memory will perform the worst of all memory available for the lookup as it has the highest latency. But, if there are enough threads running concurrently they will hide this latency to a some extent. Constant memory will not perform any better than global memory for this application, unless the same lookup value is used over and over again. The reuse of a subset of contact potential would benefit from its caching mechanism. But, because any one of 400 values could be used at any time by any one of the currently executing threads, the performance of constant memory is likely to be variable because of many situations being unable to exploit the constant memory cache. Additionally, the constant cache will only exist for the duration of a thread block, meaning that future thread blocks will not benefit from the caching affect of previous thread blocks. Considering that it is stored in the same place as global memory, its performance in this case can thus be considered comparable to using global memory.

Shared memory is on chip and is therefore much faster than global and constant memory. However, it is only this fast because of its banks and their layout, which provide access speed equivalent to that of registers [22]. Consequently, shared memory is fastest when accessed in a coalesced pattern. By using shared memory, our lookups would not match the access pattern required for the performance shared memory provides. Furthermore, the lifetime of shared memory is again only that of the thread block in which it is used, requiring it to be reinitialised every time a thread block begins execution.

Texture memory provides the most promising mechanism for implementing the lookup table. CUDA implements textures by mapping a texture onto an array in global memory. The original data is then accessed via the texture lookup functions. Accessing the texture causes the texture units on the GPU to fetch the data requested and 16K of data surrounding that location. This means that all lookups to texture memory will be cached after the first lookup because the size of the texture is only 1.6K.

We have implemented all of these options for lookup table storage and the performance of each one analysed in Chapter 7. Choosing the particular memory to use for the lookup table is determined at compile time using C preprocessor macros.

With regards to programming, using constant or global memory is identical, with the exception that constant memory requires the inclusion of the identifier `__constant__` before the variable declaration. In the case of texture and shared memory, the data is copied to global memory before being transferred to shared memory or bound to a texture. Shared memory requires that the kernel copies the data from global memory into shared memory at the beginning of each thread block. The threads then need to synchronise to ensure that all memory is loaded before continuing with their execution. For texture memory, a texture object is declared and bound dynamically to the memory assigned to the lookup table using the CUDA call *cudaBindTexture*. Because CUDA supports several texture types, the texture is defined using a template of the form `texture<float,1,cudaReadModeElementType>`. The parameters correspond to the type of data (*float*), the number of dimensions (*1*) and the read mode (*cudaReadModeElementType*). With the read mode set to *cudaReadModeElementType* the texture coordinates can be referenced using integers instead of normalised floating point numbers.

5.3.6 Asynchronous GPU Computing

Thus far, the performance gains of our kernels are dependent on the optimal instruction and memory usage individual CUDA threads. Using blocking CUDA calls mean that once a kernel is invoked or a memory transfer between the CPU and GPU is performed, the CPU must wait for the GPU or GPU memory manager to complete its task before being allowed to continue. Without asynchronous execution, either the CPU or the GPU must wait idle for the other to execute, decreasing throughput.

The aim of using both the GPU and CPU asynchronously is to exploit as much latency hiding as possible by overlapping computation with memory operations. This is done by allowing both the host and the device the opportunity to execute concurrently.

The CPU and GPU are allowed to execute concurrently because asynchronous CUDA functions are non-blocking, passing control back to the host immediately when called. Thus, execution can be overlapped between GPU and CPU where possible.

CUDA terms this mechanism for asynchronous execution *CUDA streams*. Each GPU has 16 streams associated with it. A stream is essentially a job queue with a maximum capacity of 16 instructions. Asynchronous calls populate the stream, behaving in a non-blocking manner, until the stream becomes full. Once full, further asynchronous calls become blocking until items are cleared from the stream, at which point they again become non-blocking [22].

One critical feature of CUDA streams is that they guarantee, for a particular stream, that instructions are processed in the same order in which they were queued, even though they are processed asynchronously with respect to other streams and the host. [22]

The GPU dependent parts of the algorithm for performing Monte-Carlo simulations can be separated into the following distinct steps for each replica in a worker thread:

1. Update the mutated molecule in GPU global memory.
2. Calculate an array of partial interaction potentials.
3. Reduce the array of partial interaction potentials.
4. Write back the final interaction potential.
5. Restore the mutated molecule if the mutation is rejected.

Each subsequent step requires the previous step be complete before continuing. Of all of these tasks, the second one, calculating the partial interaction potentials, takes longest to return, blocking the calling thread from performing any other work while it executes. While the CPU is performing tasks between these steps, the GPU is idle. Because of this, the system is not being fully utilised since the use of blocking (synchronous) calls means that either the GPU or the CPU is working at any one time, never both.

```
Data: $R(t)$: The set of $N$ replicas at step $t$
Data: $M_i(x)$: Molecule $x$, of replica $i$
for $k = 1$ to $N$ do
    for $s = 1$ to $\phi$ do
        $u \leftarrow U_{\aleph}(1, N_M)$
        $m \leftarrow M_k(u)$
        $R_k \leftarrow Mutate(R_i, u)$
        $\Delta E \leftarrow E(R_k(s)) - E(R_k(s-1))$
        if $\Delta E > 0$ and $U(0,1) > e^{\frac{-\Delta E}{RT}}$ then
            $M_i(k) \leftarrow m$
    end
end
```

Figure 5.15: Sequential Monte-Carlo Simulation
The inner loop of the Monte-Carlo simulation performs all the individual Monte-Carlo steps one after another in a sequential manner for a single residue, before the same simulation is performed for the next residue. This effectively means that replica N_i blocks replica N_{i+1} from executing until it is complete. Provided the steps within each replica are in the correct order, sequential ordering of the replicas is unnecessary, suggesting changes in the algorithm to allow for the use of streaming.

A logical thread of execution in this case is a Monte-Carlo simulation. Without streams, a posix thread performs each Monte-Carlo simulation sequentially. Because these simulations are data independent, the order in which they are executed and the duration of execution do not affect their respective simulations. Figure 5.15 illustrates the algorithm used to perform N Monte-Carlo simulations of ϕ steps each on N replicas. An outer loop iterates over all replicas, performing the simulations one after the other until all the required simulations are complete.

Using multiple CPU threads to execute the Monte-Carlo steps concurrently is possible if there is more than one GPU. As discussed in Section 5.2, the mapping of one thread per run-time context and preferably one thread per GPU to avoid context switching results in this approach performing unfavourably on a single GPU. Additionally, one GPU can only execute one kernel at a time, resulting in the threads blocking each other.

Introducing streams enables the execution of up to 16 instances of the Monte-Carlo simula-

```
Data: $R(t)$: The set of $N$ replicas at step $t$
Data: $M_i(x)$: Molecule $x$, of replica $i$
for $s = 1$ to $\phi$ do
    for $k = 1$ to $N$ do
        $u_k \leftarrow U_{\aleph}(1, N_M)$
        $m_k \leftarrow M_k(u)$
        $R_k \leftarrow Mutate(R_i, u)$
        $\Delta E_k \leftarrow E(R_k(s)) - E(R_k(s-1))$
    end
    for $k = 1$ to $N$ do
        if $\Delta E_k > 0$ and $U(0,1) > e^{\frac{-\Delta E_k}{RT}}$ then
            $M_i(k) \leftarrow m_k$
    end
end
```

Figure 5.16: Interleaved Monte-Carlo Simulations
Switching the inner and outer loops of the sequential Monte-Carlo algorithm results in all Monte-Carlo steps for all replicas being interleaved. This partitions them into 2 distinct parts. The first part mutates the molecules and calculates the interaction potential and the second part evaluates the mutation.

tion within the same CUDA context on a single GPU. Multiple streams do not block each other because kernel operations are queued due to the asynchronous nature of the streams.

By interleaving the steps required for performing the Monte-Carlo simulations, the algorithm, from a logical perspective is identical, although the structure is unusual. This reordering is illustrated in Figure 5.16, and is performed by switching the inner and outer loops of the algorithm. Now, instead of iterating over all replicas and for each one performing Monte-Carlo steps, one iterates through all Monte-Carlo steps, performing the same step for each replica in two parts. The evaluation of the new interaction potential is the operation that takes the longest to perform. If it is placed at the end of a loop and made into an asynchronous call, then the Monte-Carlo mutation for the next replica may be performed on the CPU while the interaction potential is computed on the GPU. This results in keeping both the GPU and CPU busy and thereby increases the throughput of the system in terms of FLOPS, which ultimately results in a faster runtimes for the simulation.

CUDA places certain restrictions on host data that has asynchronous operations performed on it, requiring, in particular, that it be page locked. This means that the host's memory manager may never swap that data out of system memory and write it to paged memory because this would result in unpredictable behaviour [22]. Page locking memory for asynchronous transfer also increases the data transfer rate of that data between the CPU and GPU [23]. One undesirable consequence of page locked memory is that the simulation must be able to fit entirely

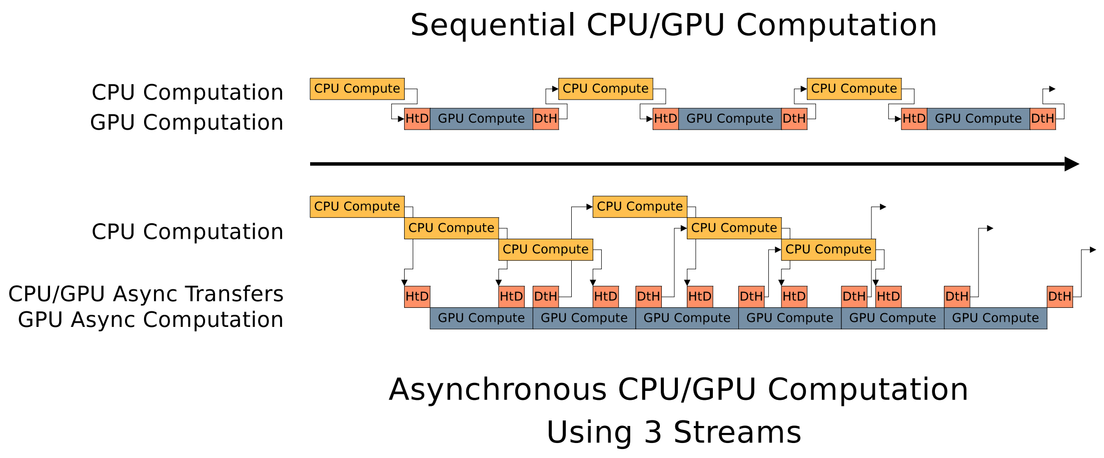


Figure 5.17: GPU Streams *Streams overlap CPU and GPU computation. At the top, 3 iterations of a simulation are executed on a single replica using blocking, synchronous, GPU calls. The CPU has to wait for each call to return before it can continue, meaning that CPU time is wasted. Using streams, replicas are interleaved. This diagram illustrates the use of 3 streams operating on three replicas using asynchronous calls for memory transfers and computation. Even though the GPU can only perform one kernel execution or memory transfer at a time, the fact that they can work concurrently results in the CPU and GPU performing more work than the synchronous algorithm in the same amount of time, this work being divided across more than one replica. DtH and HtD refer to device to host and host to device transfers respectively.*

unpaged in both RAM and GRAM. Given the way residues are represented in our implementation, it would require over 700 000 residues to use a gigabyte of memory. This, combined with the coarse-graining, means the number of residues is unlikely to ever reach this number and, consequently, no measures are currently taken to accommodate this many residues.

The steps of the Monte-Carlo simulation, from the perspective of a single replica, must be executed sequentially. It is therefore logical to map one replica to one CUDA stream as the instructions in a stream are guaranteed to be executed in the order that they have been called. The data independence of the replicas means that this suits the stream model because the instructions in multiple streams do not necessarily execute in the order they were queued, only their order in each stream [22]. This reordering is illustrated in Figure 5.17, where three streams execute concurrently. Here, it is possible to see that, given that the algorithm profile is appropriate, streams significantly increase the number of Monte-Carlo steps that the simulation can perform in a given time due to the overlapping computation of the GPU and CPU. We measure the effect of streaming on our implementation's performance in section 8.1.1.

Logically, as the number of residues in a replica increases, the benefit of streaming will become more apparent since the time taken for calculating the interaction potential will inevitably increase, whereas, if all molecules remain approximately the same size, CPU computation time per Monte-Carlo step will remain almost constant.

The number of streams the CUDA runtime must use is determined from an input parameter for the simulation. Any number of streams may be used from 1 to 16 and threads dynamically assign replicas to a stream. The stream associated with a replica is determined by the thread index, *tx*, modulo the number of streams per thread, *spt*. The number of replicas per stream, *rps*, is calculated as: replicas in the thread divided by the streams available, rounded up to the nearest whole number. For this reason, it is generally better for the number of replicas per thread to be a multiple of the number of streams. Our algorithm for using streams is listed in Figure 5.18, using the aforementioned method to divide replicas between streams to perform the Monte-Carlo simulations.

```
Data: $R(x)$: The set of $N$ replicas at step $t$
for $t = 1$ to $\phi$ do
    for $index = 1$ to $N/rps$ do
        for $r = 1$ to $rps$ do
            mutateMolecule($R[index \times rps + r]$)
            updateGPU($R[index \times rps + r]$)
            calculatePotential($R[index \times rps + r]$)
            writeBackResult($R[index \times rps + r]$)
        end
        for $r = 1$ to $rps$ do
            syncStream($r$)
            evaluateMutation($R[index \times rps + r]$)
        end
    end
end
```

Figure 5.18: Streamed Monte-Carlo Simulations
Each Monte-Carlo step of the simulation is divided into two parts. The first part mutates the molecule on the CPU, the CPU's most computationally intensive task, followed by an asynchronous transfer to the GPU together with an interaction potential calculation. Streams are synchronised before the second part of the Monte-Carlo step to ensure that the new interaction potential is ready before acceptance or rejection of the mutation for that iteration is determined.

Dividing the Monte-Carlo steps of the algorithm into two stages transforms the the system, as depicted in the lower half of Figure 5.17. The mutation and evaluation of the mutation for the previous iteration of the simulation, account for the *CPU Computation* blocks in the figure. Updating and writing back the result of the interaction potential calculation are host to device (HtD) and device to host (DtH) operations and are also asynchronous (executed concurrently with GPU kernel execution of another stream). *GPU computation* refers to work done by any GPU kernel. Depending on the size of the simulation, this may also include a parallel reduction of the result on the GPU.

A synchronisation barrier is inserted between the two parts of the Monte-Carlo step to ensure that the result from the asynchronous write back is complete. This is done on a stream by stream basis so that only the stream that the CPU is waiting for is synchronised, ensuring maximum overlap of CPU and GPU operations.

The introduction of streams also necessitates the use of storage for intermediate GPU results. Previously, an array in global memory was initialised just in time for a kernel's thread blocks to store their results. It was processed immediately and a single value returned to the host. Now, because the calculation of the result is asynchronous and the GPU reduction of the result or the transfer of the result is also asynchronous, this memory must be passed from the first to second stages of the Monte-Carlo steps to avoid introducing either errors or a synchronisation barrier. Paired patches of GPU global memory and page-locked CPU memory are reserved at the start of a simulation and reused for storing partial results for a particular stream throughout the lifespan of the simulation.

The implementation of streams in the manner described is included alongside the blocking GPU code, allowing profiling and benchmarking of each method of implementation by setting an appropriate preprocessor variable. The threading model also allows for full encapsulation of the asynchronous code as all other aspects of the simulation, remains the same.

5.3.7 Multiple GPUs

The implementation of the reusable threading model discussed in Section 5.2 and depicted in Figure 5.6 suits a multiple GPU solution. The scope that each thread provides matches that of the CUDA runtime. Because each thread has its own runtime, using a different GPU is merely a matter of selecting the required device for a thread to use by executing the CUDA call *cudaSetDevice*, because all subsequent calls in a thread are context sensitive, bound to that thread's CUDA runtime. GPUs are assigned to a thread by the thread identifier modulo the number of devices. In this way multiple threads can either have one GPU each, or share GPUs. However, the CUDA Programming Guide suggests that there be a mapping of 1 GPU to 1 thread. Testing of our implementation agrees with this guideline as performance notably decreases if more than one thread uses a single GPU.

The manner in which multi-threading is implemented means that there is no difference between a single and multiple GPU implementations. Simulation parameters merely specify the number of GPUs to use at runtime.

5.3.8 Compensated Summation

The numerical accuracy of the interaction potential is sensitive to round off errors when adding together individual pairwise potentials. The efficiency of 32-bit precision of on the GPU means that it not be beneficial to use double precision operations to accumulate potentials within each thread as it would be 8 times slower than single precision. Unfortunately, the 23-bit significand of a 32-bit number can only store 7_{10} significant figures, too few to avoid the accumulation of errors when there are potentially millions of pairwise potentials.

We compensate for the round-off error in two ways. Implicitly, the reduction performed in each kernel results in pairwise or cascade summation within each thread block which imposes an upper bound of $O(log_2 n)$ on the accumulated error in the summation [174], here n is the number of threads in the block. Practically, RMS error grows as $O(\sqrt{\log_2 n}$ [175] making a parallel summation much less sensitive to round-off errors than sequential summation which accumulates errors proportional to $\sqrt{n}$ (n in the worst case) [174].

We implement Kahan summation within each thread to minimise round-off errors in the sequential sections for our kernel code, limiting the accumulated error in an individual thread in a thread block to a constant upper bound of ε, the precision of 32-bit floating point representation, and independent of the number of additions. We also implement Kahan summation for summing the results of each thread block on the CPU.

The above two methods mean that the RMS error in the addition of the pairwise potentials in our simulations has an upper bound of $O(\varepsilon\sqrt{\log_2(\text{blockDim})})$. Errors outside this bound will be caused by differences in calculation on either the GPU of CPU.

5.4 Sampling and Clustering

The implementation details of the previous two sections are the main focus of this work. However, additional features need to be implemented to allow for analysis of implementation performance and validity of the simulation results.

5.4.1 Sampling

Generating data from simulations is performed by sampling replicas at discrete intervals. The simulation accepts two parameters at run-time that govern sampling. The first value, *Sample After*, is the number of Monte-Carlo iterations, per replica, after which sampling begins and the second value, *Sample Interval*, is the sampling interval, which is also a number of Monte-Carlo iterations.

The purpose of sampling is to determine whether or not a replica is in a bound or unbound state. A molecule is considered bound to another molecule if the interaction potential between them is less than or equal to 2 K_bB or 1.184 kcal/mol for this model [12]. A sampling function is inserted into the Monte-Carlo simulation code and is called only when the number of iterations is a multiple of the sampling rate and the requisite number of initial iterations has passed.

Fraction Bound and Acceptance Ratios

The fraction bound of a replica is the ratio of the number of bound samples to the total number of samples. It is a scalar quantity between 0 and 1 and is indicative of the binding affinity between particular molecules at a particular temperature [12]. The closer the fraction bound is to 1, the higher the binding affinity of the proteins. The fraction bound is used to determine the dissociation constant, K_d, which is used as a metric for binding affinity.

The simulations themselves do not determine K_d, instead, the fraction bound, is output for each temperature in the simulation as a pair consisting of an instantaneous fraction bound and an accumulated fraction bound. The cumulative value corresponds to the overall dissociation constant.

The acceptance ratio of the Monte-Carlo simulation is the ratio of accepted mutations via the Boltzmann distribution to all mutations. Acceptance ratios are again recorded for the current sample interval and over the entire duration of the simulation for each replica.

Fraction bound and acceptance ratios from simulations are used to validate our implementation against the results attained by Kim and Hummer [12]. Further explanation of fraction bound and acceptance ratio follow in section 6.2.

Bound Structures

Once a set of molecules achieves a bound state, the sampling process records the positions and rotations of the molecules relative to their input PDB data. This allows for the PDB data to be transformed using these relative values in order to be able reconstruct the bound complex for an all atom representation. Each bound state is output to file as a record of plain text strings, the first of which lists the Monte-Carlo iteration, interaction energy and the temperature pertaining to the specific structure in question. This is followed by a list of strings, one for each molecule in the bound structure, containing the absolute position of the centroid of each molecule and the rotation quaternion transforming the PDB input from its original orientation to the orientation of the bound structure.

A trivial tool to transform this output file into a PDB format files has have been implemented.

5.4.2 Clustering

A Monte-Carlo simulation of this type outputs many instances of bound molecules. To discover the most likely way that molecules will bind requires the use of clustering. Clustering allows for better overall analysis of the output from simulations because the simulations are likely to find many instances of the bound complexes that are nearly identical, separated by perhaps a small shift in one direction or rotational difference for the same centroid position.

Experimentally, clustering allows simulations to discover the most likely binding sites between molecules in a complex. For validation, it allows simulation outputs to be directly compared to experimentally observed structures such as those found in the Protein Database.

Kim and Hummer cluster structures according to the distance between each pair of residues from each protein molecule in the complex. A distance matrix is constructed from the distances between pairs of residues from different proteins. Structures are assigned to a cluster based on this distance matrix using a "simple clustering algorithm" [12].

We perform clustering using GROMACS' g_cluster utility, producing representative structures of docked complexes. A distance root mean square (DRMS) comparison between the simulated structure and native structures is performed according the description of DRMS by Kim and Hummer [12], providing a quantitative metric to compare the structural similarity between the simulated and native structures.

5.5 Implementation Summary

In this chapter an iterative approach to implementing various CPU and GPU functionality was presented.

The implementation of a CPU solution, encompassing only the core functionality of the model, provided a code base from which multi-threading and GPU acceleration could be developed. The initial GPU acceleration showed that the model could be implemented on a GPU and benefit from the performance advantages on offer.

Implementation of a multi-threaded CPU version of the simulations resulted in encapsulating Monte-Carlo simulations at thread level, providing a theoretically scalable layout of the algorithm for combinations of multiple cores, multiple GPUs or multiple compute nodes.

In further development we delved into the instruction and algorithmic implications of using a GPU. Managing the memory bandwidth dependency and division of work between threads further improved the GPU solution such that it performed faster and could scale favourably to a far greater problem size than the initial GPU implementation. Applying specific techniques for bandwidth management coupled with methods for maximising the utilisation of both CPU and GPU together ensures that the GPU code is optimised to an acceptable degree without the need to program at the assembly level (CUDA PTX).

Detailed discussion of the methods used to implement the GPU and CPU implementation also highlighted areas of interest when benchmarking the system and tuning the parameters for performing simulations such as those discussed in Chapter 9 pertaining to macromolecular crowding.

Finally, implementation of a minimal set of applications to transform and interpret output from our simulations allows for validation of the results produced by the application, as well as interpretation of any novel simulations to be run using our implementation.

Chapter 6

Verification and Validation

A model is built by abstracting a real world system. This model, in the form of a computer program will run and produce results in the form of simulated data. These results can, and must, be compared to the real world system from which it came [176]. Validity can be divided into three types, replicative validity, predictive validity and structural validity [176].

For a model, or simulation, to be replicatively valid, it must be able to reproduce data acquired from a the system it models. Extending this concept, predictively valid models are able to produce valid data before this data is observed in the real world. Finally, structural validity will produce data reflective of real world data and the manner in which the system operates in order to reproduce this behaviour [176].

In our implementation we assume that the model [12] possesses both replicative and structural validity. Kim et al. base their model upon general understanding of intermolecular forces by using Lennard-Jones and Coulomb potentials. With these forces, the Boltzmann distribution and the laws of thermodynamics are followed. Kim et al. found that the binding affinity estimates that their model produced from the set of simulated structures agreed quantitatively with experimentally derived values. In the case of UIM1/Ub, it effectively located both of the binding interfaces in 90% of the simulations, at least one of the interfaces correctly in the other 10%. The structures also agree with experimental structures to within 2Å to 5Å. When clustered the dominant cluster contains both native structures and has a population exceeding 40% [12].

We can therefore use similar metrics to compare our simulations to what can be considered replicatively and structurally valid references. In our case, predictive validity is impossible to guarantee and the experimentally validity of the results of a blind docking simulation is beyond the scope of this work. Given that the model is valid, our validation take is so show that we can reproduce the results of Kim and Hummer [12], explicitly showing replicative validity and by so doing assume structural validity.

Separate to validation of the simulation and model, is the verification of the implementation.

The non-trivial nature of GPU programming and the numerical inaccuracy encountered on a GPU necessitates verification of the simulation from a numerical perspective. We inspect our calculation of interaction potentials on the CPU, comparing our values to those attained using the original implementation. The same calculations are then repeated using our GPU implementation, assessing how close they are to the CPU implementation and reference values.

In conjunction with correct interaction potentials, we ensure that the mutation operations preserve the integrity of molecular structures for the duration of a simulation. Preserving these structures is critical because the structure of a molecule dictates its biological function which, in turn, affects its behaviour when docking. Thus, changes in structure due to numerical error will affect the validity of the simulated structures and statistics.

Verifying the integrity of both the interaction potential calculations and the molecular structures, allows us to perform validation of the Monte-Carlo simulations and replica exchange. Monte-Carlo simulations are dependent on acceptance/rejection sampling using the Boltzmann factor, thus, we need to show that the simulations accept only legal mutations and not mutations that cause collisions between molecules. The correctness of our implementation of replica exchange is evaluated using the fraction bound ratio as an indication that the replicas at a range of temperatures fit the Boltzmann distribution. The temperature of each replica determines the likeliness of it possessing a particular energy configuration, hence, the probability that it will be bound is related to its temperature by the Boltzmann distribution.

Finally, to validate that simulations function correctly, we perform two reference simulations. The first of these models the binding of ubiquitin and the UIM1 domain of Vps27 (abbreviated to UIM/Ub). The second simulation models the binding of the yeast cytochrome *c* and cytochrome *c* peroxidase (Cc/CcP) complex. The Protein Database (PDB) provides entries 1Q0W and 2PCC, containing the data for each respective complex. Kim and Hummer use the same simulations to validate their model [12], providing data against which we can perform replicative validation for our implementation. Output from these simulations is used to calculate the binding affinity of each complex at 300 Kelvin, allowing direct comparison of our simulation outputs with their results.

All of the work relies on the accuracy of the structures stored in the Protein Data Bank. In the case of UIM/Ub, protein nuclear magnetic resonance spectroscopy was used to generate the structural data in a study performed by Swanson et al. [177], published in The EMBO Journal, part of Nature. The Cc/CcP structure was attained using X-Ray diffraction in a study by Pelletier and Kraut published in Science [28]. Altogether, these works have been cited 102 and 328 times receptively as of January 2010.

6.1 Verification

Verification is performed on the interaction potential calculations. We casualty and compare our CPU and GPU computations with a reference set of interaction potential values calculated using the original implementation in CHARMM. Thereafter, verification of the Monte-Carlo mutations are performed. The relative simplicity of Monte-Carlo simulations limits verification to these aspects of the simulation; tasks such as sampling, recording of docked poses and the running of the simulations are trivial.

The verification of the interaction potential summation implicitly verifies the structural validity of the internal representation of proteins. Any structural corruption at the residue level will result in distinct differences in interaction potential. Thus, second to the calculation of accurate interaction potentials, minimising the accumulation of numerical error caused by Monte-Carlo mutations is critical.

6.1.1 Interaction Potentials

Kim and Hummer. use CHARMM to implement their model and perform their simulations [12]. The new implementation using C++ and CUDA needs to be validated against it to ensure that the calculation of the interaction potential concurs with the original implementation before any new simulations can be performed.

Ten reference conformations of the UIM1/Ub complex, subject of Kim and Hummer [12], are used for validating the correctness of the interaction potential calculations. Table 6.1 lists the CHARMM generated values of interaction potential for each conformation. These conformations are reproduced using our implementation and the interaction potentials we calculate for each of them are listed in Table 6.2. In the tables, U_{vdW} and U_{DH} are the respective accumulated values for the short range, u_{ij}, and electrostatic, u_{ij}^{el}, components from Equation 4.1. The reference values were provided by Dr R. Best using ten randomly selected configurations of UIM/Ub. PDB files containing the C_α positions of each amino acid residue in these conformations are output before being loaded into our implementation and the interaction potentials for each configuration are calculated using the CPU.

Tables 6.1 and 6.2 show that our implementation matches that of Kim and Hummer. to, at least, 3 significant figures (conformations 5 and 7). Further inspection of the potentials showed that CHARMM and our implementation output different values in some cases for the u_{ij} component. Specifically, when our implementation outputs zero, CHARMM would output a non-zero value. The interaction between GLU and ARG is a good example, using $e_0 = -2.27$ [12], and the contact energy from Table A.2, the value of ε_{ij} (Equation 4.2) is zero. Consequently, the correct contribution from the short range interaction potential, u_{ij}, is zero. CHARMM, in the case specific to this test, calculates $u_{ij} = 4.6 \times 10^{-8}$. As a result, our calculation of the interaction

Table 6.1: Reference Conformation Energies
Reference conformation energies produced using the CHARMM. U_{Tot} is the total potential of the conformation, comprises U_{vdW}, the total Lennard-Jones pair interaction potential component, and U_{DH}, the total Debye-Hückel electrostatic potential component.

Conf.	U_{Tot} (kcal/mol)	U_{vdW} (kcal/mol)	U_{DH} (kcal/mol)
1	-0.294	-0.081	-0.213
2	-1.056	-1.323	0.266
3	-10.278	-9.095	-1.184
4	-7.584	-5.905	-1.680
5	-7.91 $\times 10^{-5}$	-2.12 $\times 10^{-5}$	-5.8 $\times 10^{-5}$
6	-5.565	-4.812	-0.753
7	-5.453	-4.184	-1.269
8	-10.670	-9.223	-1.447
9	-9.904	-7.952	-1.952
10	-8.518	-7.448	-1.070

potential and CHARMM's calculation of the interaction potential differ. With regard to the 3 significant figures, one reason for our values and CHARMM's values differing might be that the value of e_0 is truncated, meaning that the value of -2.27, may need to be more accurate for our implementation results for each potential interaction caclualtion to match those of Kim and Hummer. But, it can be argued that these differences do not affect the outcomes of the simulations due to the influence of the Boltzmann factor.

Table 6.2: Implementation Conformation Energies
The CPU implementation of the algorithm determining conformation energy closely matches the reference values in Table 6.1.

Conf.	U_{Tot} (kcal/mol)	U_{vdW} (kcal/mol)	U_{DH} (kcal/mol)
1	-0.294	-0.081	-0.213
2	-1.056	-1.322	0.266
3	-10.277	-9.095	-1.182
4	-7.580	-5.903	-1.678
5	-7.90 $\times 10^{-5}$	-2.10 $\times 10^{-5}$	-5.8 $\times 10^{-5}$
6	-5.562	-4.810	-0.752
7	-5.480	-4.213	-1.267
8	-10.712	-9.266	-1.446
9	-9.900	-7.951	-1.949
10	-8.528	-7.459	-1.069

All contact potentials are truncated to 3 significant figures, therefore, we cannot claim to calculate interaction potential to more than 3 significant figures using this data.

Additionally, the input PDB files truncate the absolute positions of each atom to 3 decimal places, and so the value of r, the distance between two residues, is calculated using between 3 and 6 significant figures depending on a residue's distance from the origin point in a PDB file. This means that r is only accurate to the same number of significant figures, meaning that the interaction potential can only be accurate to the same degree of precision because the calculation of the Debye-Hückel electrostatic component of the interaction potential, u_{ij}^{el}, and the van der Waals component of the potential, u_{ij}, are both dependent on r.

Table 6.3: Relative Difference Between Calculated and Reference Values
The CPU implementation of the algorithm matches the reference values to a reasonable degree of accuracy. The largest discrepancy between the implementation and the reference values is in the calculation of the short-range van der Waals potential, U_{vdW}.

	Mean Relative Error $\bar{\eta}$	Maximum Relative Error η_{max}
U_{Tot}	0.00143	0.00507
U_{vdW}	0.00238	0.00756
U_{DH}	0.00114	0.00142

The relative error, $\eta = \frac{|x - x_{sim}|}{|x|}$, between the reference values in Table 6.1 and our values in Table 6.2 shows our implementation to have a mean relative error of 0.0014 compared to CHARMM's values. Table 6.3 shows that the largest differences occur in the calculation of the van der Waals component of the interaction potential. Considering the above discussion regarding significant figures, achieving a maximum relative error of less than 1% (0.0014), shows the implementation to be accurate to an acceptable degree of precision for this simulation.

GPU Verification

The only function of the GPU in our simulations is to calculate the total interaction potential. We verify that our GPU calculations are correct in two ways. First, we calculate the interaction potentials using the GPU for the ten conformations used to validate the CPU and compare them to those in Tables 6.1 and 6.2. Second, we run a 1000 iteration Monte-Carlo simulation of both the UIM/Ub and Cc/CcP complexes, and compare the calculation of interaction potential on both CPU and GPU for each mutation.

To begin, calculation of the total interaction potential on the GPU is performed for each of the ten reference conformations, the results of which are included in Table 6.4. Analysis of the values shows that the mean relative error between our CPU and GPU implementations is 3.8×10^{-7} and the mean relative error between our GPU implementation and CHARMM is 0.00146, almost identical to the mean relative error between our CPU implementation and CHARMM (0.00143).

Table 6.4: GPU Conformation Energies The U_{Tot} values for the 10 conformations from Table 6.2 are compared to the GPU calculated U_{Tot} or the same conformations. Relative errors are shown to be between 2×10^{-6} and 5×10^{-11} for the reference sets.

Conf.	CPU (kcal/mol)	GPU (kcal/mol)	η
1	-0.293705	-0.293705	5.21×10^{-8}
2	-1.056291	-1.056291	3.63×10^{-7}
3	-10.277435	-10.277431	2.78×10^{-7}
4	-7.58038	-7.58039	1.00×10^{-6}
5	-7.924×10^{-5}	-7.924×10^{-5}	5.03×10^{-11}
6	-5.562238	-5.562239	2.01×10^{-6}
7	-5.480216	-5.480217	3.74×10^{-7}
8	-10.711964	-10.711967	1.92×10^{-6}
9	-9.900360	-9.900360	5.68×10^{-7}
10	-8.527744	-8.527749	1.75×10^{-6}

All values have been calculated using single precision on the GPU. While double precision is available on a GPU, there is one double precision unit for every eight single precision units on the GT200. Consequently, performing calculations using double precision is more accurate, but at a speed eight times slower than single precision.

Compensated Summation

To reduce the effect of truncation error on our simulations, we use a combination of Kahan summation and pairwise summation in the manner discussed in the previous chapter (cf. 5.3.8). The effects of using compensated summation on our simulation are small, resulting in only a minor improvement in the relative error between the GPU and CPU calculated values. Of greater significance is that the standard deviation in the error is always smaller when using compensated summation. More importantly, compensated summation ensures that the theoretical worst case error will be bounded by $O(\varepsilon\sqrt{\log_2(\text{blockDim})})$ independent of the size of the simulation configuration. Thus, we can infer that the errors in the simulations we perform for validation will be similar to errors incurred in simulating larger structures.

To investigate the effects of compensated summation, perform simulations calculating the interaction potential for each step on both the CPU and GPU, recording the relative error at each step. In this test, the CPU uses a double precision compensated sum (effectively 104-bit precision) and the GPU uses a compensated sum of floats within each thread, this provides 46-bit precision for 4096 pairwise interactions (the block size of 64 results in 64×64 comparisons) before being truncated by reduction to the 23-bits most significant bits. A Kahan sum is once again performed on the grid results. We simulate problems ranging in size from 100 residues to 7008 residues, producing 30000 samples.

This comparison calculates that we may expect 99.7% (3σ) of all values to agree with a relative error of less than 1.6×10^{-5} without compensation and 1.0×10^{-5} when compensating. Thus, we may discard the use of compensated summation for our simulations as its computational cost does not realise a significant increase in accuracy.

Table 6.5: The Effect of Compensated Summation Summary Compensated summation marginally improves results at the cost of approximately 10% more computation. Given This result and the fact that we are interested in binding values in the range of -1.18 or less. The greater degree of accuracy afforded by compensated summation does not necessitate or require its use for our simulations. For 30000 samples we note a decrease in the standard deviation, σ, indicating that compensated summation results in the interaction potentials values evaluated by the GPU and CPU to agree more closely. But, as the mean relative error figures suggest, there is little improvement on the already acceptable error in the calculation.

	Sequential	Compensated
Mean η	1.06×10^{-6}	9.56×10^{-7}
σ	5.53×10^{-6}	3.34×10^{-6}
Minimum η	0	0
Maximum η	$6.89E - 004$	$3.49E - 004$

Analysing the components of the interaction potential reveals that the predominant reason for differences in GPU and CPU calculated values lies in the calculation of the van der Waals component of the interaction potential. This function relies on *powf*, and is sensitive to the relative inaccuracy of such a function. The functions used to implement our GPU solution, generally, have a maximum ULP error of one, two or three places, however, the *powf* function, has a maximum ULP error of 8 places [22]. This means that the mantissa of the single precision value returned by this function can only be guaranteed to be accurate to 15 of its 23 bits. This translates to accuracy of only 5 significant figures in the worst case. Fortunately, the Monte-Carlo simulations will compensate for this inaccuracy because the Boltzmann factor statistically accepts or rejects mutations using the change in interaction potential, meaning that small errors in accuracy will not cause significant changes in the the probability of acceptance of rejecting a mutation.

A critical point to consider in the calculation of the relative error is that neither the CPU nor GPU can be considered absolutely accurate. Tables 6.2 and 6.4 show that the CPU and GPU agree with values calculated using CHARMM but there is no guarantee that either CHARMM, the CPU or the GPU calculate the true interaction potential. Because the GPU and CPU calculate interaction potentials which match with relative errors many orders of magnitude smaller than the value of interest in these simulations, the CPU and GPU affirm that the accuracy of our implementation, using single precision interaction potential calculations, is sufficient.

6.1.2 Monte-Carlo Mutations

The calculation of the interaction potential is preceded by a Monte-Carlo mutation and followed by either accepting or rejecting the mutation based on the value of the interaction potential. Specifically, it is the Boltzmann Factor that is used to perform acceptance/rejection sampling. Validation of the Monte-Carlo simulations is performed in two parts, firstly, the acceptance and rejection of mutations must be shown to work correctly, such that incorrect mutations are not accepted. Secondly, the mutations must preserve the structure of the molecules they mutate, ensuring that their structure does not become corrupted during simulation, because this would invalidate all data produced by that simulation.

To ensure that the Monte-Carlo moves conserve the structure of the molecules, the rotation and translation methods are tested. Unlike the calculations performed on the GPU and CPU to determine interaction potential, the rotation and translation operations modify the state of the model. Consequently, the effect of these operations needs to be as accurate as possible in order to perform valid simulations.

Testing translations is trivial since it merely involves the addition of the translation vector to each residue's absolute position. This operation was shown to be implemented correctly as calculation of the interaction potential does not change unless the position of molecules relative to each other changes. This was tested by applying a series of identical translation operations to all molecules before evaluating the interaction potential which, as expected, was identical before and after translation.

To verify that mutations rotate the molecules correctly, a test to evaluate rotations was performed. Unlike translations, rotations are sensitive to numerical instability due to truncation errors. Initially, only single precision rotations were used for our implementation, resulting in distortion of a molecules shape over time. Molecules tended to become elongated along a particular axis while simultaneously squashed along another. Further analysis of this problem showed that the source of the distortion was the precision used when rotating the molecule.

For data acquisition, simulations are expected perform between 10^8 and 10^9 Monte-Carlo iterations. This means that, given equal probability of a rotation or translation in a binary complex, approximately 2.5×10^7 rotations will be performed on a single molecule. Therefore, rotations must preserve the structure of the molecule to an acceptable degree for at least this many operations. Due to the Markov chain nature of the Monte-Carlo simulation, each time a rotation occurs, the effect is accumulated

To test the validity of our rotations, a vector is rotated 10^9 times using the following methods of rotation:

- Single (32-bit) precision matrix rotation.
- Double (64-bit) precision matrix rotation using a single precision rotational axis.
- Double precision matrix multiplication using a double precision rotational axis.
- Single precision quaternion rotation.
- Double precision quaternion rotation using a single precision rotational axis.
- Double precision quaternion rotation using a double precision rotational axis.

The two methods of rotation, quaternion and matrix, are implemented in the following ways. Rotations are relative to the origin since all rotation operations are performed on a residue's position relative to the center of the protein to which it belongs.

A rotational matrix, R, (Equation 6.1) is generated from the angle/axis pair $(\hat{u}, \theta)$ and rotates any vector anticlockwise θ radians about the arbitrary axis $\hat{u}$ [178].

$$R = \begin{bmatrix} u_x^2 + (1-u_x^2)\cos\theta & u_xu_y(1-\cos\theta) - u_z\sin\theta & u_xu_z(1-\cos\theta) + u_ys \\ u_xu_y(1-\cos\theta) + u_z\sin\theta & u_y^2 + (1-u_y^2)\cos\theta & u_yu_z(1-\cos\theta) - u_x\sin\theta \\ u_xu_z(1-\cos\theta) - u_y\sin\theta & u_yu_z(1-\cos\theta) + u_x\sin\theta & u_z^2 + (1-u_z^2)\cos\theta \end{bmatrix} \tag{6.1}$$

Rotation via quaternions is performed by generating a quaternion, $Q(w, x, y, z)$, representing the rotation using the same angle/axis pair, $(\hat{u}, \theta)$ [178].

$$Q = \left(\cos\frac{\theta}{2}, \hat{u}_x\sin\frac{\theta}{2}, \hat{u}_y\sin\frac{\theta}{2}, \hat{u}_z\sin\frac{\theta}{2}\right)$$

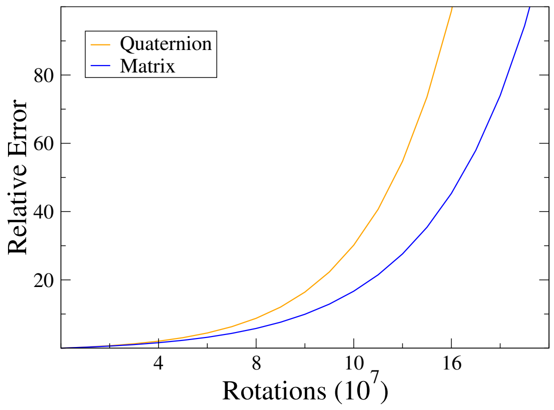
Using the Hamilton-Cayley formula (Eq. 6.2),

$$V' = QVQ^{-1} \tag{6.2}$$

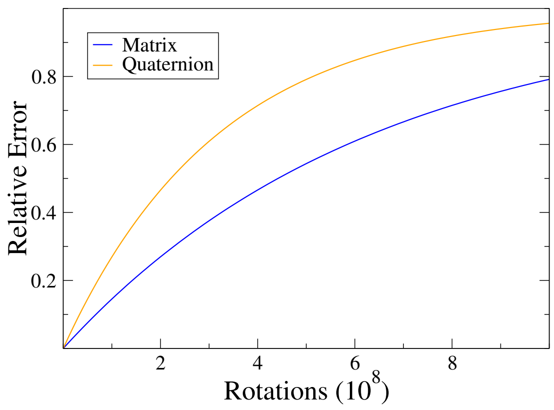
V' is the vector V rotated using the quaternion Q. Expanded this becomes the 3 × 3 matrix multiplying the vector V to produce V' [179].

$$V' = \begin{bmatrix} w^2+x^2-y^2-z^2 & -2wz+2xy & 2wy+2xz \\ 2wz+2xy & w^2-x^2+y^2-z^2 & -2wx+2yz \\ -2wy+2xz & 2wx+2yz & w^2-x^2-y^2+z^2 \end{bmatrix} V \tag{6.3}$$

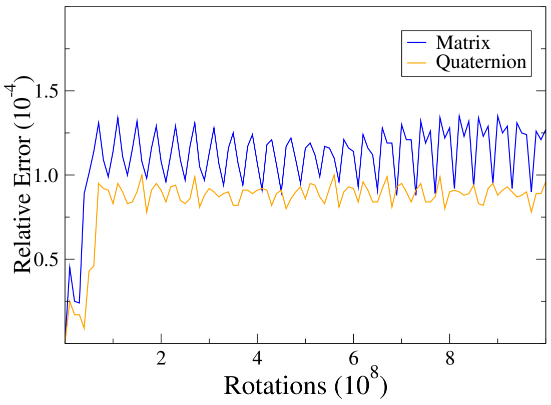
The test to inspect the effects of a rotation over many iterations involves choosing a vector of known length, and rotating it by 0.2 radians (the rotation from the model) about a random axis 10^9 times. If the rotation operation is acceptably accurate, the length of the vector will be preserved. The relative error $\frac{|x_0 - x_i|}{|x_0|}$ is calculated using the length of the original vector, x_0, and its current length after i iterations, x_i.



(a) 32-bit Rotations



(b) 64-bit Rotations, 32-bit Rotational Axes



(c) 64-bit Rotations, 64-bit Rotational Axes

Figure 6.1: Rotational Error

(a) A large number of rotations using single, 32-bit, precision performed on a single vector induces severe numerical instability. After 10^7 rotations, using a rotation matrix or quaternion, rotations suffered absolute errors of 29% and 26% respectively, with the error growing at an exponential rate. (b) Using a mixture of 32-bit inputs and 64-bit operations improves accuracy considerably, but still not to an acceptable degree. (c) Only by using double precision for rotations, does the accumulated error in the operations become tolerable, tending, after many iterations, to a numerically stable state.

Using our initial implementation, rotation about an arbitrary axis proved accurate for very few iterations of the test. After less than 10^7 rotations the length of the test vector was distorted by at least 29% using quaternion rotations and 26% using a rotation matrix. As illustrated in Figure 6.1a, the error compounds quickly using only single precision. The errors introduced after 10^9 iterations distorted vectors by a factor of 4×10^{12} using rotational matrices and 3×10^{10} using quaternions.

Changing all operations that perform the rotation to double precision improved the accuracy considerably, containing the numerical instability to a better degree than single precision. However, as illustrated in Figure 6.1b, this still results in distortion of the vector by up to the same order of magnitude as the vector itself.

Only by generating the random axis of rotation, $\hat{u}$, using double precision, does the rotation operation become stable, experiencing accumulated errors of no more than 1.35×10^{-4} times the original vector length. Inspection of the rotation operations used in our implementation shows that double precision must be used to preserve the structure of the molecules involved in simulations. Furthermore, double precision becomes numerically stable with an acceptable error bound after approximately 10^8 operations. Figure 6.1c shows that error incurred by using quaternions or rotation matrices is similar, with quaternion operations, in our case being slightly more accurate.

The GPU is not used for rotation operations, thus, use of full 64-bit precision for all parts of the rotation algorithm are viable.

The Monte-Carlo mutations can thus be considered to conserve the structure of molecules in the simulation, which, in turn, ensures that the interaction potential will be calculated as accurately as possible. Interaction potentials and Monte-Carlo mutation are the basic building blocks for this simulation, which, now verified, can be used for the higher level simulation operations such as acceptance/rejection sampling via the Boltzmann factor.

6.2 Simulation Validation

We perform two reference simulations of the docking of the UIM/Ub and Cc/CcP complexes. These simulations are evaluated using the fraction bound metric, to determine the binding strength of the complex, and clustered to determine the structures discovered by the simulation. These outputs, in turn, allow for replicative validation against known binding strengths and structures from both experimental observations by Kim and Hummer.

The fraction bound,

$$y = \frac{[A]}{[A] + K_d}$$

is a function of the concentration, $[A]$, of the molecules in a simulation, allowing us to determine the dissociation constant, K_d, for the interaction modelled by the simulation. This value can then be compared directly with Kim and Hummer's results to ascertain the validity of the simulation.

Our simulations are performed using a periodic bounding box where the size of the bounding box determines the concentration of the simulation. Six box sizes are used in our test simulations ranging from 100 micro-molar (μM) to 1000μM.

Simulations for UIM/Ub ran for 15 million Monte-Carlo steps per replica for 12 replicas. Replica exchange was performed once for every 1000 Monte-Carlo steps, exchanging replicas between temperatures ranging from 300K to 600K. In each simulation, fraction bound was calculated from data gathered for replicas at 300K. The dissociation constant, K_d, is solved for by fitting the fraction bound values of each simulation to the curve, $y = \frac{x}{x+K_d}$.

We produce two values for K_d due to the use of truncated Ubiquitin by Kim and Hummer [180]. The difference between the truncated and non-truncated ubiquitin is the absence of the last four residues in the protein resulting in different structures as illustrated in Figure 6.2 with truncated ubiquitin on the left and full length ubiquitin on the right. The blue residues on the right are the residues removed through truncation.

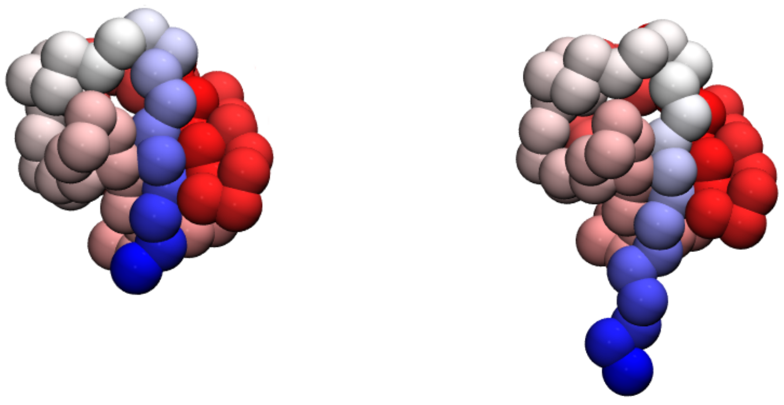

(a) Truncated (b) Full Length

Figure 6.2: Truncated and Full Length Ubiquitin
(a) Simulations performed by Kim and Hummer use a truncated form of ubiquitin with only 72 residues. (b) Full length ubiquitin, is 4 residues longer than the truncated form. The difference between the proteins is illustrated in blue. We perform the UIM/Ub docking simulation with both proteins to validate our simulations against both Kim and Hummer, using the truncated from, and Best using the full length protein as found in PDB entry 1Q0W.

Kim and Hummer use Model "A" (the surface accessible surface area for each residue equals 1) to calculate $K_d = 1240\mu M$ for UIM/Ub using truncated ubiquitin [12]. Our simulations produce a similar result, with $K_d = 1345\mu M$. For full length ubiquitin simulations, our value of K_d is compared with that of Best [180]. Best calculates $K_d = 595\mu M$ using CHARMM to perform simulations. Our simulations agree with this value, determining $K_d = 561\mu M$. Kim and Hummer do not publish a K_d value for full length ubiquitin. For truncated simulations, Best calculates $K_d = 1493\mu M$, a value which our simulations agree closely. Figure 6.3 plots fraction bound as a function of concentration for all six of these simulations, illustrating that our simulations produce values comparable to both Kim and Hummer [12] and Best [180].

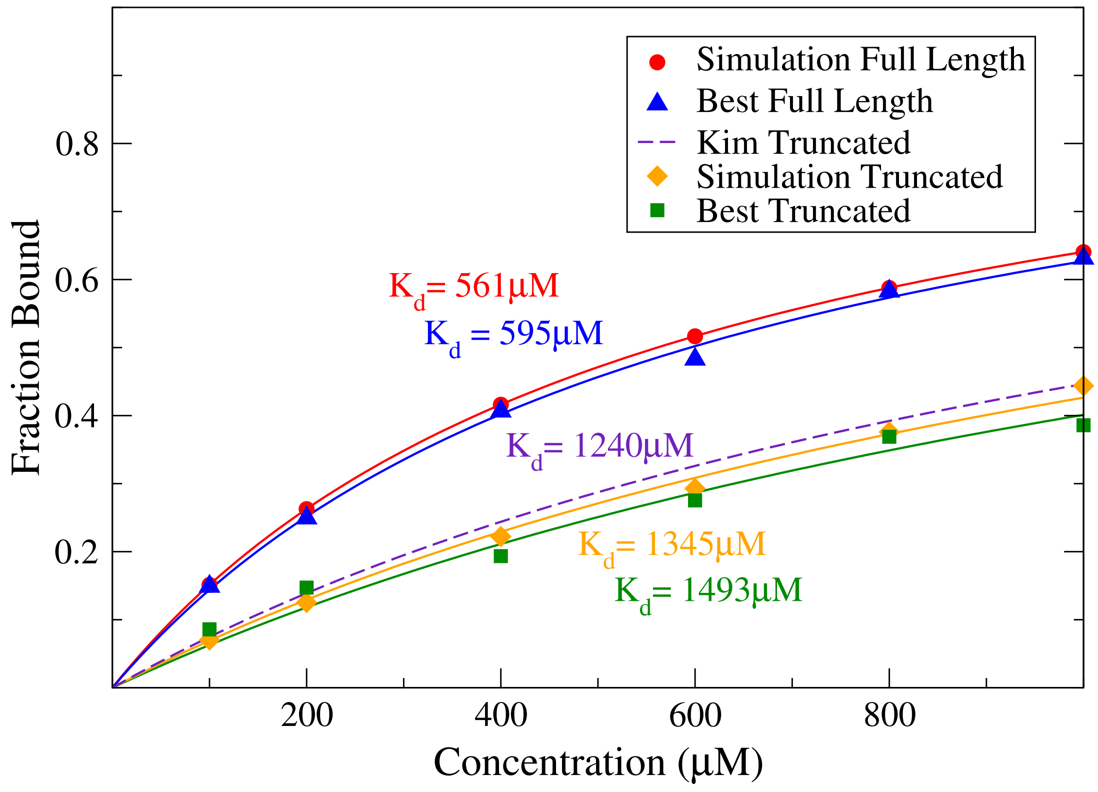


Figure 6.3: UIM/Ub Fraction Bound As A Function of Concentration
Simulations using both truncated and full length ubiquitin agree strongly with simulations performed by Best. Our simulations determine K_d to be $1345\mu M$ and $561\mu M$, respectively. These values closely match those of Best. in both cases and with Kim and Hummer for truncated ubiquitin.

Simulation pertaining to Cc/CcP are configured in the same manner as the afore-mentioned UIM/Ub simulations. Kim and Hummer determine the dissociation constant to be $1570\mu M$ [12]. Our simulations produce $K_d = 1456\mu M$, closely matching this value as illustrated by Figure 6.4.

Alarmingly, by changing the fixed translation of 0.5Å to a range of translations between 0.1Å and 0.5Å depending on temperature, simulations discover a higher binding affinity for the Cc/CcP complex with a dissociation constant of $755\mu M$. This value concurs much more strongly with the "Model C" SASA values used by Kim and Hummer as opposed to the simple "Model A" where the SASA of each residue is 1 [12]. "Model C" weights a residues interaction by the factor $\tanh(5\tan(\pi s/2))$. By comparison, UIM/Ub simulations performed with both variable and fixed translation sizes produce negligible differences to the dissociation constant for the reaction.

On the basis of our implementation agreeing strongly with the simulations by Best, in the

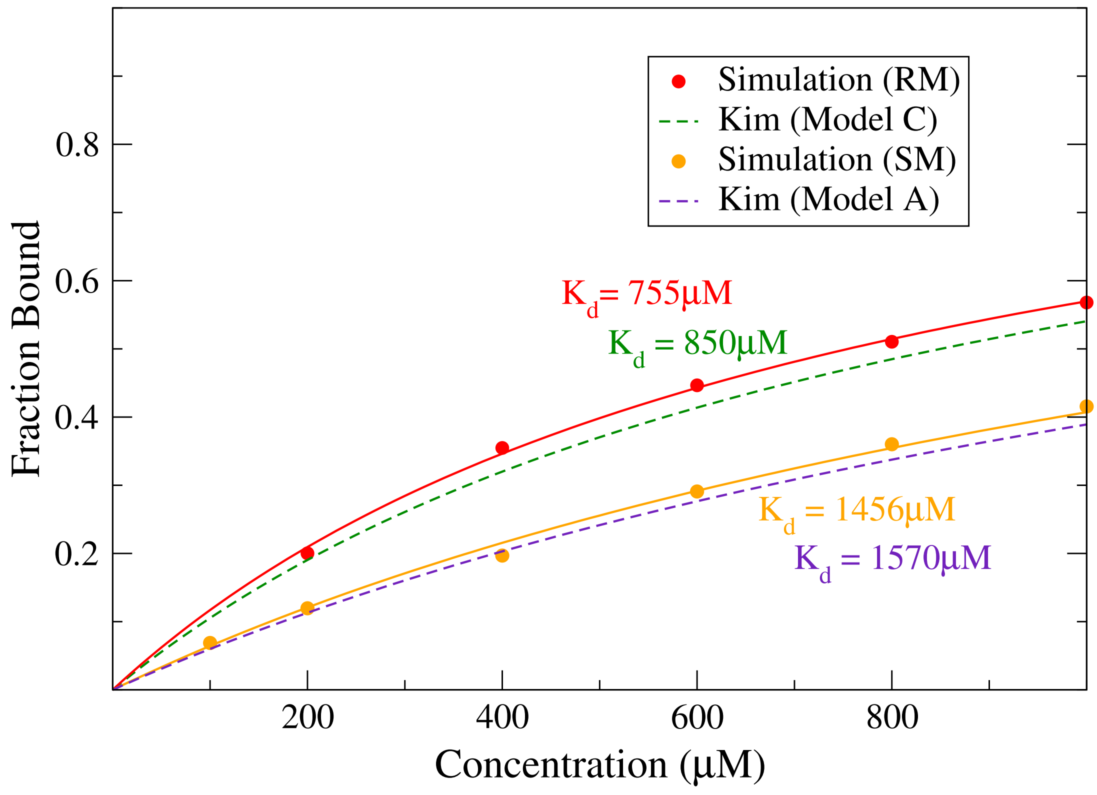


Figure 6.4: Cc/CcP Fraction Bound As A Function of Concentration
Our simulations calculate the dissociation constant of the Cc/CcP complex to be $1456\mu M$ *when using a constant translation and rotation step size for Monte-Carlo moves. This value agrees strongly with the value for* K_d *that Kim and Hummer calculate, namely* $1570\mu M$ *for "Model A". Introducing a translation and rotation proportional to temperature results in a binding curve more closely resembling that of "Model C".*

case of UIM/Ub and with Kim and Hummer in the case of Cc/CcP, we consider our implementation valid for determining the binding affinity of a complex. The ability of our implementation to calculate correct dissociation constants implies that it reaches equilibrium during the simulation and that this equilibrium is valid.

The DRMS of each structure is calculated to evaluate the structural similarity of the complexes to the native complex. Kim and Hummer find that the most populous structure is within 5Å of the respective X-ray crystal and NMR solution structures of that complex. DRMS is calculated as the sum of difference in the distance matrices for all pairs of residues in the simulated and experimental structures over the total number of pairs:

$$\frac{1}{N}\sum_{i,j}|d_{ij}^{sim} - d_{ij}^{exp}|.$$

DRMS is a better measure for structure comparison than the RMSD for large protein complexes because it is insensitive to the small relative orientational differences within the complexes that would result in large RMSD changes [12].

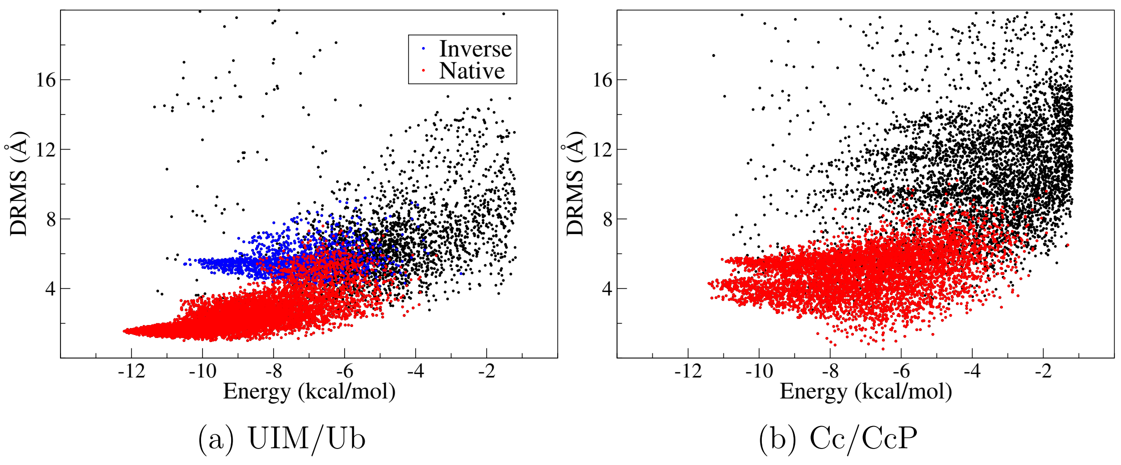


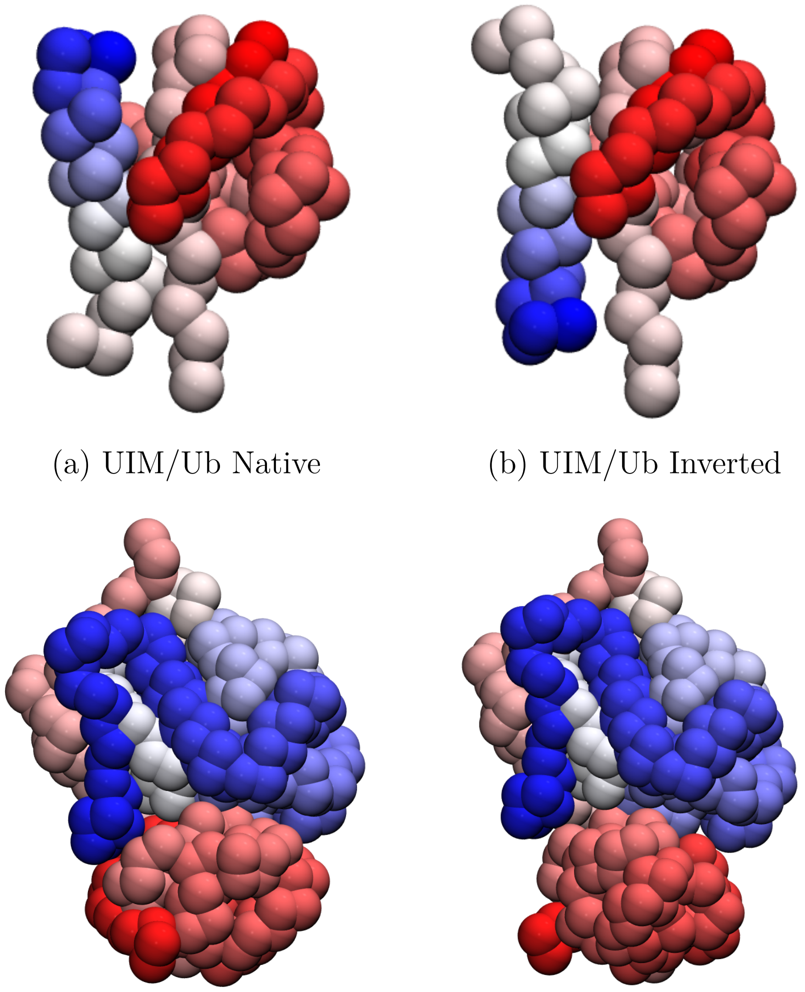
Figure 6.5: Bound Structure DRMS versus Energy
UIM/Ub (a) and Cc/CcP (b) simulations discover clusters of native like structures (red), exhibiting both low energy and DRMS values. In UIM/Ub simulations, the inverted UIM helix results in a secondary structure (blue) with increased RMSD and energy. Complexes are considered bound when the interaction potential between then is less than -1.184kcal/mol.

Plots of the DRMS versus potential energy for both UIM/Ub and Cc/CcP complexes (Figure 6.5) show that the low-energy structures exhibit small DRMS values. Kim and Hummer attribute the higher DRMS values associated with Cc/CcP to the orientational changes found in larger proteins (402 for Cc/CcP vs. 100 for UIM/Ub) will result in an increase in DRMS. Thus, for clustering, a larger cut-off value is required for Cc/CcP than UIM/Ub. Qualitatively, our DRMS versus energy plots strongly resemble the results attained by Kim and Hummer.

Finally, structural analysis of the conformations discovered is performed. Frames of the bound complexes discovered by our simulations are output as a PDB trajectory for clustering with GROMACS' *g_cluster* utility using a 1Å cut-off value for UIM/Ub and 2Å for Cc/CcP.

UIM/Ub simulations, with both truncated and full ubiquitin, discover two dominant clusters. These clusters account for 79% and 72% of bound complexes discovered in the full and truncated simulations respectively. In the full length simulations, a cluster representing the native structure (Figure 6.6a) accounts for 67% of all complexes discovered. The native cluster has a DRMS of 2.4Å ±1.0Å with the closest structure 1Å DRMS from the experimentally attained structure. The second cluster (Figure 6.6b) has DRMS of 5.5Å ±0.7Å with the minimum DRMS of 3.7Å. This second cluster inverts the orientation of the UIM helix in the binding pocket and is also present in Kim and Hummer's simulations [12].

Truncated ubiquitin produces similar results with respect to structure. Again a native like and secondary structure emerge with DRMS measures of 2.4Å ±0.8Å and 5.0Å ±0.6Å, respectively. Like Kim and Hummer, we find that the second cluster has a population of 20% and



(a) UIM/Ub Native (b) UIM/Ub Inverted

(c) Cc/CcP (d) Cc/CcP Native

Figure 6.6: UIM/Ub and Cc/CcP Clusters
Simulations discover a native docking pose (a) and a docking pose in which the UIM1 helix is inverted (b). Cc/CcP simulations produce a single dominant structure (c) containing a near native structure 0.53Å DRMS from the experimental structure (d).

a DRMS of 5Å [12]. In this case the minimum DRMS structure is the same for the inverted UIM cluster at 3.7Å, the presence of which has been observed experimentally [12]. In the native cluster, the minimum DRMS discovered is 0.78Å and approximately 1% of conformations have a DRMS of less than 1Å.

Clustering cytochrome *c*/cytochrome *c* peroxidase (Cc/CcP) simulations produce a single dominant cluster accounting for 56% of the bound complexes discovered. The cluster (Figure 6.6d) has a DRMS of 4.9Å ±1.3Å and a minimum DRMS of only 0.5Å. If clustered using a 1Å cut-off, it emerges that this cluster can be categorised as two equally sized sub-clusters. These clusters occupy the same binding site, with the cytochrome *c* protein rotated 180 degrees about the axis between the proteins. The "native" structure (Figure 6.6d) has a DRMS of 3.8Å ±0.5Å and a minimum DRMS of only 0.5Å. The rotated structure has a DRMS of 5.2Å ±0.6Å and a minimum of 2.9Å. An additional cluster locates an alternative binding site, also located by Kim and Hummer, with an associated DRMS of 11Å accounting for 2% of the simulated complexes. Kim and Hummer discover this site with a DRMS of approximately 12Å [12].

DRMS values of 2.4Å for UIM/Ub and 3.8Å for the near native Cc/CcP clusters agree qualitatively with those of 2Å and 4Å for the same structures [12]. The ability of our simulations to reproduce such structures and binding characteristics convinces us that it is replicatively valid. In the case of UIM/Ub, the presence of flexible linkers in the UIM protein allows simulations to discover bound conformations with lower energy and DRMS than in our case.

6.3 Summary

In this chapter, our implementation of replica exchange Monte-Carlo simulation is validated by first assessing the critical operations composing the simulation before comparing our output with known results, to verify that our implementation functions correctly.

The interaction potential calculations on both the CPU and GPU are validated against the values generated by Dr Robert Best using Kim and Hummer's CHARMM implementaton. The average relative error between our calculations and the reference values is calculated to be 0.0014, which is within acceptable limits. Our CPU and GPU calculations can be considered equivalent as the relative error between them is, at worst, in the order of 10^{-6} for our reference simulations. We also show that accumulated error behaves favourably as problem size increases when compared to the CPU, thus allowing for accurate simulations of large complexes.

Validation of the Monte-Carlo mutations shows that our implementation rotates and translates molecules with sufficient accuracy to preserve structure when double precision operations are employed over single precision. For this reason, double precision rotation operations are used exclusively.

The outputs of our simulations of UIM/Ub and Cc/CcP at various concentrations are shown to agree Kim and Hummer's results and the supplementary data provided by Best. These results show that our simulation, at equilibrium, produces accurate results, which implies that the integrity of the simulation data is preserved and that calculation of the interaction potential is sufficiently accurate.

Finally, we illustrate that our simulations are capable of producing near native structures with low DRMS and energy values. The discovery of such structures, in agreement with Kim and Hummer.

Chapter 7

Interaction Potential Performance

We performed profiling and benchmarking of our application to quantitatively evaluate the benefits of using GPU technology for acceleration of our simulations, with the ultimate goal of maximising simulation speed-up.

Speed-up is measured as the quotient of serial time over parallel time, producing a factor, which, when greater than one indicates an improvement in performance and when less than one indicates a decrease in performance. In parallel computing the execution time of an application, T, is calculated as the sum of the time taken to execute the serial and parallel components,

$$T = T_{serial} + \frac{T_{parallel}}{N}$$

implying that T_{serial} will run in a mutually exclusive fashion to $T_{parallel}$. Thus, speed-up is achieved by dividing the parallel work among numerous processors, N. In doing so, the overall runtime decreases. Traditionally, N would be the number of CPU's or nodes on which the code runs.

In replica exchange Monte-Carlo, the relationship between parallel and serial code occurs at two levels. At the highest level, each Monte-Carlo simulation is independent, thus the simulations can be divided into serial replica exchange and parallel Monte-Carlo. At the second level, each Monte-Carlo simulation is a serial Markov chain containing a parallel interaction potential kernel. This chapter evaluates the performance of the lowest level of parallelism, the interaction potential calculation, seeking an optimal configuration of the GPU kernel while the following chapter evaluates simulation performance using higher level parallelism.

We begin by profiling our application to determine the time costs associated with each part of the algorithm and the effects of input size on runtime. This identifies the portions of the algorithm requiring most optimisation. Inspection of the algorithm indicates that the calculation of the interaction potential is the most computationally intensive aspect, accounting for the majority of simulation runtime. The profile also identifies the maximum amount of speed-up for a simulation.

Concomitantly, we benchmark simulation times for a representative set of simulations on a CPU only, providing a baseline against which all subsequent simulation configurations can be compared and evaluated.

There are a variety of ways in which to configure a GPU kernel, as discussed in Chapter 5. Certain parameters, due to the manner in which they map the algorithm to hardware, will result in better performance than others. It is difficult to determine an optimal parameter set by inspection, necessitating the use of benchmarks to determine this configuration. This task is further complicated by multi-threading on the CPU and the use of asynchronous computation on the GPU.

Each permutation of kernel configuration is benchmarked independently to determine an optimal configuration, which is then used to evaluate the effects of multi-threading and asynchronous GPU computing in Chapter 8. This chapter specifically inspects the performance of the interaction potential calculations determining the optimal block, and memory usage model for the implementation.

The system configuration used for profiling and benchmarking, in this and the next chapter, is included in Table B.1 and the proteins participating in each simulation are from those listed in Table B.2. Eight simulations are used for performance evaluation: Vsp27 and Ubiquitin (UIM/Ub) simulation of 100 residues (24 and 76 for each protein, respectively) contains 1824 pairwise interactions, Yeast cytochrome *c*/cytochrome *c* peroxidase (Cc/CcP) simulations of 402 residues (108 and 294 respectively, 31752 pairs) and Hepatitis B virus capsid proteins (HBV) simulations of 2 (568 residues, 80656, 4 (1136, 483936), 8 (2272, 2258368), 16 (4544, 9678720), 24 (6816, 22261056) and 27 (7668, 28310256) instances of the capsid pieces. UIM/Ub and Cc/CcP complexes, provide two small scale simulations, neither of which are large enough to saturate the system resources on both the CPU and GPU. Although small, these complexes are well studied using the Kim and Hummer model and are representative of simulations between only two participating proteins. HBV, allows us to benchmark biologically relevant structures on a linear scale from 568 up to 34080 residues, in increments of 284 and simulates performance in a multi-protein complex study.

7.1 Interaction Potential Calculations

The interaction potential is calculated as the sum of each pairwise interaction potential between all residues in each molecule. In the context of benchmarking this is referred to as the kernel. Using the CPU to calculate the overall interaction potential requires no data preparation, molecules and residues are in contiguous arrays in system memory, as described in our implementation chapter.

For each data point, the average of 20,000 individual kernel invocations is taken to be the runtime of a specific kernel.

Figure 7.1 shows the average kernel execution for a range of problem sizes. Note that this is the average time to calculate a single interaction potential for one replica. The $O(N^2)$ complexity of the operation is evident: the time required for the calculation of the total interaction potential on the CPU grows quadratically with the number of residues. The CPU implementation scales predictably for problems of this size, for 100 residue simulations, the interaction potential can be calculated in as little as 1 millisecond, while the same operation takes almost 14 seconds for 7886 residues.

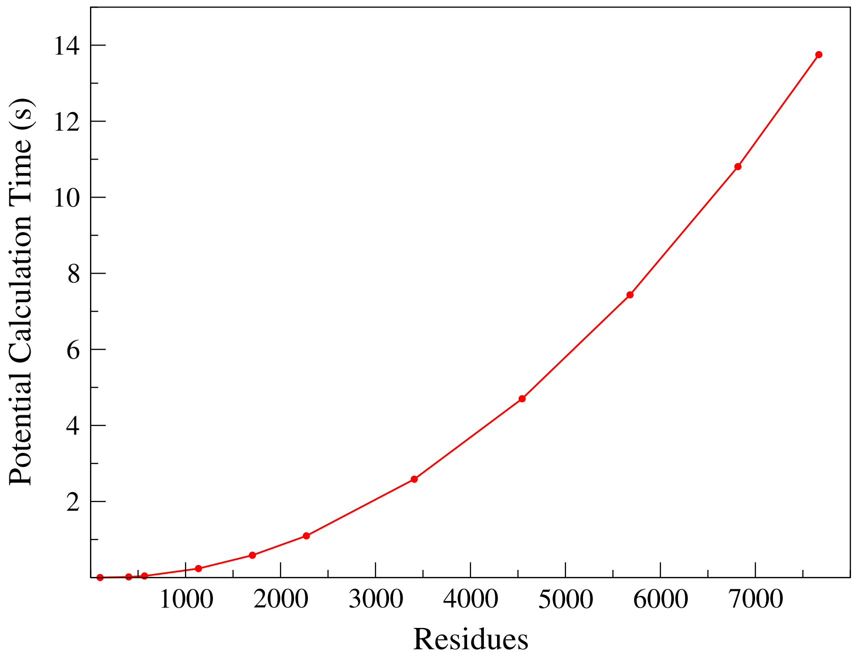


Figure 7.1: CPU Kernel Calculation Time
The average CPU time required to calculate the interaction potential sum for increasing simulation sizes from 100 to 7668 residues.

Profiling the simulation with respect to time indicates the effect of the interaction potential on the overall runtime of the simulation. Consequently, a theoretical calculation of maximum speed-up can be made with this data. Initially, profiling is performed without a GPU, providing a performance baseline against which all future benchmarks are compared.

We only consider time inside the *REMCSimulation* for profiling. The Monte-Carlo searches are called within POSIX threads called by the simulation function. However, these threads are created once and reused for the duration of the simulation, waiting while the replica exchange part of the algorithm executes. For profiling, the Monte-Carlo searches are performed in a single thread and profiled using *TAU*, while the parent thread is timed using the *cutil* timers (the C utility provided in the CUDA SDK).

The modularity of our code means that the blocking *REMCSimulation* function call is the only part of code that is profiled, all code outside this function executes in comparatively negligible time.

Monte-Carlo simulations are computationally demanding: for the smallest simulation of only 100 residues, for each second of Monte-Carlo simulation, far less than 1 millisecond is required for replica exchange. This is expected, as our implementation of replica exchange only exchanges counter variables and temperature values, resulting in almost zero memory transfer overhead. The time required for replica exchange is $O(\frac{R}{2})$, involving iteration over the list of replicas and switching distinct adjacent pairs for each exchange step, where R is the number of replicas in the simulation. By contrast, the Monte-Carlo simulations are dependent on the size and number of molecules per replica: each Monte-Carlo rotation or translation requires an operation on each of the residues of the selected molecule, meaning it is an operation of linear complexity. Evaluation of this move requires that the new interaction potential be calculated, an operation that is approximately $O(\frac{1}{2}n^2)$, where n is the number of residues, as it requires a comparison between each distinct pair of residues from distinct molecules in the replica.

Saving and restoring molecules in the Monte-Carlo simulation is achieved using a simple memory copy, and can therefore also be considered linearly dependent on the size of the mutated molecule.

Profiling analysis allows us to estimate, assuming a pairwise comparison and a mutation on a single residue are of approximately equal cost, how our application will perform serially. Using the ratio of interaction potential pairs over the combination of the interaction potential pairs and the average number of mutation operations performed per Monte-Carlo step, we estimate that the interaction potential will require over 97% of all simulation time. Table 7.1 shows that, once measured, the estimated percentage CPU time used for interaction potential calculations increases from 97.332% to 99.999% with ascending benchmark size. This is in agreement with the benchmark results. Clearly, the ratio of the interaction potential execution time to all other code is so large that, for even small problems, no optimisation of code other than interaction potential will be of significant benefit.

Within each Monte-Carlo loop, computation time can be divided between interaction potential calculations, mutations (either translation or rotation), saving and its complementary restoration operation and the Monte-Carlo specific operations such as acceptance/rejection and random number generation. Broadly, these are grouped as interaction potential, rotation, translation, save and other operations for profiling. Figure 7.2 shows that the contribution of computational operations is similar relative to the cost of calculating the interaction potential. The *memcpy* reliant save and restoration proves least expensive, approximately an order of magni-

Table 7.1: Interaction Potential CPU Profile

Estimation of the proportion of time required for interaction potential calculations. Systems of Vsp27 and Ubiquitin (UIM/Ub), Yeast cytochrome c/cytochrome c peroxidase (Cc/CcP) and Hepatitis B virus capsid proteins (HBV) require more computation time as the number of residues in the system increases. Estimated and actual proportions of runtime prove similar.

Molecules	Residues	Interaction Pairs	Ave Mutations	Estimate %	Actual %
2	100	1824	50	97.332	98.030
2	402	31752	201	99.371	99.702
2	568	80656	284	99.649	99.796
4	1136	483936	284	99.941	99.922
8	2272	2258368	284	99.987	99.983
16	4544	9678720	284	99.997	99.996
24	6816	22261056	284	99.999	99.998

tude faster than mutations.

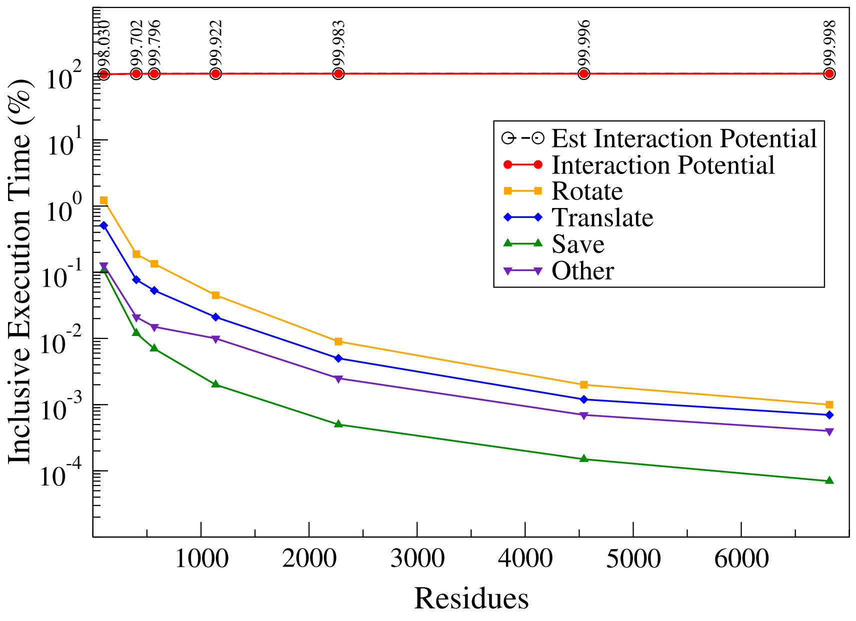


Figure 7.2: CPU Inclusive Profile

CPU profiling indicates that the interaction potential accounts for over 98% of the time required for simulations. The Monte-Carlo simulations are profiled by measuring the time required for the interaction potential calculation, rotation, translation, saving (if a MC move must be reversed) and other operations such as random number generation.

Potentially, we could port mutations to the GPU, as many of the floating point operations performing both rotation and translation map to fused multiply add (FMA) instructions. However, this would remove all computation from the CPU, resulting in underutilisation of the CPU in the asynchronous CPU-GPU case.

If a single Monte-Carlo simulation is performed, it would be faster to perform all operations on the GPU since there would be zero memory overhead. But, this would prove problematic considering our findings regarding the accumulation of inaccuracy in the rotation operation using floating point arithmetic on the GPU (see Section 6.1.2).

7.2 Interaction Potentials on the GPU

Referring back to the CUDA Best Practices Guide [23], performance of a GPU code is dependant on three overall optimisation strategies: maximising parallel execution, optimising memory usage and optimising instruction usage to achieve maximum throughput.

In implementing the interaction potential kernel, these criteria are followed. Algorithmically, the tiled approach to the n-body problem is known to provide optimal memory bandwidth for the underlying problem [17,18,25,26] and the optimised CUDA reduction solves the summation of each body's contribution to potential [82]. The instruction usage strategy is largely invariant with decisions regarding the choice of instructions determined by theoretical knowledge of the GPU. But, parallel execution and memory bandwidth characteristics are more dependent on the problem and data, because they are sensitive to latency and execution parameters.

In benchmarking the interaction kernel we evaluate the influence of three factors: the location of the contact potential lookup table, thread block size and occupancy. The contact potential lookup table can reside in shared, global, constant or texture memory on the GPU. Characteristically, shared memory is fast but scarce, constant memory caches values and is best used to broadcast constants and frequently used values, global memory is not cached and has very high read/write latency and, finally, texture memory caches values spatially local to the value accessed, affording better memory performance to threads with non-coalesced access patterns. Benchmarking the performance of kernels using these types of memory will illustrate the efficacy of each type of memory for the purpose of random access lookups from a table such as ours.

The relationship between all three factors is more difficult to determine. Thread block size determines occupancy because the amount of shared memory on each SM must be shared between all the threads in a block and all the blocks on an SM. Occupancy, in turn, determines the amount of latency hiding within a thread which ultimately affects the performance of the lookup table since more threads will lessen the effect of a bad access pattern. Also, SMs limit the number of warps per block and the number of thread blocks per block. Hence, smaller thread blocks mean fewer warps, which means lower occupancy. For example, a thread block size of 32 threads per block will mean that 8 blocks and 8 warps will execute. This is only 25% of the maximum number of warps allowed on a GT200 SM, but the maximum number of allowable thread blocks. Raising the block size to 64 immediately results in better occupancy because now 16 of a 32 possible warps execute concurrently. Using shared or texture memory for the residue data changes the optimisation problem again. In this case, occupancy can be raised because there is less dependence on shared memory.

Due to the use of reduction in our kernels, thread block size may be any power of 2, up to 512. For practicality, we benchmark kernels of 32, 64, 128 and 256 threads per block as they are all divisible by 64 as per the programming guide and powers of 2, which allows us to use nVIDIA's optimised CUDA reduction [82]. Regarding addressability, the number of threads per block imposes a limit on the maximum number of addressable residues due to CUDA's grid and block size limits and our algorithmic tiling. The limits are 8192, 16384, 32786, 65536, 131072 for 32, 64, 128, 256 and 512 threads, respectively. Consequently, the optimal configuration of kernel may be limited by the problem size dictating the minimum number of threads per block.

Benchmarking uses the following approach to evaluate the performance of each of the aforementioned factors.

We configure kernels to use either shared memory, texture memory or a combination of the two types to store a low latency cache of residues as per the CUDA n-body model [17,18,25,26]. This allows us to determine the deterministic characteristics of our implementation without random contact potential lookups and thus determine the optimal memory usage model before introducing the lookups. The cost of a lookup should theoretically be constant as there is exactly one lookup between each pair of residues and a constant cost proportional the number of residue pairs. This benchmark provides a baseline against which to compare the effects of the lookup table.

The random lookups for each contact potential can be configured to use texture, constant, global or shared memory in our kernels. With the data from the deterministic part of the interaction potential calculations, we can evaluate the effects of introducing the random lookup and the performance implications of the type of memory used for it.

Finally, the best memory performance cases are paired with either using shared, texture or a mixture of texture and shared memory for residue data with the aim of increased occupancy. The results of which are compared to the theoretical best case of shared memory for the residue cache and texture memory for the lookup table. The *CUDA Occupancy Calculator*[1], is used for this purpose.

[1]included in the CUDA SDK

7.2.1 Shared vs. Texture Memory Caches

To access the underlying kernel performance, we benchmark our kernels without the contact potential lookup, replacing it with e_0 (Equation 5.3.5). This means that no lookup will occur, allowing us to benchmark only the performance of shared or texture memory when used as a pre-fetched cache for the residues. The purpose of this benchmark is to provide a baseline value to show clearly how the introduction of the random lookup changes performance characteristics.

Our GPU-accelerated kernel comprises calculation of the interaction potential of a replica. This involves zeroing the memory which the CUDA kernel writes back to the host, launching the CUDA kernel and performing a final reduction on the CPU. As already shown in the CUDA SDK reduction example, reduction on a GPU is only worthwhile for very large datasets. Therefore, we perform the final reduction on the CPU, which is limited to a maximum of 65535 elements (2^{16} residues), as CPU and GPU implementations of this function are comparable in this context.

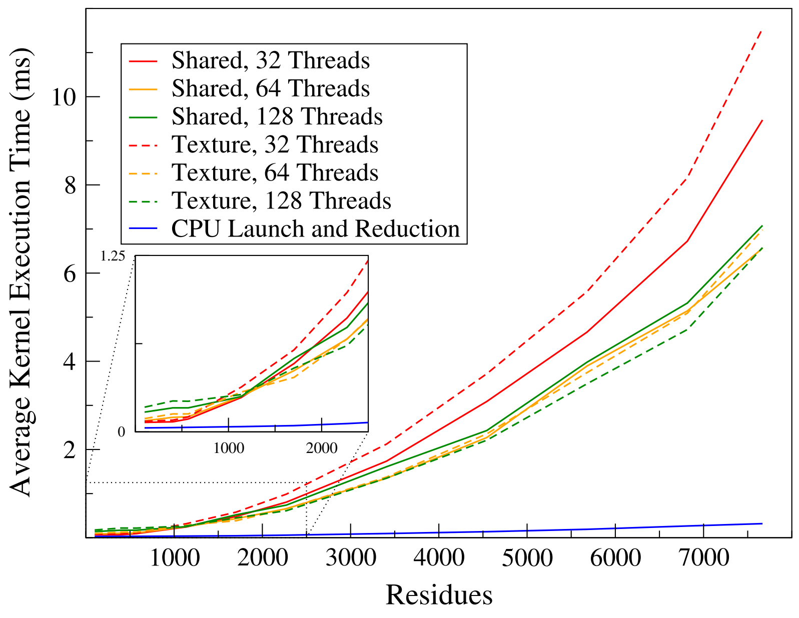


Figure 7.3: Baseline Kernel Performance
(a) Generally, the performance of 64 or 128 threads per block using either texture or shared memory for residue caches is best because of the performance of shared memory or the latency hiding and higher occupancy due to using texture memory. (b) For smaller simulations, 32 threads per block and shared memory for the residue cache is best due to an inability to latency hide because of the maximum occupancy of only 25% when using either texture or shared memory. 32 threads per block also results in more concurrent blocks being scheduled on the GPU, and hence greater throughput at low occupancy because more SM's will be active than if more threads per block are used. For all problem sizes, the host bound portion of the interaction potential is increasingly less significant with problem size.

Invocation of a CUDA kernel takes in the order of 30 microseconds with the additional calculations required to configure the kernel before launch being trivial (and of constant cost). The host functions require from 0.03 milliseconds to 0.32 milliseconds of the kernel time, accounting for 34% of the runtime for a 32 thread-per-block kernel (the fastest kernel for small simulations), this includes the time required to perform the CPU reduction. As simulations increase in size, the runtime required by the host increases at a slower rate than the time required for the pairwise interaction calculations on the GPU and tends to account for only 4% of runtime for the largest simulations (figure 7.3). We conclude that the proportion of interaction potential performed on the host scales favourably with problem size.

CUDAProf reports that there are no non-coalesced access patterns to global memory for this kernel type, indicating that it minimises the time required to read residue data from global memory. The GT200 architecture relaxes the conditions under which coalescing occurs, allowing a programmer to use arrays of type *float4*, which we do. G80 architectures are more strict and require 4 arrays of type *float* [22]. We also implement this manner of storing residue data, but find it to be slower than using float4 to store each residue's variables as it requires eight 32-bit loads versus two 128-bit loads per thread when initialising a block.

The two types of GPU memory for residues perform comparably (Figure 7.3). Shared memory only convincingly outperforms texture memory in the case of 32 threads per block. This is due to a combination of occupancy and block-level parallelism, for example, a simulation of 512 residues will be divided into 16 thread blocks for 32 threads per block, 8 if 64 threads per block are used and only 4 in the case of 128 threads per block. The block-level parallelism of 32 threads per block is higher than the others, resulting in improved performance for small problems because the simulation physically runs over more SMs. Additionally, 32 threads per block results in only one warp per block, eliminating the synchronisation steps required for reduction when more than one warp is present. This mean that reduction is more efficient in the case of 32 threads per block, with decreasing efficiency as block size increases.

When more threads per block are used (64 or 128) then the occupancy of the kernel becomes significant for more than 1000 residues. At this point, kernels using only texture memory achieve higher occupancy and consequently more latency hiding and can therefore perform in similar, or in some cases, faster time than kernels using shared memory. Once the degree of block level parallelism saturates the GPU, occupancy is important in determining performance and the block sizes of 64 and 128 threads perform better than those of 32 or 256 for this reason. Note that the performance of 256 threads per block is not included in the figure as it is worse than that of 32, 64 and 128 threads in all cases. The use of 256 threads per block is unsuitable unless simulations are so large that they require 256 threads to address all residues.

These results hinge on the effect of occupancy. Our kernel requires 36 bytes of shared memory per thread, 32 bytes for each residue it loads into the tile and 4 bytes for its accumulator. We note that no configuration can achieve greater than 50% occupancy due to the shared memory requirements of this application. For block sizes smaller than 32 threads, the hardware limit of

8 warps per block, limits the occupancy of the GPU to 25% in all cases. For 64 or more threads per block, shared memory is the limiting factor. Using 64 threads per block allows 6 blocks to execute concurrently on a multiprocessor compared to 3 blocks per multiprocessor when using 128 threads per block. Our configuration limits the hardware to 8 blocks per multiprocessor, resulting in 37.5% occupancy for both 64 and 128 threads per block. 256 threads require over half the amount of shared memory on one SM, consequently only one block can run per SM. Because an SM can schedule 1024 threads concurrently, this results in 25% occupancy.

Using texture memory for the residue cache raises the occupancy of each thread block configuration, with the exception of 32 threads per block. This sees 64 threads per block rise to 50% occupancy and both 128 and 256 rise to 75% occupancy, limited now by the number of registers on the SM as opposed to shared memory.

Two observations arise from this benchmark. Firstly, the use of 64 threads per block will give good general performance across the entire range of simulations. Secondly, the principle of using shared memory wherever possible in CUDA is not necessarily optimal for every problem. In our case, pairwise interactions implicitly exploit the locality of the 1D texture cache in exactly the same manner that shared memory does when managing this cache explicitly. Thus, other algorithms which require more shared memory than a 16 byte per body case (gravitational or electrostatic simulations) stand to benefit from using texture memory over shared memory, with this becoming increasingly true as the memory requirements per body increases.

The introduction of the random lookups will change these benchmarks considerably. The lookup table has to reside somewhere on the GPU die (in shared, texture or constant memory), otherwise the latency would be too great when accessing values. Shared, texture or constant memory will be essential for this reason. This, in turn, will increase the memory bound features of the algorithm because now both residue and contact potential values need to be sent to registers on the SM on the same data buses. This aspect is discussed at a later stage in section 7.2.3 when combinations of memory are used to achieve both occupancy and performance cognisant of these limitations.

7.2.2 Lookup Table Memory Performance

We now report the performance of various kernels, each using different types of GPU memory for the contact potential lookups. We begin with texture memory, as we suspected that it would perform best of the four due to its tolerance of random memory access patterns. There is exactly one random contact potential lookup per pairwise interaction, which should add a constant time to each pairwise interaction, as determined by the memory type, to the baseline benchmarks. For the following benchmarks, we opt to use shared memory for residues because it is guaranteed to be faster when performing the coalesced memory accesses used to retrieve residues.

There are four choices of memory location for the lookup table: texture, global, constant and shared memory.

Texture memory caches an 8 Kb patch of linear memory [2], meaning that a residue interaction lookup will cache all the lookup values in the 1600 byte table on the first read. The texture cache per SM is the same size as the amount of shared memory per SM, 16 kilobytes, meaning that the remaining 90% of this cache can be used for residue data if we do not employ shared memory.

Constant memory caches up to 64 Kb of values accessed by a thread block. Unlike texture memory, which caches spatially local values, constant caching caches singular values. For this reason, we expect texture memory caching to outperform constant memory caching because, assuming a completely general protein structure, any combination consisting of the 20 amino acid types could occur in one thread block.

Global memory is not cached, relying on the latency hiding effect of many threads. By using global memory for contact potential lookups, we incur an overhead of hundreds of clock cycles for every lookup. In the case of the constant memory kernel, this would only be on the first occurrence of a particular lookup. Therefore, the performance of a global memory lookup kernel indicates the effectiveness of the constant memory lookup kernel.

Shared memory is explicitly managed and banked for performance, the upside to shared memory is that it is as fast to access as a register under the right circumstances, namely, coalescing. Unfortunately, before use, values need to be explicitly loaded into shared memory. For this to be efficient, the dimensions of this load need to be functionally dependent on the thread block size, i.e, each thread loads t values in a coalesced manner with the other threads. The second downside to shared memory is that it is scarce on the GPU, meaning were each thread block to store a 1.6 Kb table, very little would be left for residue data, decreasing occupancy.

We begin with texture memory benchmarks. Figure 7.4 shows that thread blocks configured to use 64 or 128 threads per block perform best for more than 1000 residues with kernels of 64 threads executing between 5% and 10% faster than 128 threads for larger problem sizes. Due to block size, 32 threads per block performs best for our benchmarks of less than 1000 residues (up to 584), executing between 12% and 31% faster than the 64 threads per block configuration and twice as fast as 128 threads per block. This repeats the results of the baseline benchmark. Overall the kernel using 64 threads per block is about 12% slower than when no lookups are performed.

[2]CUDA 2.1 FAQ

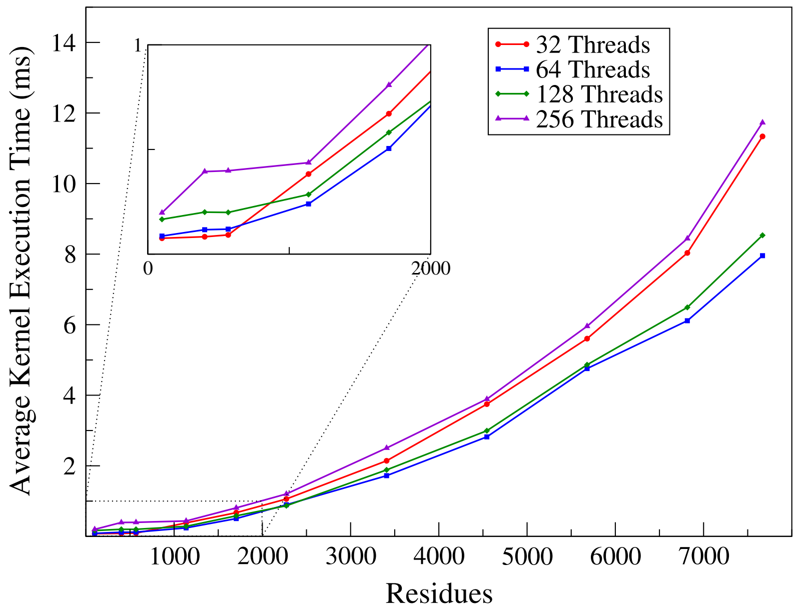


Figure 7.4: Texture Memory Kernel Time
Benchmarks using texture memory are performed for simulations from 100 to 7668 residues. For fewer than 1000 residues, block-level parallelism proves most important. Larger simulations hide the latency of the lookups, with higher occupancy resulting in better performance.

A small block size of 32 threads per block performs best for the smallest benchmarks, this configuration requires less padding and affords the most parallelism to a problem too small to benefit from latency hiding through multiple blocks. A configuration of 32 threads per block is capped by the limit of 8 warps per SM, and can thus only achieve 25% occupancy unlike the 38% occupancy of 64 and 128 threads, which are limited instead by the amount of shared memory on the SM. Shared memory also constrains 256 threads per block to 25% occupancy. The effect of occupancy for each configuration is evident in Figure 7.4, with 32 and 256 threads performing approximately 50% slower than 64 and 128 threads. For fewer than 1000 residues, 32 threads per block provides the best balance between block-level parallelism and padding. Conversely, 256 threads per block confines computation to fewer SMs and over pads simulations, resulting in poor performance at the bottom end of the scale.

The order in which the texture is addressed does not change performance, indicative of the fact that the entire texture is cached when one value is read. Each texture unit for a 1D texture will cache an 8Kb line of values, since our lookup table is only 1.6Kb, we always cache the entire table. But, theoretically, if the contact potential table were larger than 8 Kb, the cache size, addressing the table using a row major scheme ($T[nx+y]$) would result in cache hits for residue x because we iterate over y residues. Column major addressing ($T[x+ny]$) would result in each lookup belonging to a different cache line in the texture, degrading the performance.

Subsequent benchmarks will be compared to texture memory lookups for 64 threads per block (Tex64). This configuration offers good generic performance across the entire range of benchmarks, marginally outperforming 128 threads per block.

Conceptually, the only difference in using constant and texture memory for contact potential lookups is the degree of caching. While texture memory caches values spatially local to the initial read, constant memory only caches singular values on first use. This is acceptable if all lookups are of the same type or broadcast. Unfortunately, they are likely to differ much of the time. Benchmarking results confirm this finding with the use of constant memory for pair potential lookups performing slower than the equivalent configuration using texture memory to perform lookups in all cases. A block size of 64 is again fastest for larger problems.

As with texture memory, no cache hit can occur on the first iteration of a thread block, but, in the worst case there is an extremely low chance of a cache hit occurring on the second iteration, improving as the block iterates over all residues, resulting in performance illustrated in Figure 7.5. But, due to the random access nature of the lookup, warps are required to serialise the memory transactions to fetch the contact potentials. The effects of this are most evident for simulations larger than 1000 residues. Thread blocks of 32, 64 and 128 threads require approximately 20%, 14% and 12% more run time than the texture memory equivalent, respectively, with larger thread blocks being more likely to cache a value, resulting in performance closer that of texture memory. Once again, 256 threads per block is limited by occupancy, impairing its ability to latency hide and make use of its higher likelihood of cache hits. This kernel is approximately 8% slower than its texture memory equivalent, but still slowest overall.

The latency hiding effect is evident in thread blocks of either 32 or 64 threads when dealing with 1704 residues or more, since constant memory manages to perform, on average, only 6.5% slower than texture memory in these cases.

When compared to using global memory, (Figure 7.6), it becomes apparent, that the use of the GPUs constant cache fails to perform significantly better than merely loading values from global memory on demand. Although constant memory kernels outperform global memory kernels in all but 2 of the 44 benchmarks the differences are not significant. Generally, the lack of caching is seen for smaller block sizes when the simulation size is small (1136 residues or less). In these cases, global memory usage is approximately 13.5% slower than using texture memory compared to a more favourable 6.5% performance deficit when using constant memory. For the remaining cases, global memory is less than 2% slower than constant memory. For all block sizes of 32, 64 and 256, global memory exhibits near identical performance to constant memory.

The sensitivity of a kernel to occupancy is more noticeable when using either global or constant memory. For constant caches to speed-up the kernel, kernels must attain higher occupancy such that the constant cache is as accessed as many times as possible. This highlights that, even though the randomness of the lookups is identical whether constant, texture, shared, or global memory is used, the texture units on the GPU better accommodate such an access pattern.

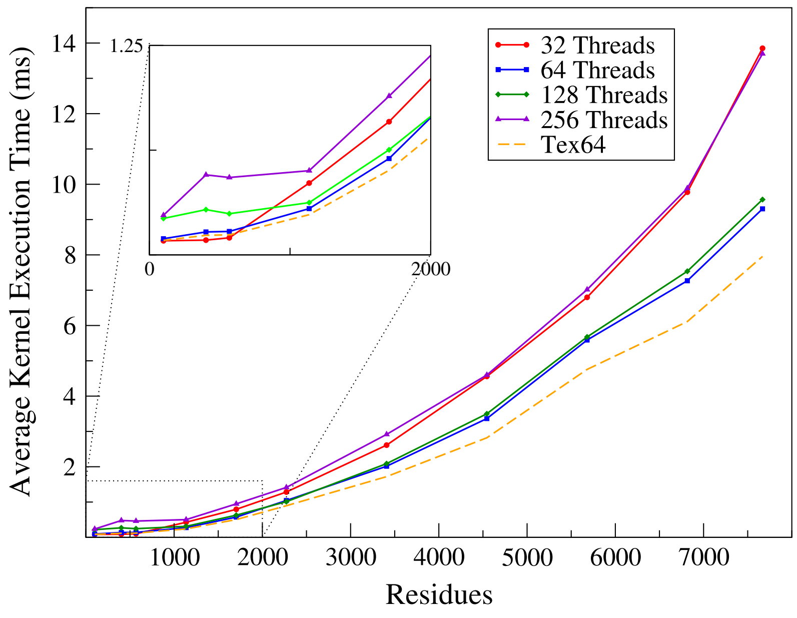


Figure 7.5: Constant Memory Kernel Time
Constant memory performs approximately 10% to 20% worse than texture memory when performing contact potential lookups. Tex64, the texture lookup configuration of 64 threads per block shows the comparative difference between the different types of memory. Notably, the performance profile of each thread block configuration is the similar to texture memory.

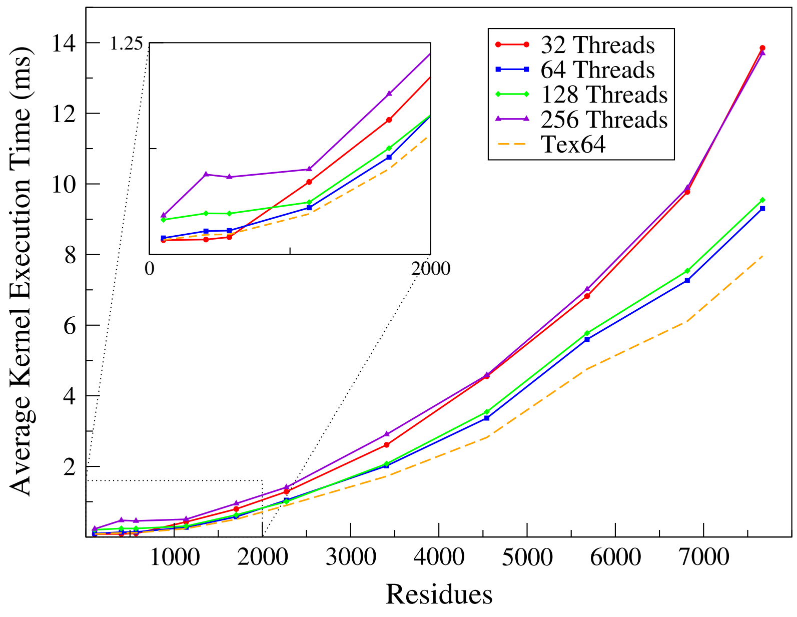


Figure 7.6: Global Memory Kernel Time
The performance of global memory is almost identical to that of constant memory (fig 7.5), differing on average by less than 2%. This indicates a very low cache hit rate for the lookup table.

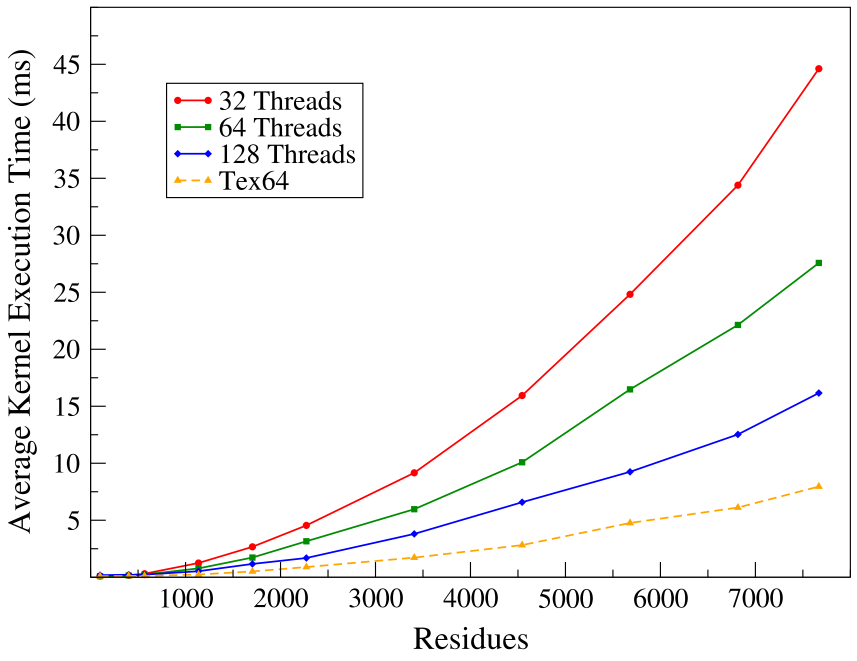


Figure 7.7: Shared Memory Kernel Time
Shared memory lookup performance is determined almost entirely by occupancy, with performance 5, 3.4 and 2 times slower than the fast Tex64 kernel.

While performance remains similar in the cases of texture, constant and global memory, it degrades severely if shared memory is used for the lookup table. Shared memory is ill suited to storing the lookup table because its dimensions are independent of, and indivisible by the thread block parameters. Each thread block must copy its own instance of the lookup table into shared memory, and the consequent smaller thread block size results in lower occupancy and lower performance due to this operation accounting for a greater proportion of runtime. Second, shared memory is shared between all the blocks on an SM, thus, configurations already limited by shared memory experience even lower occupancies. Thread blocks of 32 or 64 threads achieve 6% occupancy and thread blocks of 128 threads 13%. A kernel of 256 threads per block cannot run as there is insufficient shared memory to initialise such a kernel. For our smallest problem size of 100 residues, the use of shared memory for the lookup table is comparable to the other types of memory, but as the problem size grows, the performance of shared memory degrades. When compared to our fastest kernel using texture memory, configurations of 32, 64 or 128 threads per block are, on average, 5, 3.4 and 2 times slower, respectively (Figure 7.7).

7.2.3 Thread Blocks and Occupancy

A feature of all kernels using the shared memory cache for residue data is low occupancy, with the best kernels only achieving 37.5% due to the scarcity of shared memory. A strategy for increasing occupancy would be to use constant or texture memory for meta-data lookups, making thread blocks less dependent on shared memory at the cost of higher latency. This exploits the implicit caching of texture memory in place of the explicit shared memory cache. An occupancy of 50% is achieved for 64, 256 and 512 threads per block and an occupancy of 63% for a block of 128 threads is achieved using this strategy. The change also enables use of the maximum allowable thread block size of 512 and raises the addressable residue limit to 131 072 residues for a single kernel. Previously, shared memory limited thread blocks to a maximum size of 256 threads, resulting in a limit of 65 536 residues per kernel.

Constant memory can immediately be discounted for this purpose due to the fact that it would perform as badly as global memory in this usage pattern, something all other algorithms in MD and gravitation avoid due to high latency [15–18, 25, 26].

This leaves texture memory as the only other option to shared memory. Texture memory, because of its cache, achieves the same objective as shared memory: values are cached to avoid the high access latency of global memory. We implemented texture arrays of residue data, in part or entirely. Splitting the data between an array containing the position and molecule identifier and another containing charge, radius and type. The downside to this strategy is that this may rely too heavily on the four texture load/store units on each SM, essentially causing a 4 way access conflict within each half warp, in addition to the serializing the table lookups.

By using texture memory for all residue data on the GPU, the occupancy of 128 and 256 threads per block can be raised to 75%. But as illustrated in Figure 7.3, the performance of 128 threads per block becomes notably worse when shared memory is no longer exploited. This is also the case for 256 threads per block which is consistently slowest. In this section, kernels are referred to using *lookup table location, threads, residue position/data location* for brevity. For example, *Tex, 64, shared/shared* means a texture lookup table, a thread block size of 64 and all residue data cached in shared memory.

A search of block and memory parameters in the range of 32 to 256 threads per block and the choice of either texture or constant memory for the lookup were benchmarked with either some or all of the residue data in texture memory. Figure 7.8 shows that through higher occupancy alone, each manages to perform as well as the theoretical best fit model of using only shared memory for residues and texture memory for lookups. Included in the plot is the best case *Tex, 64, shared/shared* (Tex64) benchmark which represents the 64 thread per block kernel using texture lookups and shared memory for residues which achieves 37.5% occupancy.

Generally, configuring a kernel by specifically trying to increase occupancy results in an improvement in runtime for most kernels with the exception of our Tex64 kernel. For simulations larger than 2000 residues, we achieve compatible performance from kernels using texture or constant memory for the lookup data. This is largely do to with the increase in occupancy and the ability to latency hide texture fetches. By using texture memory for either residue position

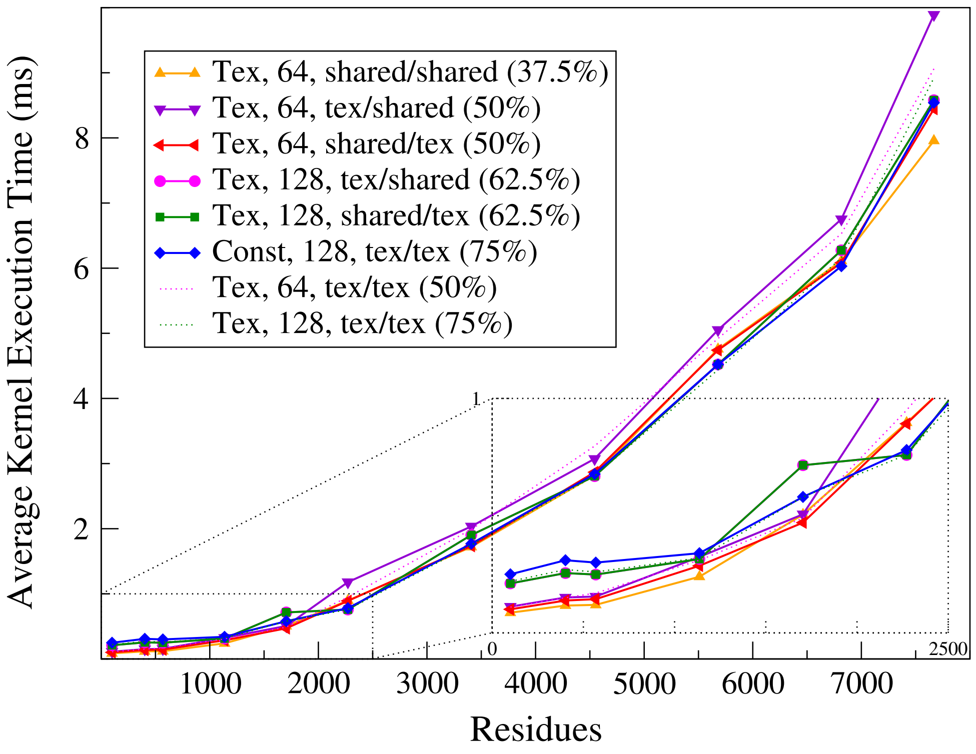


Figure 7.8: Alternative Kernel Memory Configurations
The performance of kernels using texture memory for residue data. Kernels are keyed according to the location of the lookup table, the number of threads per block, the location of the residue arrays and occupancy.

and molecule identifier, configurations of 64, 128 and 256 thread per block experience respective increases in occupancy from 37.5%, 37.5% and 25% to 50%, 63% and 50%. By only using texture memory for residue data, both 128 and 256 threads per block rise to 75% occupancy.

Tex, 64, shared/shared performance is faster for less than 1000 residues, but almost matched by *Tex, 64, shared/tex* and *Tex, 64, tex/shared*, indicating that the block-level parallelism is more vital for small simulations, provided some type caching mechanism is used for residues. Also of interest is that configurations of 32 threads per block outperform 64 threads per block, but degrade much more for larger problems than their equivalent shared/shared configurations due to the 25% cap on occupancy for any configuration of 32 threads per block.

A kernel which experiences a distinct improvement from this strategy is the *Const, 128, tex/tex* kernel with 75% occupancy. Notably, this kernel uses constant memory for its lookups and texture memory for its residue caches and is counter-intuitively fast. Previously, the best performance for constant memory lookups was generally 10% to 16% slower than texture memory (Fig. 7.5). But the increase in occupancy from 37.5% to 75% appears to be the reason why the increase in performance occurs for larger problem sizes. For smaller problem sizes, less than 2000 residues, we observe that this kernel is slower than the original configurations, and in simulations smaller than 1000 residues, it is almost three times slower than a kernel using 32 threads per block with an occupancy of only 25%.

These characteristics are repeated when only texture memory is used with 128 threads, seeing *Tex, 128, tex/tex* and *Const, 128, tex/tex* perform almost identically.

The increasing latency of using texture memory for the position data is evident with 64 threads per block, with *Tex, 64, tex/tex* and *Tex, 64, tex/shared* performing slower than the original *Tex, 64, shared/shared* configuration.

Ultimately, all of the aforementioned configurations perform acceptably well for simulations of more than 1000 residues as they speed-up the CPU kernel by a similar amount in almost all cases.

In summary, the presence of the lookup table has less of an impact on the results than expected. While the lookup can be considered a random access pattern, it is confined to relatively few values. If a larger tale of 2000 values as opposed to 400 were used, we could expect to see performance equivalent to that of global memory lookups. Thus, the presence of the texture memory cache of 8 Kb per SM on the GT200 will be able to accommodate up to 2000 lookup values with equivalent performance to that of our kernel (figure 7.4). However, when we use texture memory for both the lookup table and residue data, the texture cache will have to be shared, resulting in a drop in performance. But, configurations such as this stand to benefit older hardware, such as the G80, which only has 8 Kb of shared memory per SM and would consequently only achieve half the occupancy of our kernels using shared memory caches. On this hardware, placing some residue data in texture memory and some in shared memory would prove beneficial.

The point at which optimal block size for a simulation changes from 32 to 64 threads per block is lowered by the lookup function from between 1000 and 1500 residues for no lookups (Figure 7.3) to between 750 and 1000 residues. The shift is solely due to the increased latency of the lookup and the latency hiding that larger systems afford. It also indicates that a dynamically selected block size, determined by simulation size must switch between 32 and 64 threads per block at this point. Using this strategy, we find that the lookup introduces an average overhead of 22% runtime for the majority of benchmark sizes with simulations of 2272 residues experiencing the worst performance degradation, requiring 41% greater runtime.

Generically, choosing a block size of 32 or 64 dynamically, based upon simulation size, caused the use of either global or constant memory for the contact potential to be at most 18% slower than using texture memory (figure 7.9) with alternative configurations of texture and shared memory for the residue caches performing with comparable performance to texture memory lookup and the shared memory residue cache. We note that the relative performance of shared memory using the scheme is over three times slower, prohibiting its use under all circumstances,

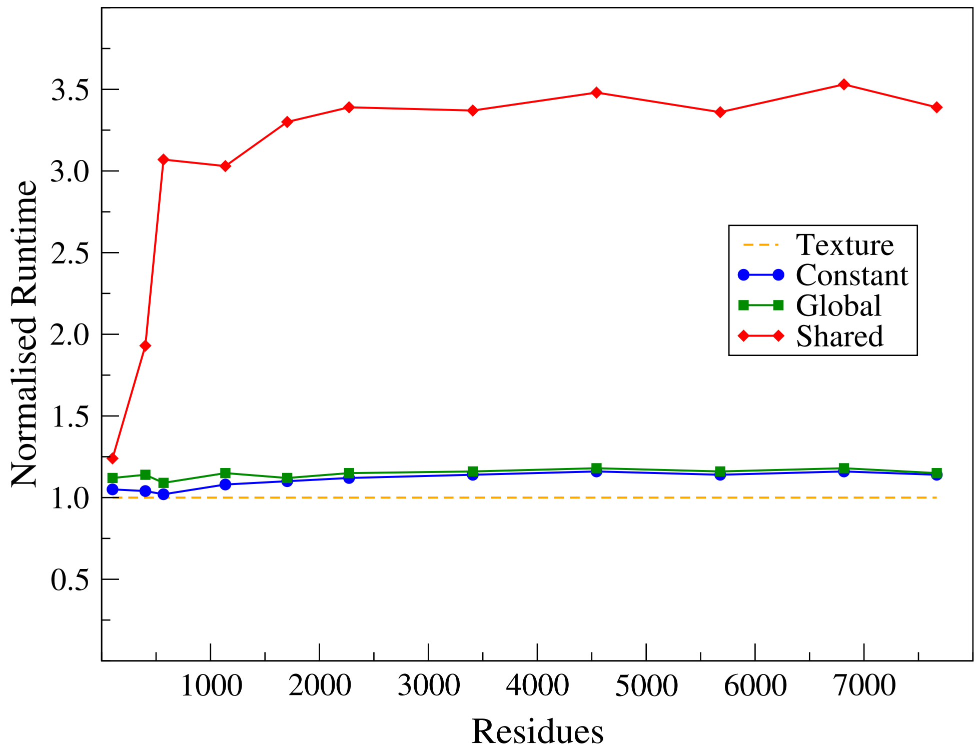


Figure 7.9: Relative Kernel Performance
The relative runtime of kernels by memory type relative to texture memory. Using either 64 threads per block for more than 1000 residues or 32 threads for fewer than 1000 residues, the relative performance of global and constant memory is never worse than 18% slower than texture memory. Shared memory is between 3 and 3.5 times worse using the same scheme for more than 500 residues.

apart from the very smallest simulations.

7.2.4 Data Transfer

A primary difference between using a GPU and CPU is the requirement that data be transferred from host to device before any computation can be performed. We only transfer updates to molecules on the GPU for every iteration of the Monte-Carlo simulation because GRAM is persistent for the duration of the simulation. The bandwidth bottleneck between the host and device is a known shortcoming in the GPU programming model. Thus, knowledge of how this bottleneck affects our algorithm is important.

To minimise the amount of data transferred across the PCI-Express bus, we restrict transfer to the position vectors of the mutated molecule only; meta data such as the amino acid type, charge, radius and identification is already on the GPU an remains unchanged for the duration of a simulation. We found that by only updating and individual molecule after each mutation resulted in transfer times of the same order of magnitude as the initialisation time required for each transfer. The amount of data transferred each time is at most 4704 bytes, the size of a cytochrome *c* peroxidase protein, taking in the order of 0.02 milliseconds, an effective transfer rate of less than 300 megabytes per second, far below the performance of the PCI-E 16× bus. This performance can be explained by the size of the transfer, a transfer of such a small magnitude will not provide an accurate measure of transfer speed, especially if the initialisation cost of the transfer is of the same order of magnitude as the transfer time itself (The CUDA bandwidth test example in the CUDA SDK 2.3 reports that our system performs larger transfers at 2 GB/s).

Kernel execution times put transfer times in perspective, with kernels requiring tens of milliseconds to execute compared to the 0.02 milliseconds required for transferring data, a difference of four orders of magnitude. This shows us that the transfer bottleneck between the host and device is not a determining factor in the case of our application.

7.2.5 Performance Discussion

The performance of our GPU kernel is comparable with that achieved by Friedrichs et al. [17] for similar molecular potential evaluations on the same GTX280 architecture (Figure 7.10). The potential evaluation in molecular dynamics (MD) map similarly to the GPU. However, there are some key differences in the Monte-Carlo (MC) algorithm that makes it more computationally expensive and accounts for the better performance of the MD code. Our kernel is required to perform a reduction of the n^2 pairwise potential to a single energy value (which is used to either accept or reject the MC move). Reduction requires synchronization before each of its iterations and performs one arithmetic operation for every two loads and one store, which impacts negatively on the kernel performance relative to the MD code. Reduction operations are memory bound with low algorithmic intensity and therefore have relatively poor gigaFLOPS performance on the GPU architecture. Conversely, the MD kernel includes an $O(n)$ integration step after the pairwise accumulation, the higher algorithmic intensity, separability of the force calculations and lack of synchronization is better suited to the GPU architecture and results in better GFLOP performance. Furthermore, we include the transfer and CPU reduction as part of our performance evaluation of the interaction potential, often performance measurements only consider the kernel invocation and execution on the GPU.

We count every mathematical operation in the kernel code between a pair of residues as one floating point operation. We determine the number of operations required to calculate a pairwise interaction and add it to the total interaction potential as being 45. This allows us to calculate the effective FLOPS achieved by our implementation. The actual FLOPS achieved would incorporate all control structures, parallel summation and compensated summation. however, the effective FLOP rating may be used as it measures the amount of useful work done as opposed to total FLOPS. For example, on the GPU square roots, exponentials and power functions are considered a single FLOP but on the CPU these operations require 15, 20 and 20 flops respectively [17].

Furthermore, we can only compare the FLOPS for interaction potential. That being the calculation of each pairwise potential and the summation over all pairs, which is performed within each thread block on the GPU and for the grid blocks on the CPU.

Table 7.2: Kernel FLOP Performance
Our CPU implementation performs at a consistent 95-96 MFLOPS due to the indirection of the lookups in the algorithm, whereas, the GPU performs increasingly well as the problem size grows with performance averaging 160 GFLOPS.

Residues	**CPU GFLOPS**	**GPU GFLOPS**	**Speed Up**
100	0.095	2.74	28.8×
402	0.095	42.8	450.5×
568	0.095	72.0	757.9×
1136	0.095	113.3	1192.6×
1704	0.096	124.8	1300.0×
2272	0.096	124.8	1300.0×
3408	0.096	147.2	1533.3×
4544	0.096	160.4	1670.8×
5680	0.096	148.0	1541.7×
6816	0.096	166.9	1738.5×
7668	0.096	162.4	1691.7×

The apparent lack of performance from the CPU implementation is comparable to simulations implemented in CHARMM for Cc/CcP and UIM/Ub, implying that the memory-bound nature of contact potential look-ups is what slows it down when compared to an n-body simulation. It is important to distinguish the difference between a pipeline friendly gravitational n-body method and the Kim and Hummer model. Firstly, the memory access characteristics of gravitational or electrostatic problems differ in that these simulations require less than half the data per body in comparison to ours and, secondly, require no additional data unique to the current pair being evaluated. Finally, we require a reduction operation to be performed on the force/charge experienced by each body as opposed to the independent streaming nature of gravitation or electrostatic forces acting upon a body. Consequently, the data independent model in gravitational case results in a method that maps well to the instruction pipelines on both a CPU and GPU, as opposed to our data model which contains conditional random memory access which limits our simulation to the cache and memory access performance of the processor core. Thus, the lack of ability to pre-fetch the random access variable on the CPU and the inability to latency hide this operation result in constant performance of 0.095 GFLOPS on our Intel Core 2 Duo.

To put both CPU and GPU FLOP results in context, the indirection in the lookup table is

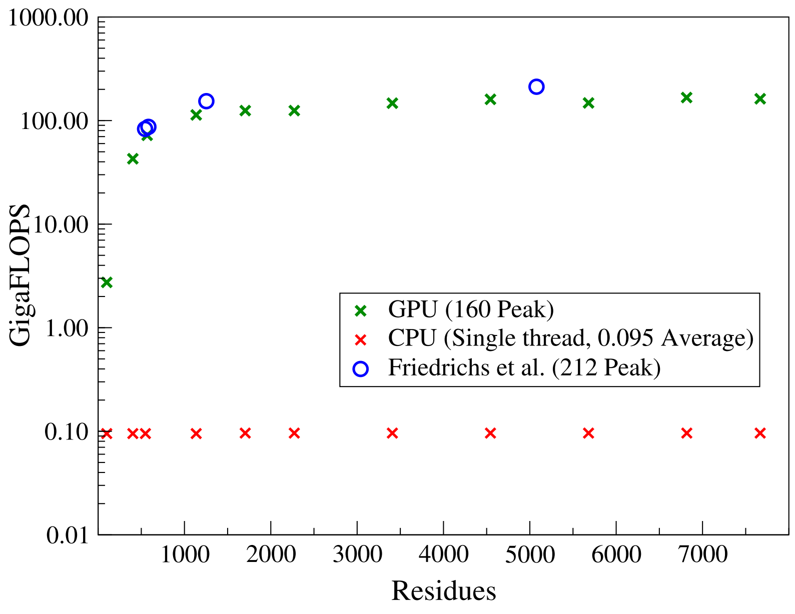


Figure 7.10: Kernel FLOP Performance
CPU performance is limited to approximately 0.1 GFLOPS for any problem size, indicating that problem size does not adversely affect the performance of the CPU. The GPU/CPU hybrid algorithm sustains computation of the interaction potential at 147 to 167 GFLOPS for larger problems. This indicates favourable scalability on the GPU since larger problem sizes, of biological significance, do not cause a drop in GPU performance.

a major stumbling block on a scalar processor such as a CPU. Modern CPUs have long instruction pipelines and achieve high FLOP rates by efficiently scheduling instructions such that these pipelines are always full. The indirection, determining the position of the data as opposed to the data itself, i.e. the index of the correct contact potential based upon the current residues, results in the instruction pipeline having to wait while that value is loaded, severely hampering performance. In N-body gravitation or electrostatics simulations where the model determines the interactions of bodies based solely on their type, as opposed to aggregate behaviour through course graining, this is not a performance issue. Furthermore, branching within the pairwise potential and checking whether or not the pairwise interaction potential calculation must be performed (e.g. residues must be from different molecules) also limits ultimate performance.

In comparison, other GPU electrostatics implementations, such as Friedrichs et al., achieve 0.29 GFLOPS (calculated using quoted values of 212 GFLOPS on a GPU and 735 times speed-up) on a CPU for their MD kernel [17] and Stone et al. achieve 0.28 GFLOPS for a highly optimised direct electrostatic summation kernel with GCC and SSE extensions [18]. The 3D vector operations of MD make it more amenable to SSE acceleration than our kernel, as many component-wise calculations can be performed independently, whereas, we can only calculate our distance squared (a dot product) in this way. Achieving FLOPS performance within 30% of these SSE implementations shows that our kernel performance can be considered a generic performance estimate for high level C++ algorithmic optimisations without exploiting the SSE's four way SIMD.

Unlike the CPU, CUDA warps can hide the latency of the indirection, far out-performing the CPU implementation and achieving values in the expected performance range. Figure 7.10 shows our CPU and GPU kernel FLOPS performance and that of Friedrichs et al. Both simulations use the same forms of non-bonded forces and GPU, providing a comparable performance figure, even though it is for MD as opposed to interaction potential summation. Friedrichs et al. also point out that the FLOPS performance of the CPU is an underestimate since the fast GPU functions such as sqrt and exp require in the order of 15 and 20 CPU floating point operations as opposed to one on the GPU, creating a large difference between effective and actual FLOP count of the kernel.

With regard to scaling, GPU performance for small problem sizes is modest, performing only 2.7 GFLOPS in the case of a 100 residue problem, but scales quickly to performing up to approximately 160 GFLOPS for larger problems, as listed in Table 7.2. The performance figures for the GPU-enhanced evaluation include a final reduction of the partial sums on the CPU, meaning that the FLOP rate on the GPU is actually higher than the values in Table 7.2. These figures also assume that the conditional branch of the algorithm is executed every time, when in reality it will only execute for a subset of residues in contact at the surface of a molecule. This will degrade performance marginally as it results in one additional store, FMA and subtract, when compared to the other side of the branch.

Finally, the effects of kernel optimisation can be determined by profiling the GPU using a set kernel configuration. Figure 7.11 shows that the even with the use of the GPU for interaction potential calculations, this function still accounts for over 92% of runtime for the smallest simulation. Interestingly, the time required for kernels in the three smallest problems is similar, implying that the time required for mutations and saves is long enough to make the proportion of kernel runtime decrease to as low as 90%. However, once the critical 1000 residue point is passed, interaction potential dominates runtime once more, increasing to 99.75% of simulation runtime for 6816 residues.

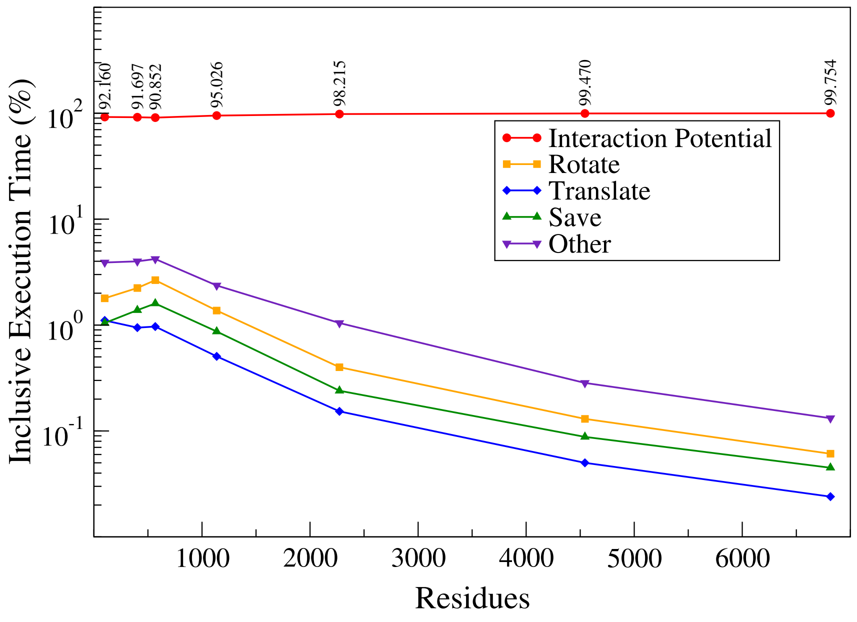


Figure 7.11: GPU Inclusive Profile
Re-profiling of the benchmarks with a GPU accelerated kernel sees the proportion of runtime devoted to the kernel decrease from 98% to 92% for the smallest benchmark, resulting in an overall decrease in runtime of 8.9 times from 18 seconds to 2 seconds. Benchmarks show that even with GPU acceleration, the proportion of runtime required for interaction potential calculations makes optimisation of other parts of the simulation unnecessary.

7.3 Summary

Profiling reveals that the interaction potential kernel accounts for over 98% of runtime, even for simulations of 100 residues. Theoretically, a perfectly efficient 100-way parallel implementation would allow this to be sped up by a factor of 34. Larger simulations should experience more speed-up due to larger problem sizes.

We find that the underlying performance of our implementation is limited by the algorithmic implementation on the GPU, not the choice of memory for the random lookup table. We note that thread block sizes of 32, 64, 128 or 256 threads perform similarly for constant, global and texture memory lookups compared to the CPU. As expected, shared memory, due to its scarcity, does not provide a viable solution for random lookups, and when used for this and as a cache for residues, kernels suffer from severe resource starvation and low occupancy. Even so, shared memory affords up to 2 orders of magnitude improvement over the CPU kernel.

It is possible, given such parameters to auto-tune a simulation at runtime. In terms of a parameter search space, the block size is dynamic, requiring only one kernel implementation. However, the fundamental differences in the use of shared and texture memory usage would necessitate kernel methods, generated at compile time, and selected by the host at runtime. Such a system would allow for the host to perform a search of the parameters in the manner performed for our benchmarking to select an optimal kernel configuration. The configuration may not be the optimum solution, but careful inspection during the algorithm design and implementation will narrow the parameter set such that an exhaustive search of the reduced candidate parameter space will result in an acceptable simulation parameter set with minimal time cost.

Kernel execution can be considered the smallest work unit on the GPU. As such, simulation time and overall speed-up are limited by the time required to execute the kernel. Additional methods such as multi-threading and asynchronous methods do not affect the kernel execution time, as these methods overlap computation with memory operations. They will improve the time required to calculate the interaction potential, since part of this is operation is performed on the CPU, but, the GPU performs the bulk the work. The following chapter discusses the overall performance of simulations using both sequential CPU-GPU use and multi-threaded CPU-GPU use and finally, asynchronous GPU use.

Subsequent benchmarks assume a kernel using texture memory for lookups and the use of shared memory for residue caches. Block size is selected based on the simulation size, choosing 32 threads per block for simulations of less than 1000 residues and 64 threads per block for larger simulations.

Chapter 8

Simulation Benchmarking

The effects of high level optimisation and parametrisation are measured in this chapter, providing ultimate speed-up figures for simulations with consideration of the use of multi-threading, GPU sharing and asynchronous computation on the GPU. With a set of predetermined kernel parameters, the effects of GPU acceleration on the runtime of a simulation can be determined. As stated in the previous chapter, the interaction potential calculation can be viewed as an atomic unit of work in the context of the overall simulation.

The use of texture memory for lookups, shared memory for caching residue data and a block size of either 64, for simulations of greater than 1000 residues, or 32 for simulations of 1000 residues or less is fixed for the following benchmarks.

The multiple levels of parallelism in the REMC problem are now addressed at the high level. The runtime of the simulation,

$$T = T_{serial} + \frac{T_{parallel}}{P}$$

is determined by the number of parallel Monte-Carlo simulations, P. The Monte-Carlo simulations can run in parallel without any communication, other than synchronisation for performing replica exchange. Furthermore, the ratio of Monte-Carlo simulation to replica exchange causes execution time, T, to tend to $\frac{T_{parallel}}{P}$ resulting in a theoretical speed-up of P for the Monte-Carlo simulations, provided that there are sufficient non-blocking processes, P.

For the CPU, this is the case using a dual core processor, i.e. a $2\times$ speed-up. Simulations of 20000 Monte-Carlo steps are performed, 1000 steps per replica, using one, two and four threads of execution, performing the Monte-Carlo simulations in different replicas concurrently. The results from these benchmarks are plotted in Figure 8.1. The serial (single threaded) simulations perform approximately 1000 Monte-Carlo iterations per second for the smaller benchmarks with run-times increasing quadratically with benchmark size, slowing the number of Monte-Carlo iterations per second to approximately 0.07 for a simulation of 7668 residues.

The use of POSIX threads to perform concurrent Monte-Carlo simulations results in a speed-up proportional to the number of CPU cores in our test machine. Figure 8.1 illustrates, for a dual core machine, that the performance of 4 threads is nearly identical to that of 2 threads. Four threads scales with approximately 97% to 98% efficiency on two processors for all but the 100 residue benchmark, where it is 92% efficient. When only two threads are used this result is largely unchanged, with larger simulations scaling with 1% less efficiency than in the case of 4 threads. However, performance dips for the 402 and 584 residue benchmarks scaling with 95% and 92% efficiency, respectively.

This result shows the CPU implementation to be limited by the hardware parallelism of the CPU. Due to its compute bound nature, it is likely that similar performance will be observed for four, six and eight cores, with decreases in the efficiency of the scalability due to sharing the architecture connecting the CPU to the rest of the system. This assumption also relies on there being sufficient replicas in the simulation to occupy all CPU cores.

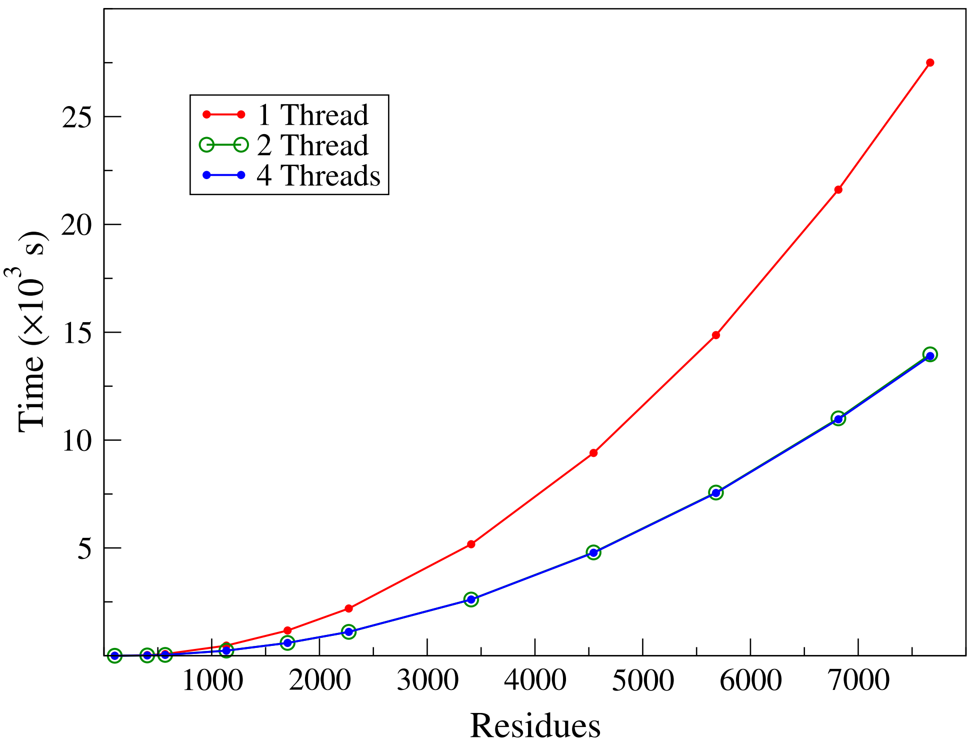


Figure 8.1: CPU Simulation Time (20000 MC Steps)
Timing the reference simulations on a single core shows that, for small simulations, approximately 1000 Monte-Carlo steps are performed every second, this rate slows as the problem size grows, dropping to a mere 0.07 steps per second for 7668 residues (approximately one every 14 seconds). Multi-threading the simulations, by performing the Monte-Carlo loops in parallel on the CPU, speeds up the simulation, but is limited by the number of cores available. Thus, a speed-up of 2 is recorded for a multi-threaded CPU implementation. Increasing the number of threads to 4 (2 per CPU core), marginally improves run-time. Because of the compute bound nature of the interaction potential, the use of multiple threads per core does not benefit significantly from overlapping memory operations with computation.

To put these figures in perspective, it is possible to perform 1.9×10^8 Monte-Carlo steps per day when simulating the interactions of UIM/Ub and 1.1×10^7 steps per day for Cc/CcP simulations using 4 threads and two cores. Kim and Hummer used CHARMM for their simulations, which only performs 9×10^6 and 8×10^5 steps per day respectively, notably slower than our CPU implementation for these simulations.

Table 8.1: CPU Implementation vs. CHARMM Performance
For the reference simulations, UIM/Ub and Cc/CcP, our CPU implementation outperforms CHARMM by an order of magnitude when measured in Monte-Carlo iterations per day.

Complex	Residues	CHARMM	CPU Implementation	Speed-up
UIM/Ub	100	9×10^6	1.9×10^8	21.1×
Cc/CcP	402	8×10^5	1.1×10^7	13.7×

Attaining a baseline benchmark with the CPU allows us to evaluate our GPU implementation. All GPU benchmarks are relative to the serial figures presented in Figure 8.1. For simplicity, speed-up is quoted relative to a sequential CPU simulation on the Intel 3 GHz E8400. This means that speed-up figures can be halved or quartered for 2 or 4 CPU cores, respectively. Additionally, for these and the following figures, initialisation time is ignored as it contributes negligibly to overall simulation runtime. Simulation time is measured from the beginning of the first Monte-Carlo simulation until the all threads exit after the final Monte-Carlo step.

8.1 GPU Benchmarking

For GPU benchmarking, we initially use a serial CPU solution with synchronous GPU acceleration of the interaction potential calculations. This model is extended to a multi-threaded benchmark, with Monte-Carlo simulations divided among CPU threads. Each thread initialises its own CUDA runtime and shares the GPU with other threads. However, nVIDIA advises against such an approach because of the overhead in managing run-times [22]. Finally, we use asynchronous CUDA streams to overlap computation on the CPU and GPU to make best use of system resources yielding our best results.

For synchronous GPU execution each replica's Monte-Carlo simulation is performed in its entirety before the next replica performs its Monte-Carlo loop. This mean that only one CUDA runtime context is used and the only extra overhead is the copying of residue data between the CPU and GPU.

For our GPU benchmarks, execution time ranges from 2.2 seconds up to 455 seconds, in stark contrast to the CPU which for the same benchmarks, requires from 17 seconds to 275000 seconds.

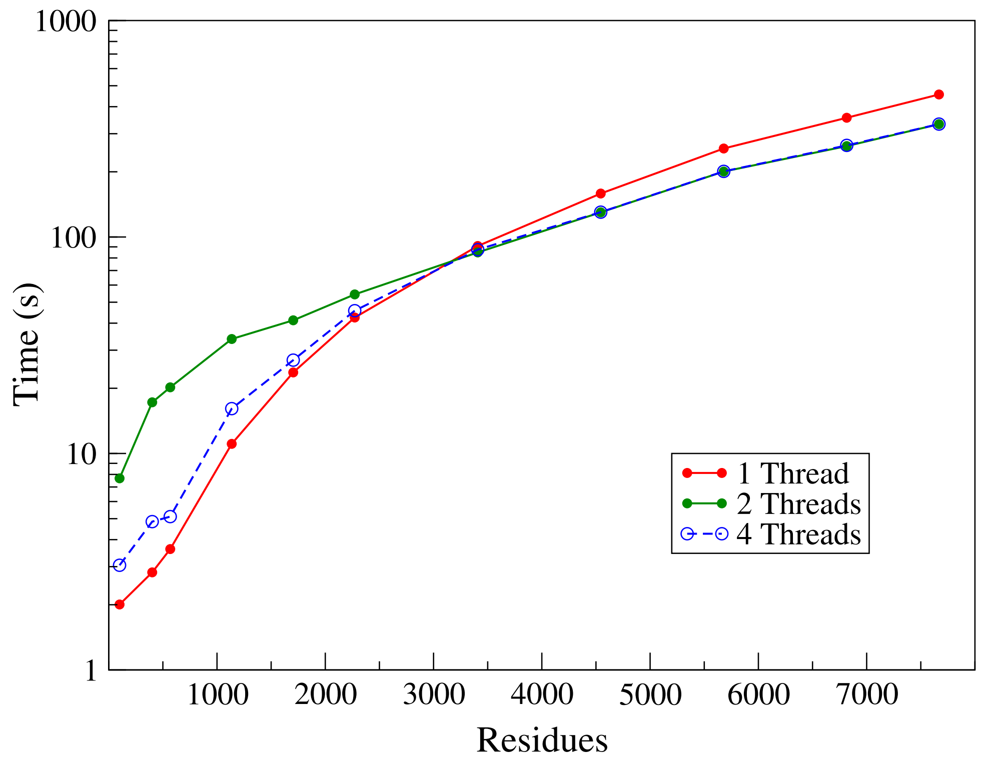


Figure 8.2: GPU Simulation Performance
When the GPU is used exclusively by a single POSIX thread smaller benchmarks, up to approximately 3000 residues, perform best before the performance of multiple threads and increased duration of interaction potential kernels outweighs the cost of CUDA context switches. The performance of both 2 and 4 CPU threads sharing the GPU is equivalent once context switches are negated by the need to parallelism. However, for smaller problem sizes, 4 threads outperform 2 threads because of they can hide CUDA context switches behind host computation more effectively than 2 threads.

GPU benchmarks are performed using 1, 2 and 4 threads on the host, sharing a single GPU. Figure 8.2 shows that the serial case of 1 host thread performs best for up to approximately 3000 residues, but for larger simulations, 2 and 4 thread configurations outperform a single thread.

For simulations of less than 3000 residues, the overhead of switching CUDA contexts and the resultant blocking effect on the CPU threads is clearly evident from the relative performance of 1, 2 and 4 thread benchmarks. One thread of execution requires no context switching, seeing control pass from CPU to GPU for computation and memory operations. For two threads, the introduction of a context switch and the blocking effect of GPU calls from each thread on the other causes a slow down for smaller simulations. But, interestingly, use of two additional threads and the resultant CPU-GPU computational overlap, results in much better performance than merely two threads. The behaviour of these benchmarks clearly exhibits the expected behaviour that nVIDIA advertises [22], limiting the programmer to the recommended single CPU thread and CUDA context per GPU.

However, in contrast to the CUDA guidelines, independent Monte-Carlo simulations with the GPU only used for the interaction potential calculations achieve a break even point when the runtime of each kernel invocation negates the cost of context switching due to the computational overlap between host and device. Here, GPU sharing between two or four threads is of similar cost, as the CPU is always waiting for the GPU. Thus, the final reduction step on the CPU is fully hidden behind kernel invocations and memory copies from other contexts. Once the CUDA kernel execution time far outweighs the context switch time, the number of threads sharing the GPU becomes less important because the GPU becomes the scaling bottleneck. The difference between 1 CPU and multiple CPU threads sharing the GPU lies in time wasted by having an idle GPU. The GPU/CPU combination is also limited by memory transfers between the CPU and GPU since they are blocking and prevent the CPU from working ahead on Monte-Carlo mutations while the GPU calculates the interaction potential, causing that thread to be idle until its associated runtime completes its instructions.

On hardware such as Tesla C870 cards using the original G80 [1], asynchronous GPU functionality does not exist, meaning that the performance achieved using synchronous GPU operations are optimal, and our results therefore scale according to the resources available on a G80. Hardware of CUDA 1.1 compute capability supports asynchronous GPU usage and can benefit from streaming. We employ this technology as our final strategy in implementing our algorithm.

8.1.1 Asynchronous Performance

The objective of asynchronous GPU computing is to maximise the utilisation of computational resources on both the CPU and GPU, as well as memory transfers. The host is able to transfer page locked memory between RAM and the GPU while both the GPU and CPU perform concurrent compute tasks. A fortunate feature of our algorithm is that the Markov-chains within the Monte-Carlo simulations are independent, comprising $O(N)$ CPU operations and $O(N^2)$ GPU operations. Furthermore, having 20 replicas means that all 20 can be at different instructions of the Markov-chain such that some can be performing memory transfers between CPU and GPU, some can be performing mutations on the CPU, and others can be calculating the interaction potential on the GPU, making the algorithm highly suited to asynchronous GPU streams.

The modification to the algorithm is simple. Instead of performing all the steps in a Monte-Carlo iteration sequentially for one replica, followed by all the steps of the next in the same manner, the first step of every replica is performed, followed by the second step of every replica, continuing until all steps in an iteration are performed. While in a CPU only usage model this makes no difference to performance because all processing occurs on the same device, in GPU programming it enables processing overlap between GPU and CPU.

[1]Cambridge University's chemistry cluster zero.ch.cam.ac.uk uses Tesla C870 GPUs.

The sequential usage case requires the GPU to return control to the host before the simulation can continue, deviating from this behaviour would violate the rules of the Monte-Carlo simulation. Thus, either the GPU or the CPU is idle while the other is processing. To avoid this, the GPU and CPU can perform concurrent computation by interleaving blocking sections of the Monte-Carlo simulations.

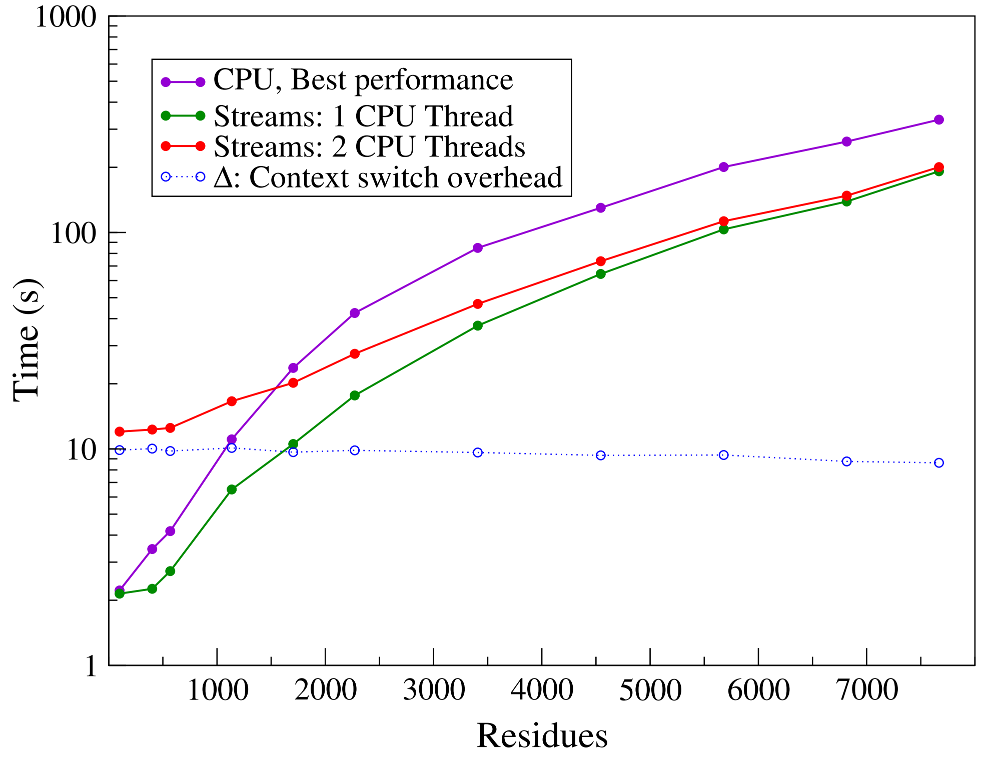


Figure 8.3: GPU Stream Performance
The use of the asynchronous CUDA API vastly improves runtime for the performance benchmarks. Instead of using POSIX threads to implicitly overlap computation on the host and device, streams explicitly manage this within a single CUDA context. Results are the average of 1,2,4,5 and 10 streams per POSIX threads which for a particular thread, result in practically identical performance. The context switch is most noticeable for smaller problem sizes due to the relative proportion of runtime devoted to context management. The cost of using more than one context is relatively constant for all benchmarks, ranging from 8 to 10 seconds across the entire range of residues, illustrated here by Δ, the blue dashed line.

Plotted in Figure 8.3, we observe that streams outperform all other configurations of kernel due to the efficiency of concurrent CPU and GPU computation. Additionally, CUDA kernels are guaranteed to be scheduled continuously on the GPU, thus maximising its utilisation. The performance of 1, 2, 4, 5 or 10 streams is almost identical, meaning that the GPU's hardware performance is the limiting factor in the simulation. If the CPU and context switching were a bottleneck, we would observe a significant increase in performance as more streams were used. In fact, more streams are not significantly more efficient. Multiple streams are on average 0.4% faster than a single stream and smaller simulations tend to be slower by approximately 1%. The optimal number of streams, according to our benchmarks is 5 per thread, achieving the best benchmark scores for every simulation size. This performance variation is indiscernible when plotted in Figure 8.3.

When streams and threads are mixed, the cost of context switching becomes critical. Unlike sequential GPU operations, the performance gained by streams is reliant on the GPU runtime context remaining active. The difference in performance between 1 and 2 runtime contexts is evident in Figure 8.3. Again, as the size of the simulation increases, the difference between the performance of one thread with no context switching and 2 threads with context switches becomes less significant. The penalty of context switches is clearly measurable (Δ in fig. 8.3), with benchmarks of 2 threads requiring from 8.6 to 10.1 seconds longer than a single thread across all benchmarks. As is the case with synchronous GPU use, the difference decreases as the simulation size increases, but critically, multi-threaded streams cannot outperform a single thread of streams for a single GPU.

The cost of the overhead associated with context switching is also evident in Figure 8.4 which compares the runtime of both single threaded and multi-threaded stream use, showing that the multi-threaded asynchronous benchmark is slowest for benchmarks of fewer than 1000 residues, but unlike the number of pairwise interactions, the events causing the overhead remain constant for all benchmarks, resulting in the inefficiencies of such a benchmark becoming less critical for larger simulations.

We find that CUDA streams significantly improve our simulations, resulting in performance better than synchronous GPU operations in all but the smallest case. When the single threaded asynchronous case is compared to the best synchronous case, that is, single threaded for less than 3000 residues and 4 threads for greater than 3000 residues, the asynchronous simulation can be expected to perform between 1.25 and 2.6 times faster. This is illustrated in Figure 8.3 where the minimum runtime for the each synchronous CUDA benchmark is plotted. The recommended model of 1 CPU thread per GPU [22] proves best when all GPU benchmarks are compared to each other. As illustrated in Figure 8.4, the synchronous single threaded solution is fastest for the smallest benchmark, but is bettered in all other cases by its asynchronous counterpart, with the asynchronous benchmark tending to be 2.4 times faster once simulations are larger than 1704 residues. The improved performance of multiple CPU threads sharing the GPU over the single threaded GPU implementation is dwarfed by the benefits of using streams.

An important illustration of customised performance parameters are the three smallest benchmarks. A simulation in the order of 10 million Monte-Carlo steps per replica for as few as 100 residues would be delayed by up to 100000 seconds (27 hours) if multiple CPU threads with streams were used as opposed to a single CPU thread with or without streams, illustrating that a well chosen set of performance parameters will save much simulation time.

Importantly, the ability to share the GPU between threads is a valuable feature to have in

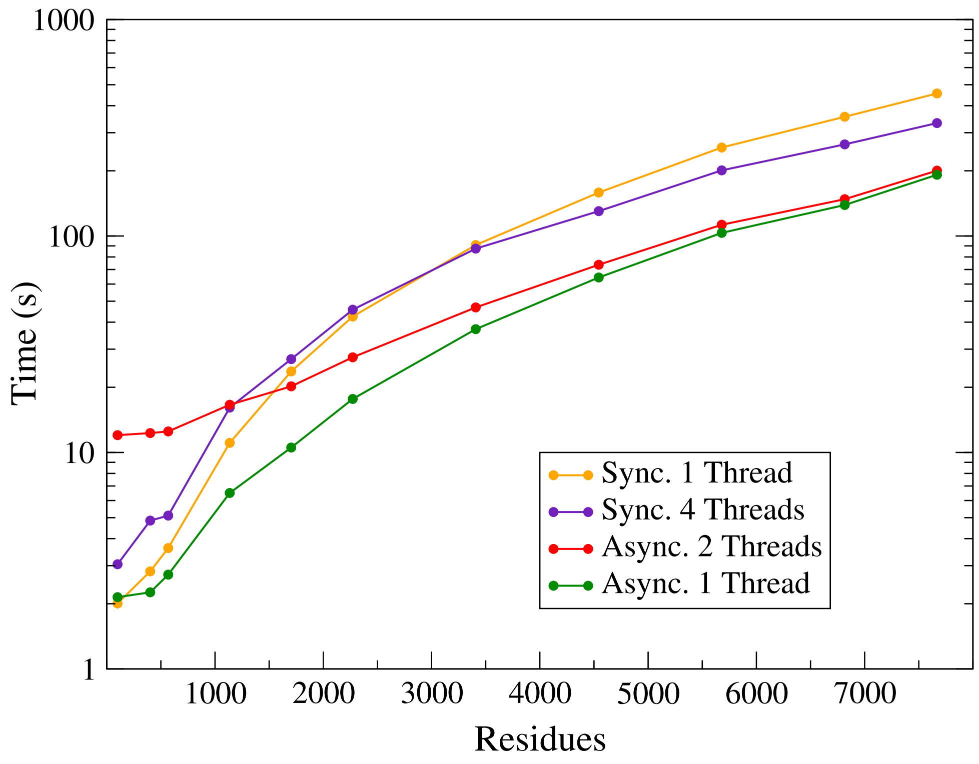


Figure 8.4: Synchronous vs Asynchronous GPU Performance
A comparison of asynchronous and synchronous CUDA API reveals the asynchronous single threaded benchmark to be best in all but the smallest benchmark where the sequential synchronous GPU benchmark is marginally better. The single threaded asynchronous benchmark, outperforms the synchronous CPU single and multi-threaded benchmarks by a factors of 2.4 and 2.6 respectively, peaking in the 2000 to 3000 residue range. For simulations larger than 1000 residues, asynchronous GPU use is generally twice as fast as synchronous GPU use.

cases where an application is able to use the GPU for independent processes. This case beyond computational chemistry to any field that benefits from GPU acceleration. The POSIX thread model is analogous to separate processes. In our case we can perform multiple simulations of replicas in different threads sharing a GPU, but practically, this is no different from independent simulations on the host which share the GPU, allowing for concurrent simulation on each core as opposed to waiting in a queue. Notably, this approach is beneficial in incrementally improving a model, where multiple simulations can run concurrently on a single GPU as opposed to the same simulations in series, in turn the time saved reduces the *time to discovery*.

8.2 Simulation Speed-up

When compared to the CPU benchmarks (figure 8.5), the GPU outperforms the CPU by at least 2 orders of magnitude in all but the smallest benchmarks. This result would hold true even if our CPU implementation were excessively tuned to extract the same performance from our CPU as Stone et al. [18] or Friedrichs et al. [17]. Additionally, the emergence of dual hexacore processors

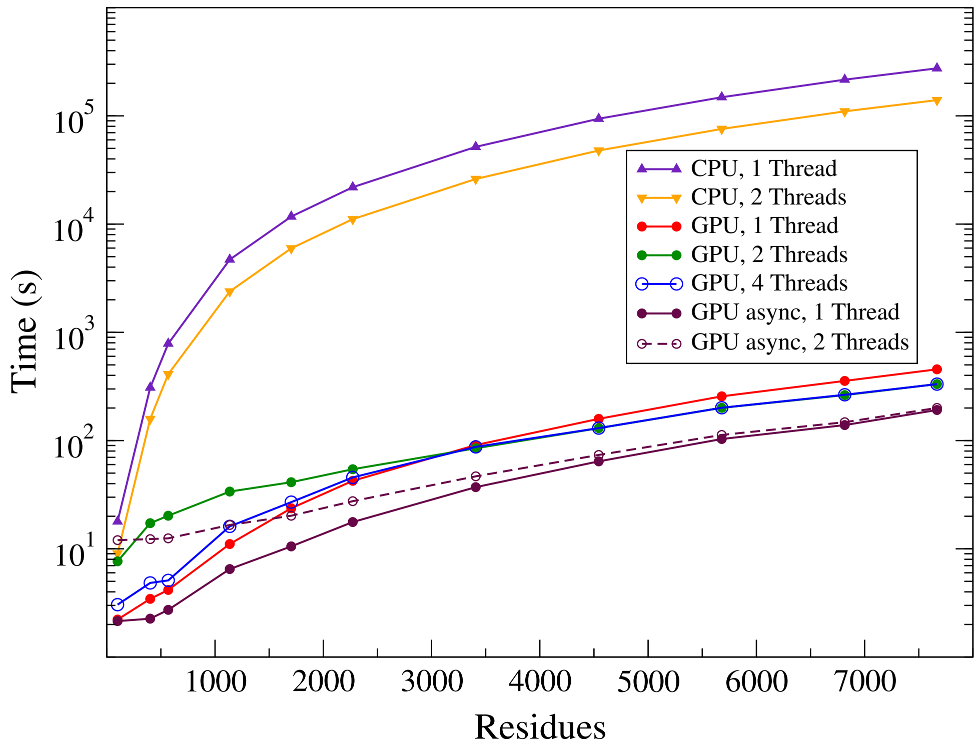


Figure 8.5: CPU vs. GPU Simulation Performance
The CPU scales efficiently, exhibiting a 1.98 times speed-up when both cores are utilised by either 2 or 4 threads. Once a GPU is used to accelerate the simulation, we observe large decreases in runtime for our benchmarks. These improvements may be minor, in the region of 17 seconds down to 2 seconds for 100 residues, or major: 275 000 seconds (76 hours) to 455, 332 or 191 seconds for the largest benchmark of 7668 residues, depending on the configuration. Effectively configured GPU benchmarks are 2 orders of magnitude faster than both single and multiple threaded CPU solutions at merely 568 residues. Practically, this equates to a speed-up over 100 for simulations involving the docking of 2 small proteins. The data for this figure is available in tabular form in Appendix C.

in supercomputing means that CPU hardware alone can increase our serial CPU benchmarks by up to a factor of 12 on a single compute node without any GPU acceleration. In fairness, the same approach may be taken when scaling up the number of GPUs in a node, maintaining the 2 to 3 orders of magnitude performance gap between the CPU and GPU implementations.

Overall the speed-up we observe using synchronous GPU operation, plotted in Figure 8.6, ranges from 8.9 times to over 600 times for a single GPU with a single CPU thread pairing. If the GPU is shared between CPU threads, speed-up increases to up to 828 times. Noteworthy data points are those of 402 and 568 residues, the second and third smallest simulations. These are simulations for docking 2 proteins, a common molecular modelling task and one for which we attain speed-ups of 109 and 217, respectively [2].

What is of critical importance is that the speed-up is dependent on the problem size. For

[2]A complete tabulation of all benchmark times and speed-ups is included in Appendix C.

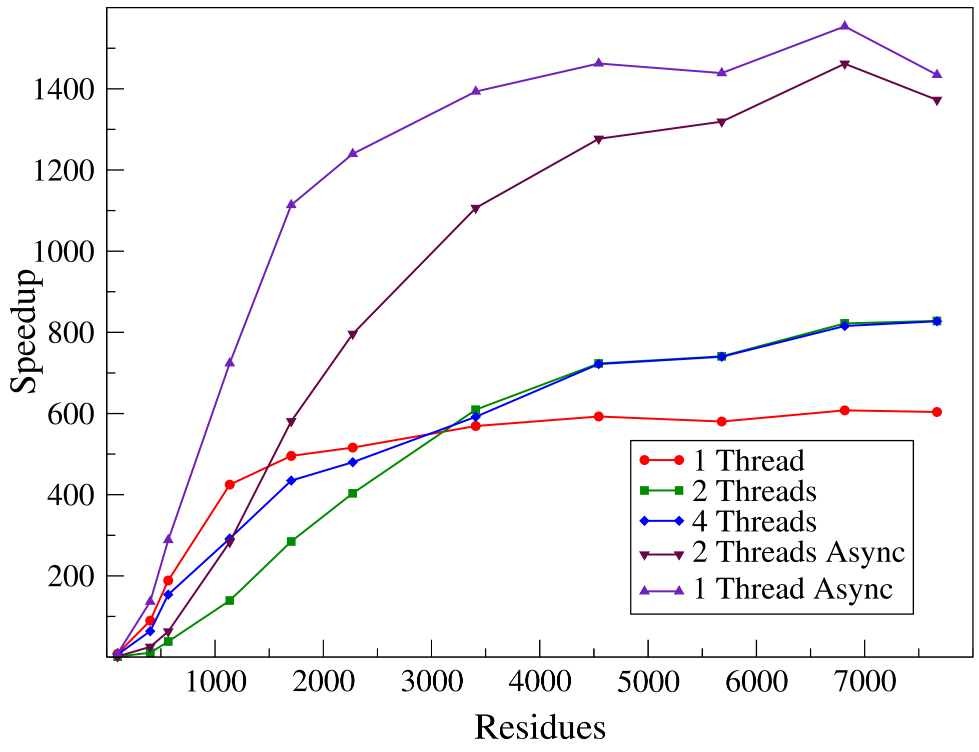


Figure 8.6: GPU vs. Serial CPU Speed-up
Speed-up peaks at an impressive 1553 times the serial solution for 6816 residues when using a single POSIX thread and one of 1, 2, 4, 5 or 10 CUDA streams. The 5 largest benchmarks all experience speed-up in the order of 1400 times or more for this configuration. Speed-up grows quickly with simulation size, starting at only 8.3 but immediately jumping to 137, 288 and then 724 before reaching the 1000 times speed-up mark for 1704 or more residues. The overhead in multiple threads using CUDA streams results in this configuration's speed-up figures being less than the former, but still in excess of 1000 once a simulation is large enough. Without asynchronous calls, both 2 and 4 threads achieve speed-ups tending to just over 800, whereas a single threaded synchronous GPU configuration will be limited to 600 times the performance of a single CPU thread. All figures exhibit a 'sawtooth' like trend due to the blocking and scheduling on the GPU, most noticeable for 5680 and 7668 residue benchmarks.

example, our 568 residue simulation is two halves of a viral capsid component. In our simulation the interactions between the 284 pieces in each, result in a speed-up of 217. But this is on top of the performance by the reduced representation of course graining which reduces the simulation from 4658 atoms.

Recalling Table 7.2, which indicates that the GPU can perform interaction potential calculations up to 1928 times faster than the serial CPU, a peak speed-up of 607 is less than optimal when you consider that it accounts for almost all of the runtime. Similarly, the multi-threaded case peaking at 828 times speed-up is less than expected.

In isolation, the performance of the blocking, sequential GPU benchmarks are better than expected. Initial estimates were on the order of less than 100, taking into account what both

GPU and CPU are theoretically capable of.

The introduction of asynchronous CUDA streams has a profound impact on speed-up, with the best case performing 1553 times faster than the serial CPU solution, an increase of 2.6 times that of the serial synchronous benchmark, and 1.9 times that of the multi-threaded synchronous benchmark.

The asynchronous figures match those of the interaction potential calculations far more closely, indicating that the efficiency of the streaming model is superior to that of the blocking synchronous model. The efficacy of the asynchronous configuration is an excellent illustration of how a heterogeneous CUDA/CPU algorithm should work. The only calculations performed on the GPU are the $O(N^2)$ pairwise electrostatics and reduction among threads in a block. All other CPU operations are $O(N)$ or, in the case of the final pairwise reduction, $[\frac{N}{blocksize}]^2$ floating point additions with an upper bound of 2^{16}. The CPU is able to perform this sum of 2^{16} 32-bit floating point elements in less than 500 microseconds. The CPU is able to perform all tasks of lower order complexity within the time that the GPU requires to execute the CUDA kernel, thus allowing complete overlap of computation on the GPU and CPU with only one stream. Peak performance occurs when the limiting factor in computation is the runtime of the interaction potential kernel on the GPU.

Particularly noticeable in Figure 8.6, is that the performance of kernels tends to fluctuate. In these benchmarks, this occurs most noticeably for 5680 and 7668 residues. The difference occurs within the CUDA kernel and is a function of number of residues. We observe the same performance trends for block sizes of 32, 64, 128 and 256. Speed-up for an entire benchmark merely amplifies the effect of minor differences between the behavioural trend and specific benchmarks. This effect is due to the runtime of each kernel being discretely divisible by the number of thread blocks on the GPU. For example, if there were 30 SMs on a GPU, a kernel executing 10, 15, 20, 25 or 30 blocks will run in similar time because none of the blocks compete for resources, however, as soon as 31 blocks are present, overall kernel runtime should double as 30 of the 30 blocks will be finished before the final block runs (discounting the latency in data transfers to each SM). Hence, runtime and speed-up figures on a GPU for a constant block size will have a “sawtooth“ pattern due to this scheduling peculiarity, absent from a CPU bound program.

8.3 Discussion

All of the benchmarks and analysis performed here applies to a single host and single GPU. However, due to the parallel nature of replica exchange Monte-Carlo and our decomposition scheme for multi-threading, this code scales almost linearly on 2 or 3 GPUs. Simulations in the following chapter have used used 3 GTX470 GPUs. Using 12 replicas, these simulations scale with 99% efficiency for 2 GPUS and 2 POSIX threads and 94% efficiency for 3 GPUs and 3 POSIX threads. The scaling effect further speeds up simulations over an above the performance increase due to the use of the GTX470 over the GTX280.

Due to nVIDIA's scalar SM array architecture, the increase in cores from 8 to 32 per SM, and the doubling of shared memory per SM means that larger thread blocks are no longer limited in the same manner as on the GT200. This results in an automatic occupancy increase in the case of 64 and 128 threads per block using shared memory tiles and the texture lookup for the contact potential.

The algorithm we implement scales across a cluster due to the embarrassingly parallel nature of the Monte-Carlo simulations. Multiple replica can be bound to a compute node with or without GPU acceleration. As discovered in the profiling process, the synchronisation of threads and the performing of replica exchange is trivial and fast, meaning that the communication overhead of using multiple nodes via MPI or a similar technology factor favourably into speed-ups.

The effect of the speed-up achieved using the GPU is most significant in its effect on wall time which translates into simulation runtime. Even for the smallest simulation of 100 residues, on a single CPU/GPU platform, we can perform 36 million Monte-Carlo iterations per hour. This equates to a one billion step simulation in approximately 28 hours. By comparison the CPU, with both cores fully occupied, would require over 5 days.

The second simulation, the binding of cytochrome *c* to cytochrome *c* perioxidase performs even better due to a four fold increase in problem size, resulting in a runtime reduction from 91 days on the CPU to merely 39 hours on the GPU.

The jump in performance between these simulations is massive in terms of time saved but highlights the inefficiency of the GPU for a small simulation since an improvement of 4 days is not particularly valuable. However, a reduction in simulation time of 98% for slightly larger simulations is critically important and allows for the iterative development of models and methods that would otherwise require runs of 3 months or longer between iterations.

Realistically, simulations such as these would be performed on high performance clusters and would not require 91 days of wall time. The speed-up from a GPU can be read as the relative increase in the amount of work possible. The quality of a Monte-Carlo simulation relies on the number of samples performed and from this perspective, a GPU can allow over 50 to 100 times more sampling in the same time for a 2 protein simulation in the same runtime. Furthermore, these results apply generally to GPU-enabled nodes, since each node with a similar GPU to ours should experience similar performance. The scalability imposed by the number of replicas is application dependent. If only a single simulation performed with multiple replicas, a large cluster of GPUs will not be utilised, but, as in the next chapter, many combinations of concentration and the number molecules are needed for a data set, meaning that simulations can be performed independently as a set of GPU-enabled tasks on a cluster, or as a sequential set of

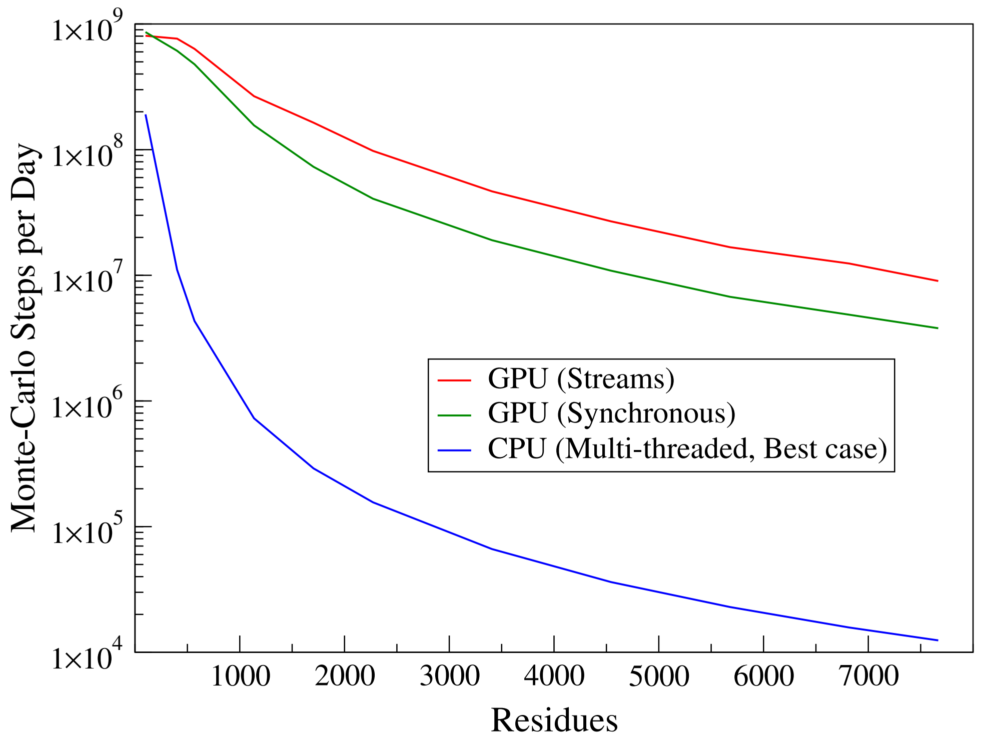


Figure 8.7: Monte-Carlo Steps per Day Performance
Simulation throughput from our GPU implementation is two orders of magnitude better than a CPU. These results illustrate that GPU acceleration of an inherently time consuming task such as docking can compress months of computation on a CPU into a matter of days on a single GPU. This means that either more samples can be performed in the same time, providing better quality data, or more simulations can be performed, providing more data..

batch jobs on a single GPU-enabled desktop.

Chapter 9

Applications

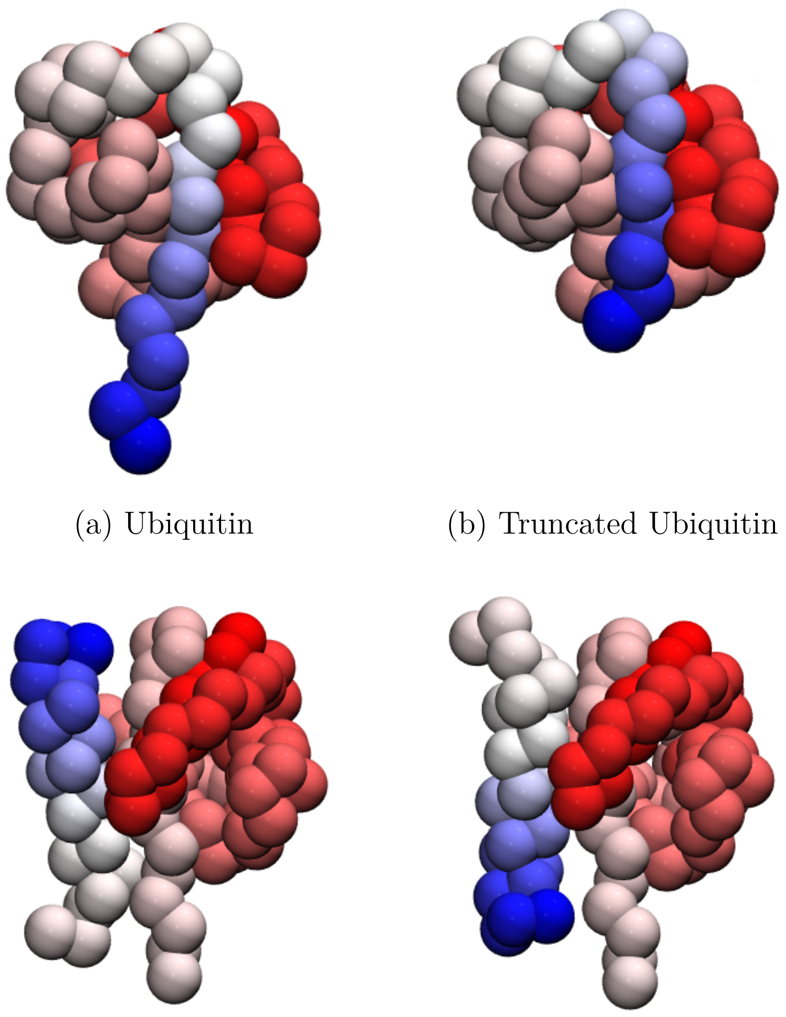
The benefit of possessing a fast docking code enables more tractable investigation into docking simulations. The utility of our application is demonstrated in this chapter. To begin, the validation stage of development reveals an interesting structural feature in the Ubiquitin/UIM simulations, highlighting the significance of the Ubiquitin tail. Thereafter, two applications of a fast docking code are discussed. We use our code to investigate the effect of macromolecular crowding by CspA, the cold shock protein of Escherichia coli, on binding of the cytochrome *c* to cytochrome *c* peroxidase. Finally, we investigate the assembly of viral capsid fragments using the Kim and Hummer model.

9.1 Ubiquitin C-Terminus Tail Truncation

Validation of the implementation resulted in an interesting observation in the Ubiquitin-UIM1 interaction. During the validation of the implementation, it was observed that the dissociation constant, K_d, was much lower than the expected value attained by Kim and Hummer [12]. Using the Miyazawa and Jernigan contact potentials [158] with equal solvent accessible surface area weightings (all residues have SASA equal to 1) [12], the expected dissociation constant is $1240\mu M$. However this value is determined by truncating the ubiquitin molecule to 72 residues (figure 9.1b), removing the final four residues from the C-terminal tail of the complete molecule (figure 9.1a).

Truncation has a significant effect on the dissociation constant. Simulation using the GPU at 100μM, 200μM, 400μM, 600μM, 800μM and 1000μM concentrations using both full length and truncated ubiquitin reveal that the truncated molecule results in weaker binding (higher K_d), with a dissociative constant value of $1574\mu M$. A similar result from Best of $1493\mu M$ using CHARMM confirms this finding.

Modelling the full ubiquitin molecule, even with a rigid tail, has a positive effect on the results of the simulation. Both our GPU implementation and Best report lower dissociative constants of 634μM and 595μM respectively. This result can be considered more favourable



(a) Ubiquitin (b) Truncated Ubiquitin

(c) UIM/Ubiquitin (d) Inverted UIM/Ub

Figure 9.1: UIM/Ub Structure and Docking Poses
(a) Ubiquitin consists of 76 residues, terminating in a flexible 4 residue C-terminus tail (blue). (b) Removing the C-terminus tail results in rigid 72 residue ubiquitin molecule, used by Kim and Hummer [180]. (c) The correctly orientated (experimentally observed) docking configuration of UIM/Ub. (d) The inverted docking configuration of UIM/Ub with the inverted UIM helix.

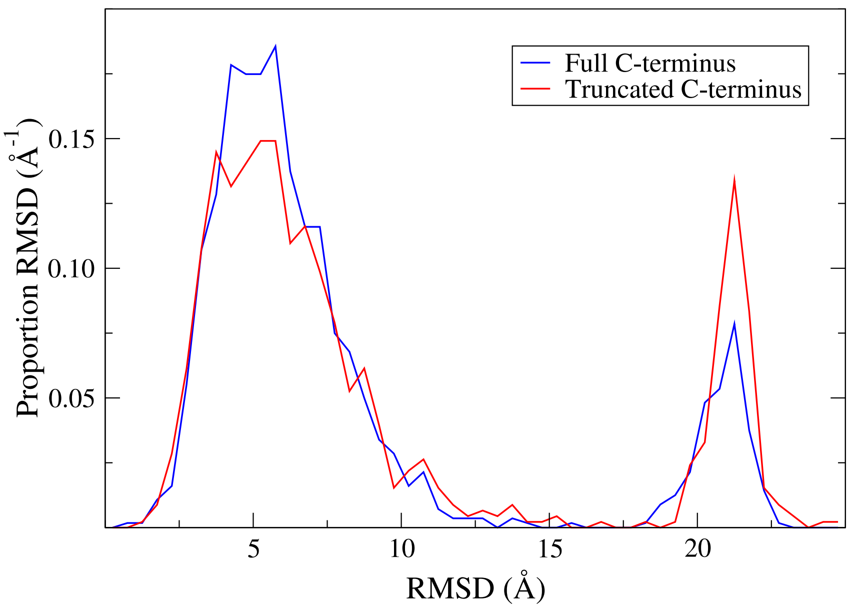


Figure 9.2: UIM/Ub Cluster Population Shift
Truncation of the ubiquitin tail causes greater occurrence of inverted UIM orientation (red) accounting for approximately 20% bound configurations discovered through simulation. Inclusion of the ubiquitin C-terminus tail results in a decrease in inverted UIM orientations (14%) and a greater propensity for the formation of the correctly bound complex.

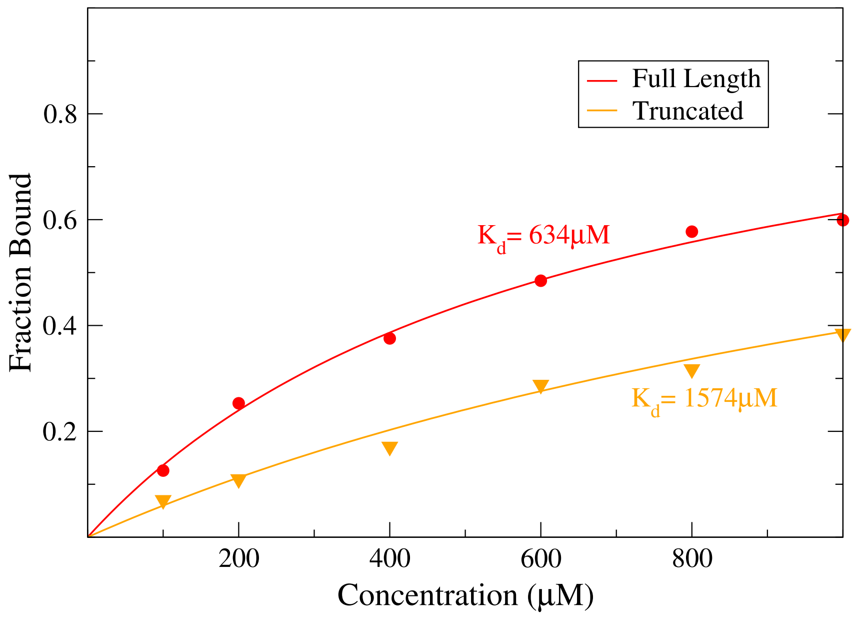


Figure 9.3: Ubiquitin Truncation: Effect on Binding Affinity
The presence of the ubiquitin C-terminus tail results in a decrease in the dissociation constant K_D for the formation of UIM/Ub. The difference in K_d for the simulations indicates that the presence of the tail results in stronger binding behaviour.

because it is much closer to the experimentally observed result of 280μM.

Most significantly, the inclusion of the ubiquitin tail changes the ratio of incorrect to correct poses discovered by the simulations. In the original study, the truncation resulted in a cluster of native-like structures accounting for 40% of the bound complexes discovered by simulation. This cluster featured two orientations of UIM1 helix, 80% of which were orientated correctly (Figure 9.1c) and 20% were inverted (Figure 9.1d). Inclusion of the tail results in a population shift from 4:1 to 6.2:1, indicating that the C-terminus tail interactions stabilise the correctly bound complex. A frequency plot of the RMSD of the bound complexes using the truncated vs full length simulations (figure 9.2) illustrates the population shift.

In the rigid body case, the presence of the tail enhances both the specificity and the strength of the binding. The strength of the binding increases, inferred from the decrease in K_d from $1574\mu M$ to $634\mu M$ (figure 9.3). The population of the correct binding orientation increases from 80% to 86% of the bound samples. Treating the flexible tail as a rigid structure is biologically unrealistic, future study of flexible and rigid tails and their effect on the correctly orientated binding state would prove interesting.

9.2 Cc/CcP Macromolecular Crowding

Macromolecular crowding is the phenomenon whereby the properties of molecules in a solution are altered due to the high concentration of macromolecules such as proteins are present in that solution [181]. Since molecules have evolved and function within intracellular environments that are crowded with other macromolecules, it is beneficial that simulations studying the interactions of such molecules be conducted in crowded environments as opposed to protein-protein interactions studied in uncrowded buffers [181].

Experimental studies have shown that crowding results in large quantitative affects on both the rate of reaction and the equilibrium of the system [181]. The interiors of cells are crowded places, as opposed to concentrated since there are no high concentrations of a specific molecule, but when considered as one, macromolecules typically occupy between 20% and 30% of the volume inside a cell. This volume is physically inaccessible to other molecules in the cell, effectively increasing the concentration of the reaction. However, these additional the intermolecular forces, e.g. steric effects, arising from the crowding effect generate energetic consequences that not generally considered in more conventional in vitro simulations [181].

Several theories accounting for the crowding effect in protein docking are based on the excluded volumes [182–184]. These theories are based on a simplified description of the protein molecules, such as a spheres. The value of such models will be greatly enhanced if experimental or simulation data fits the predictions [185].

Kim, Best and Mittal use scaled particle theory (SPT) to quantitatively predict the effects of macromolecular crowding [185]. The dissociation constant is the ratio of concentration of the unbound proteins to bound proteins,

$$K_d = \frac{[A][B]}{[AB]}$$

governing the the equilibrium of the reaction:

$$AB \rightleftharpoons A + B$$

where proteins A and B transition between the complex, AB, and unbound proteins, A and B.

Thus, K_d is related to the free energy change of the reaction, ΔG_{bind}, by the relation $\Delta G_{bind} = -RT \ln K_d$. ΔG_{bind} can also be expressed as the sum of chemical potentials: μ_A, μ_B, and μ_{AB}:

$$\Delta G_{bind} = \mu_A + \mu_B - \mu_{AB}$$

SPT calculates the excess potential, μ, for inserting a sphere, R_s (Cc/CcP, Cc or CcP) into a bath of hard spheres R_c (CspA):

$$\Delta\mu(\phi) = (3z + 3z^2 + z^3)\rho + (\frac{9}{2}z^2 + 3z^3)\rho^2 + 3z^3\rho^3 - \ln(1-\phi)$$

where ϕ is the packing fraction (the ratio of crowder volume to simulation volume), z is the ratio of radii from R_s over R_c and ρ is $\phi(1-\phi)$ [180, 185]. The difference in free energy as a result of crowding is calculated to be

$$\Delta\Delta G_{bind} = \Delta\mu_A(\phi) + \Delta\mu_B(\phi) - \Delta\mu_{AB}(\phi)$$

allowing the a new dissociation constant for a specific packing fraction, $K_d(\phi)$, to be calculated from the reference dissociation constant K_0 using

$$K_d(\phi) = K_0 \exp(-\Delta\Delta G_{bind})$$

Kim et al. performed crowding simulations of both UIM/Ub and Cc/CcP in the presence of spherical crowders, finding their results to agree closely with those predicted by SPT [185].

We extend this study by introducing a greater level of detail to each crowder because our implementation makes this level or detail tractable. We perform in docking simulations of cytochrome *c* and cytochrome *c* peroxidase and multiple CspA crowders. CspA is one of the most abundant small proteins in E. Coli, which is one reason it was chosen [180]. It is a suitable "crowder" because it doesn't interact strongly with cytochrome *c* and cytochrome *c* peroxidase. Our initial simulations show that the dissociation constants associated with Cc/CspA ($34080\mu M$), CcP/CspA ($21785\mu M$) and itself ($50264\mu M$) are two orders of magnitude higher than the reference dissociation constant of $755\mu M$ for the Cc/CcP complex (Figure 9.4) Thus, CspA can be used to study the the crowding effects of a hard repulsive sphere [185] or as a fully interactive protein [180].

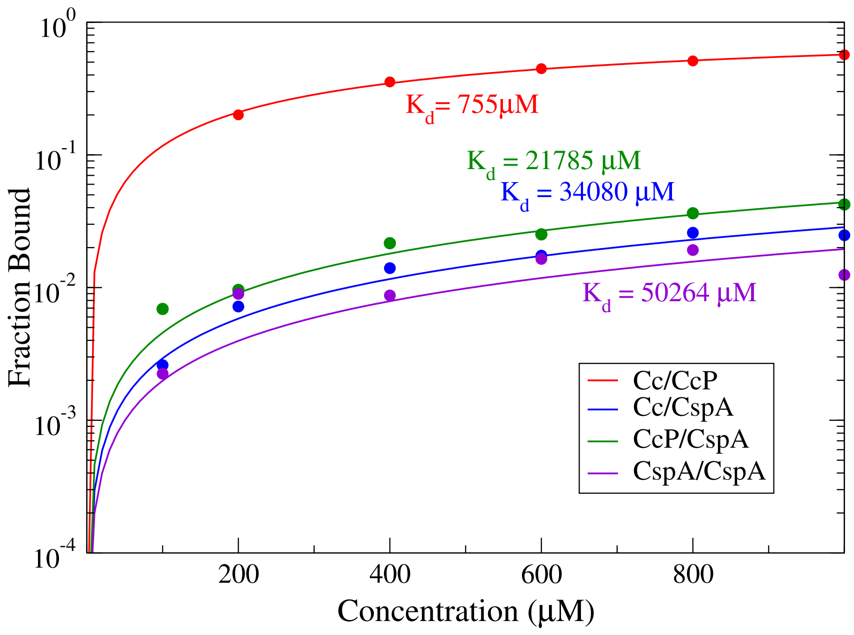


Figure 9.4: Cc/CcP and CspA: Isolated Binding Affinities
how each interacts in isolation with the other, CspA exhibits extremely weak binding with both Cc, CcP and itself.

9.2.1 Crowding Models

The effect of crowders on the free energy of the configurations is modelled in two ways. The simplest way, assuming that the crowders will effectively repel the other proteins in a simulation. This the interaction potential is composed of the pairwise Kim and Hummer potential (Chapter 3, Equation 4.1),

$$\varphi_{ij}(r) = u_{ij}(r) + u_{ij}^{el}(r)$$

for the interaction of the molecules of interest, namely, Cc and CcP in these simulations. All other pairwise contributions to the interaction potential are

$$V_{ij}(r) = (6/r)^{12}$$

modelling a weak repulsive potential between the crowders and all other molecules in the system in accordance with the model of Kim et al. [185]. $V(r)$ effectively models the crowders as hard spheres with r^{-12} providing a continuous short range repulsive force. Thus, crowder residues sharply repel other residues when their molecular surfaces overlap.

For performance, a cut-off distance of 15Å is used for repulsive crowder potential. Inclusion of the repulsive potential in the interaction potential kernel is achieved via a simple branch. Inclusion of an identifier in each residue's meta-data provides the means to discern between crowder and non-crowder residues and thus, select either the full or repulsive potential as required. Due to the ordering of residues in memory such an operation avoids the performance hit of divergent branches in warps other than those processing both crowder and non-crowder residues.

9.2.2 Simulations

Replica Exchange Monte-Carlo simulations are performed in the crowded environment using periodic boundary conditions. These simulations are configured at packing fractions (ϕ) of 0.05, 0.1, 0.2 and 0.3. That is, the volume of the crowders accounts for 5%, 10%, 20% or 30% of the volume of the periodic bounding cube which edge lengths determined by the number (n) and volume (V_c) of the crowders as $\sqrt[3]{\frac{nV_c}{\phi}}$. For each crowder is considered to have a volume of 12770Å^3, determined using Monte-Carlo integration with a probe of 20Å on the PDB files used for simulation. Similarly, SPT predictions of the dissociation constant use the volume of the CspA, Cc, CcP and Cc/CcP.

Simulations are initialised by distributing crowders on a cubic lattice within the bounding volume and inserting the Cc/CcP complex of interest in the centre of the volume in a near bound pose to effect a bound docking simulation with crowders. During each MC step, a protein is allowed to make both translations and rotations. Like Kim et al., these simulations are for 12 replicas from at temperatures of 300K to 600K, with $T_i/300$ forming a geometric progression. Similarly, translational moves range from 0.5Å to 5Å and rotations range from 0.1 to 0.5 rad. All proteins are subject to the same translational and rotational moves; the increased level of detail in each crowder requires explicit rotation. Simulations are allowed equilibrate for 5×10^6 steps before sampling is performed every 1000 steps.

9.2.3 Results

Simulations of 10, 15, 20 and 25 crowders were performed for packing fractions 0.1, 0.2 and 0.3. Due to the speed of our implementation, each of these 180 million Monte-Carlo iteration simulations, totalling over 2 billion interaction potential evaluations are performed in just over 6 days on a dual GTX470 GPU desktop.

Simulated fraction bound results for each simulation agrees with SPT predictions. Expected values, plotted as dotted lines in Figure 9.5, differ by a mean relative error of 2.8% from SPT predicted values. Dissociation constant calculations agree both quantitatively and qualitatively with those predicted by SPT (Table 9.1).

Energy vs DRMS plot of the simulations (Figure 9.6) illustrate the tendency of the Cc/CcP complex to favour the native binding site as opposed to simulations of only Cc/CcP. This effect can be illustrated by the manner in which complexes not contained or near the native binding site disappear from the DRMS/Energy plots. By taking the ratio of bound samples in the native cluster over the the total number of bound samples for a simulation at a specific packing frac-

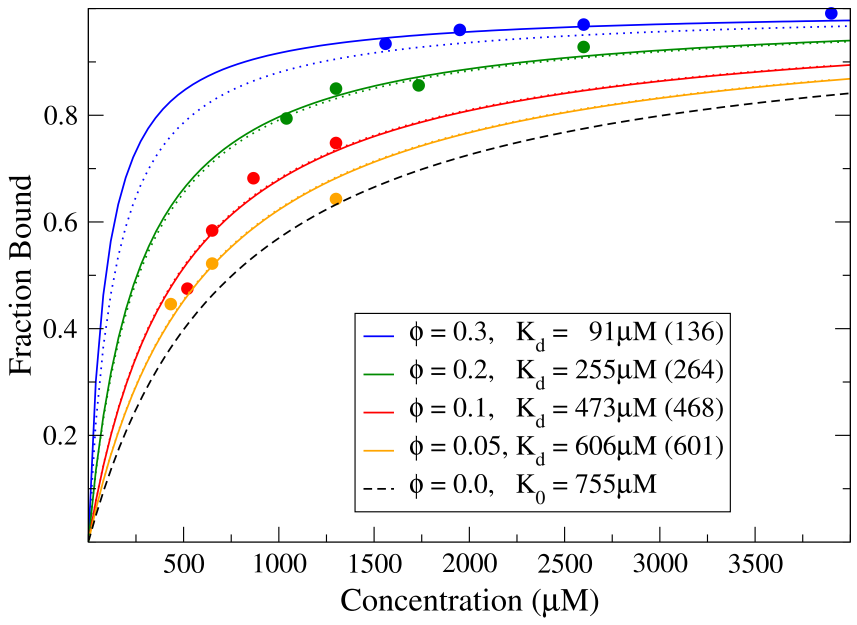


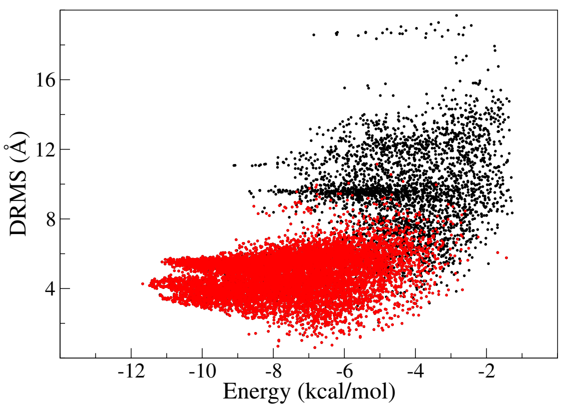
Figure 9.5: Crowded Cc/CcP: Fraction Bound
The introduction of crowders results to the Cc/CcP simulation results in increased binding affinity (solid lines) depending the packing fraction, ϕ. These results closely match the SPT predictions for each packing fraction (dotted lines). Simulated dissociation constants K_d are included in the legend, with expected values in brackets.

tion, the proportion of native to total bound complexes increases linearly with packing fraction (Table 9.2) from 56% for uncrowded simulations to 91% for the simulations performed at the highest packing fraction of 0.3.

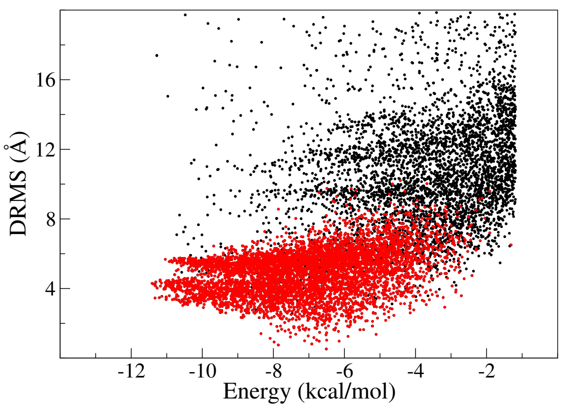
The native structure does not possess the lowest interaction potential for this model. Using the structure from the PDB entry 2PCC of experimentally observed cytochrome c cytochrome c peroxidase (Figure 9.7a), an interaction potential of -6.96 kcal/mol is calculated. The lowest energy structure discovered by simulation (Figure 9.7b) has an interaction potential of -11.7 kcal/mol and differs from the native structure by a DRMS of 4.27Å. Structurally, the poses differ by a rotation of approximately 30° only.

Table 9.1: Crowder Dissociation Constants
Simulated dissociation constant (K_d) values, agree strongly with SPT predictions.

Packing Fraction (ϕ)	**SPT Prediction (μM)**	**Simulation (μM)**
0.05	601	606
0.1	468	473
0.2	264	255
0.3	136	91



(a) Crowded Cc/CcP



(b) Cc/CcP

Figure 9.6: Crowded Cc/CcP: Energy vs. DRMS
Energy vs DRMS plots for crowded simulations (a) illustrate the tendency of the Cc/CcP complex to favour the native binding site as opposed to simulations containing only Cc/CcP (b). As evident from both (a) and (b) illustrate that samples with the lowest DRMS possess an interaction potential of approximately -6 kcal/mol to -8 kcal/mol in contrast with the DRMS of 4.3Å for the lowest energy complexes.

As evident from the DRMS and interaction potentials of the samples, accessible non-native structures with energies less than -7 kcal/mol with cause simulations to favour these configurations over the higher energy native configuration producing clusters in the correct binding site, but closer to the lower energy configuration (according to the model) than the native configuration. Thus, the predominant cluster has a representative DRMS of between 3.3Å and 5Å and an interaction potential of between -8.4 and -9.5 kcal/mol when performed under crowded conditions. In uncrowded conditions, configurations cluster similarly with DRMS and energy values of 4.5Å and -10.1 kcal.mol respectively.

Reduction of the clustering cut-off value (from 2Å to 1Å) results in an almost equal division of structures in the native binding region for all simulations, one with a representative DRMS of approximately 3.6Å containing the lowest energy structures and native-like structures. The

Table 9.2: Proportion Native Conformations
Clustering reveals that the proportion of native confirmations discovered by sampling increases almost linearly with the packing fraction, for packing fractions in the range 0 to 0.3.

Packing Fraction (ϕ)	**Proportion Native**
0.0	0.56
0.1	0.66
0.2	0.79
0.3	0.91

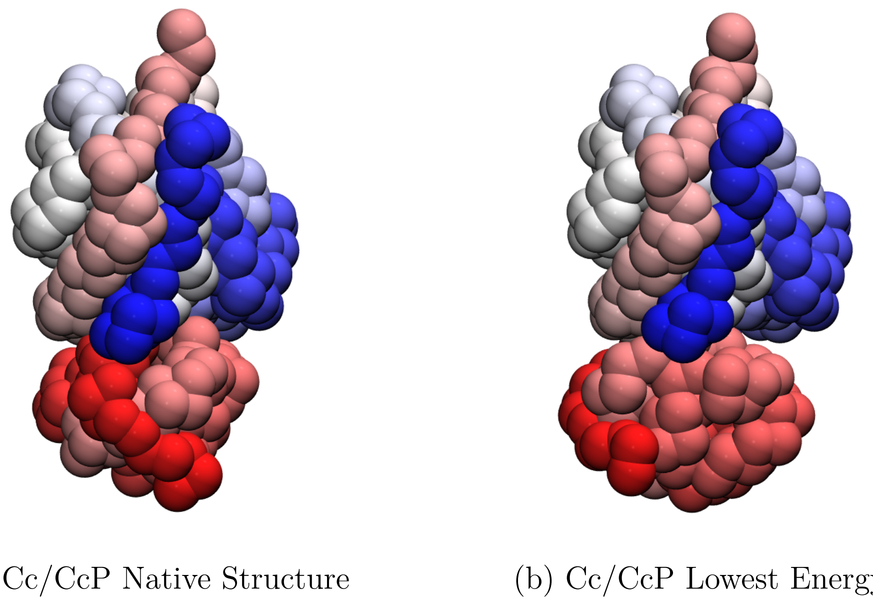


Figure 9.7: Cc/CcP Structures
(a) The native Cc/CcP structure, taken from the PDB (2PCC) has an interaction of -6.96 kcal/mol using the Kim and Hummer potential. (b) The lowest energy structure from our simulations has an interaction potential of -11.68 kcal/mol and differs from the native structure by a DRMS of 4.27Å. Consequently, configurations are attracted toward more energetically stable (lower potential) states during Monte-Carlo simulations.

non-native cluster contains a 180 degree rotation of the CcP protein with a representative DRMS of approximately 5.2Å.

To ensure that proteins do not aggregate within a simulation, that is, that crowder proteins form cohesive clusters within the periodic bounding box, the radial distribution function for each residue in the simulation space is calculated relative to the centroid of every other protein. The procedure for calculating the radial distribution functions, g(r), is to loop over all proteins, using the centroid of that proteins residues as $r = 0$. For increasing values of r, a count of all residues in the spherical shell from distance r to $r + dr$ is recorded for values of r from 1 to half the size of the bounding box. These values are normalised by the volume of each shell using a normalisation factor of $\frac{V}{N4\pi r^2 dr}$ to ensure a dimensionless g(r) that tends to 1 when no structure occurs within the distribution of residues.

Thus, a higher value of $g(r)$ will occur for protein interactions due to the increased density and structure at ranges of r associated with a docking site; $g(r)$ will be zero when residues do not occur within distance r of each other, indicative of repulsion and intersection; and $g(r)$ will tend to 1 at arbitrarily large r as no interaction occurs between proteins at this distance, implying no aggregation or structure [180].

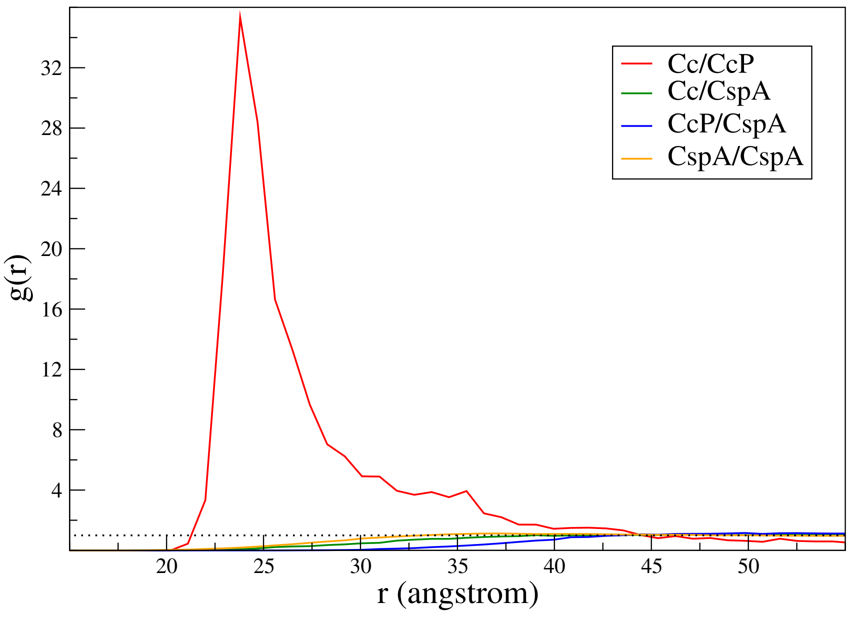
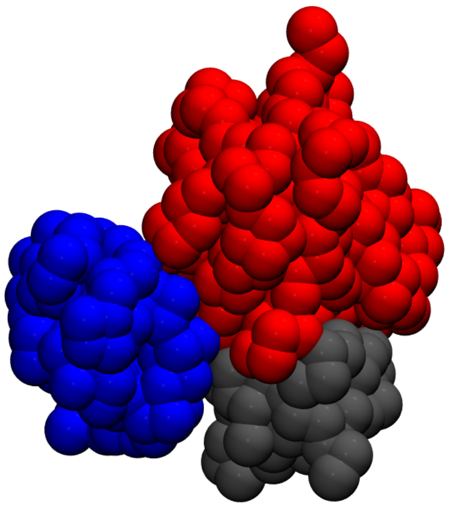


Figure 9.8: Radial Distribution of Cc/CcP/CspA
The radial distribution function g(r) shows that in both repulsive and fully interactive simulations, that the crowder CspA proteins avoid aggregative poses. The hard sphere repulsion of the CspA crowders results in no residues within 20Å of its centroid. Additionally, g(r) reveals no emergent structure between either of the proteins from cytochrome c/cytochrome c peroxidase and CspA. The convergence of radial distribution of CspA-CspA to 1 beyond self repulsive distances (20Å) shows that crowders lack structure due to aggregation.

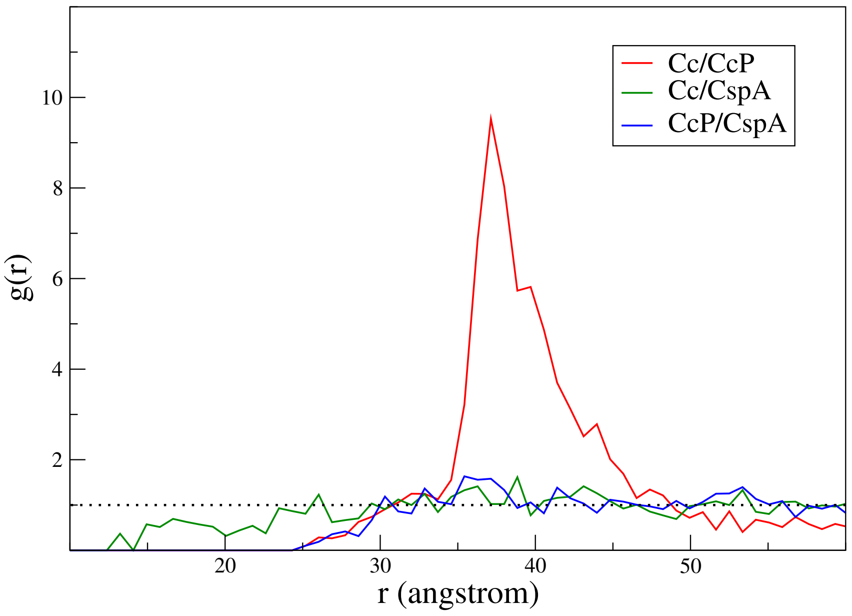
Crowder simulations performed exhibit such behaviour (Figure 9.8). The radial distribution function for Cc/CcP is largely unchanged from its uncrowded distribution, with a peak in $g(r)$ occurring between 25Å to 50Å due to the Cc/CcP binding site. The lack of structure and, consequently, aggregation of any CspA complex is as expected. Values of g(r) for Cc/CspA, CcP/CspA and CspA/CspA approach 1 for larger radii. Each complex registers a g(r) of zero for radii within its molecular boundary.

Future investigation into the effects of crowders will require the modelling of fully interactive crowders. Within a cell, molecules will not behave repulsively with potential functions mimicking $V(r)$. Therefore, fully interactive crowding is a future area of interest for this study.

An initial investigation into crowders with attractive potentials is performed by using a single CspA crowding molecule. Two simulations of Cc/Ccp and CspA are performed, one using repulsive potentials for the crowder and the other using the Kim and Hummer potentials for both crowder and complex. Both simulations are performed at a $755\mu M$ concentration containing 12 replicas at temperatures between 300K and 600K. The results of these simulations show an expected increase in binding affinity for the repulsive crowder, with K_d dropping from the reference $755\mu M$ to $670\mu M$. For the fully interactive CspA, K_d increases to $886\mu M$, implying that the presence of the single crowder interferes with the formation of the Cc/CcP complex.



(a) Structure



(b) Radial Distribution

Figure 9.9: CspA Interference
(a) Docking simulations containing fully interactive cytochrome c (red), cytochrome c peroxidase (blue) and CspA (grey) reveal that a fully interactive crowder can occupy the native binding site for the Cc/CcP complex, lowering the binding affinity in the single crowder simulation. Numerous samples indicate that the presence of a Cc/CspA or CcP/CspA will inhibit the formation of the native Cc/CcP complex. (b) The radial distribution of the fully interactive crowder and Cc/CcP shows that the crowder molecule is able to approach both Cc and CcP more closely than its repulsive counterpart. Both Cc/CspA and CcP/CspA begin emerge at radii related to molecular boundaries.

Samples supporting this clearly illustrate that the CspA protein binds with Cc in the native binding site of CcP (Figure 9.9a). Thus, less stable Cc/CcP complexes are formed at secondary binding sites, causing a lower binding affinity. The radial distribution of the fully interactive crowder supports this finding, with the distributions of CcP/CspA and Cc/CspA approaching 1 at much lower radii corresponding to the molecular boundaries of the respective complexes (Figure 9.9b). A comparison of fully interactive crowding and repulsive crowding for the simulations performed here is left as future work, promising some interesting results.

While we produce results consistent with scaled particle theory and in agreement with purely repulsive hard spheres, the environment within a real cell is fully interactive, requiring the simulation of interactive proteins to attempt to replicate these *in vivo* conditions. This task the subject of future work with this model and implementation.

9.3 Viral Capsid Construction

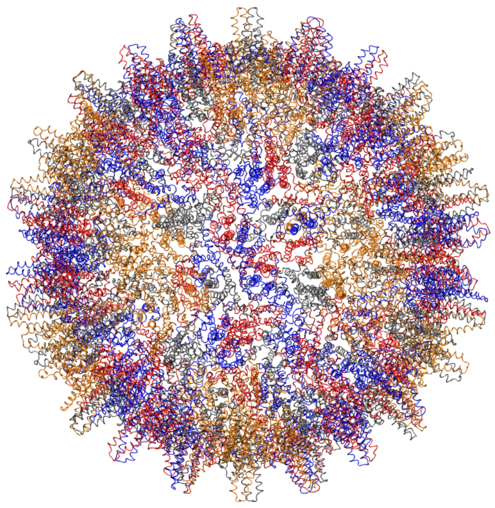
The final application of docking performed using our implementation is that of viral capsid assembly. Viruses are infectious agents about 100th of the size of bacteria. Viruses, unlike bacteria can be considered to be non-living because they lack a cellular structure and use their host to replicate and synthesise new products. There are over 5000 types of known viruses, many gaining notoriety throughout history for causing epidemics and pandemics [165].

Virus life cycles differ depending on the type of virus, but the cycle adheres to six basic stages: the binding or fusion of the viral capsid to receptors on the host cells surface, penetration of the virus into the host cell, the release of viral genomic nucleic acid into the host cell, replication and assembly within the host cell, post-translational modification of the viral proteins and, finally, release from the host cell [165].

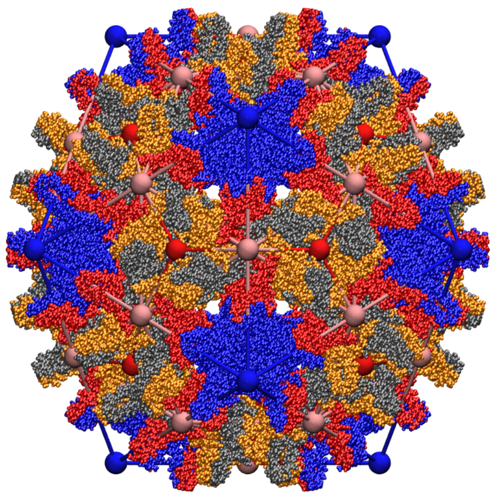
Viruses consist of either a deoxyribonucleic acid (DNA) or ribonucleic acid (RNA) molecule contained in a protective package. The package transmits the infectious agent in a functionally intact state to a susceptible host cell. In most animal viruses this package or capsid, is spherical with icosahedral symmetry [165] (Figure 9.10b)

The smallest DNA viruses are the hepadnaviruses, such as Hepatitis B, which is 42 nm across [165]. This virus consists of 240 similar chains, labelled A, B, C and D (Figures 9.10e and 9.10f) forming “V-shaped” tetrameric complexes (PDB ID:2G24). These complexes tessellate in the manner depicted in Figure 9.10d, with five complexes forming a sub-capsid hood structure (Figure 9.10c). The full capsid consists of 12 of these hoods, and a total of 240 peptide chains. This equates to 279480 atoms or 35280 coarse-grained beads.

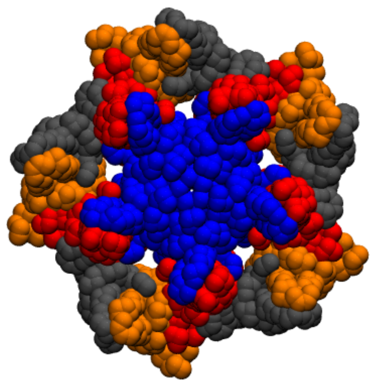
The formation of the capsids themselves is an marvel of nature; the protein subunits assemble into complete, reproducible structures under different conditions, avoiding kinetic and thermodynamic traps [186]. Understanding of this process has enormous potential impact in the



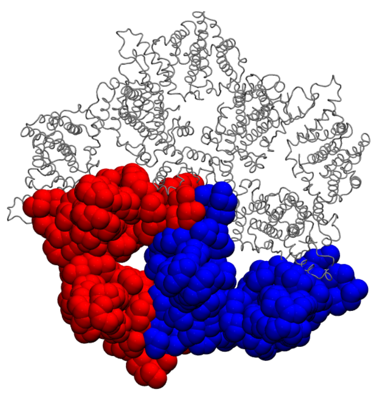
(a) Hepatitis B



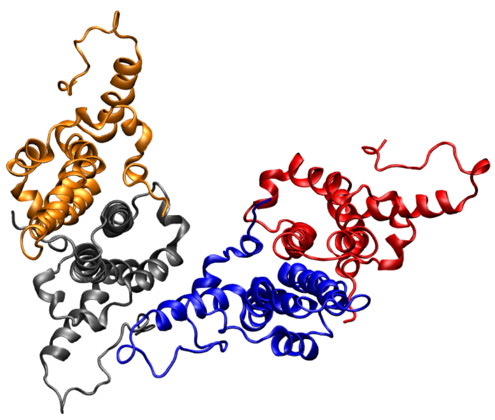
(b) HBV (All-Atom with lattice)



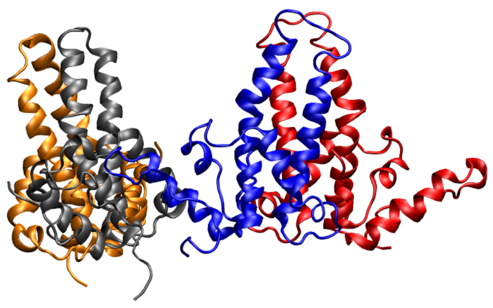
(c) Sub-capsid



(d) Docked Viral Proteins



(e) AB:CD Chains (top)



(f) AB:CD Chains (side)

Figure 9.10: Hepatitis B

a) The full Hepatitis B Capsid, illustrating the shape of the peptide chains only. b) The capsid comprises sub-complexes tessellated on a truncated icosahedral lattice. c) The capsid comprises 12 subunits of 20 peptide chains. d) Subunits form by docking "V-shaped" tetrameric complexes (PDB ID:2G24). e and f) Top and side views illustrating the secondary, tertiary and quaternary structures of the 2G34 complex.

design of antiviral drugs inhibiting the construction, and consequently, the propagation of the virus. Although viruses assemble with the aid of a DNA or RNA molecule, effectively providing a scaffold upon which to assemble. The study of the dynamics of the capsid pieces is important in understanding the assembly process.

Hagan et al. have studied the HBV capsid at a coarse-grained level [186], but, this coarse-grain representation encapsulated an entire 4-chain complex (depicted in figures 9.10e and 9.10f) as a single bead. Advances in computational tractability, such as our GPU implementation allow us to inspect these interactions at a residue level, and thus, increase the complexity and specificity of interactions in the simulation. Thus, 60 capsomer beads used by Hagan et al. to model the HBV capsid are scaled to 60 molecules, each containing 584 residues and their full geometry.

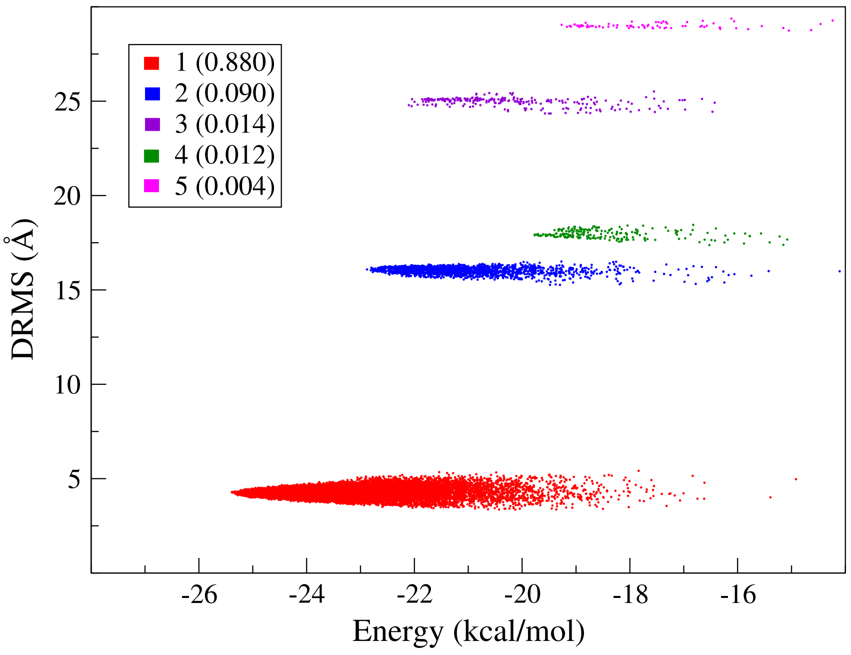
Due to the size of the viral capsid, it is an ideal test candidate for the study of large assemblies with the GPU implementation. Ultimately, the aim of such a study would be to simulate the assembly mechanism of viral capsids from their constituent components. To test the feasibility of such a study, a trial docking simulation of a pair of protein fragments (Figure 9.11 was performed.

The results from this trial are very encouraging. The bound conformations were initially clustered using the same method as for the Ubiquitin/UIM complex. Due to the symmetry of the participants, some of the clusters were then combined once the symmetry of the complex (i.e., AB:CD is equivalent to CD:AB) was accounted for, resulting in five clusters emerging.

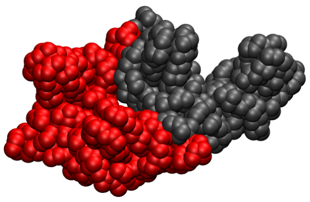
Figure 9.11a plots the distance root-mean-square (DRMS) of structures from each cluster against their energy, showing that the near native cluster, with a DRMS of 4.2Å and energy of 24.9 kcal/mol, to be the dominant emergent structure. Second to this is a malformed complex, where the proteins pack too closely together in the space available. This complex has a DRMS 16Å from the native configuration, but, significantly, has a low interaction potential of -22.7 kcal/mol. A number of subsidiary, higher energy, bound clusters are also identified, indicative of mild frustration on the binding energy landscape.

When simulations attempt to discover the docking pose between only two (or three) proteins, the frustration is handled by the ability of simulations to move most of the proteins away from the bound configuration due to the Boltzmann distribution. But once large assemblies of strongly interacting proteins are present, individual proteins become trapped due to the stability of the malformed complexes. The topology of the proteins results in many complexes with interaction potentials of -10 kcal/mol and -21 kcal/mol occurring.

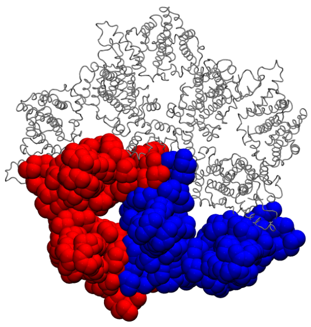
Unfortunately, due to the complexity, in number and topology of proteins in the viral capsid and the manner in which Monte-Carlo simulations occur in the current implementation, the



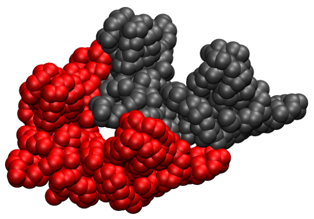
(a) DRMS Clustering



(b) Cluster 1



(c) Native



(d) Cluster 2

Figure 9.11: HBV Clusters

Clustering HBV simulations discovers 5 clusters, grouping and treating symmetric poses as belonging to the same cluster. (b) The cluster with the highest population, 88% of all samples, forms a complex within 4.2Å of the native configuration, (c). The second largest cluster forms an incorrectly docked structure (d), with a considerably higher DRMS of 16.0Å. However, the interaction potential of both clusters is similar, -24.9 kcal/mol and -22.7 kcal/mol respectively.

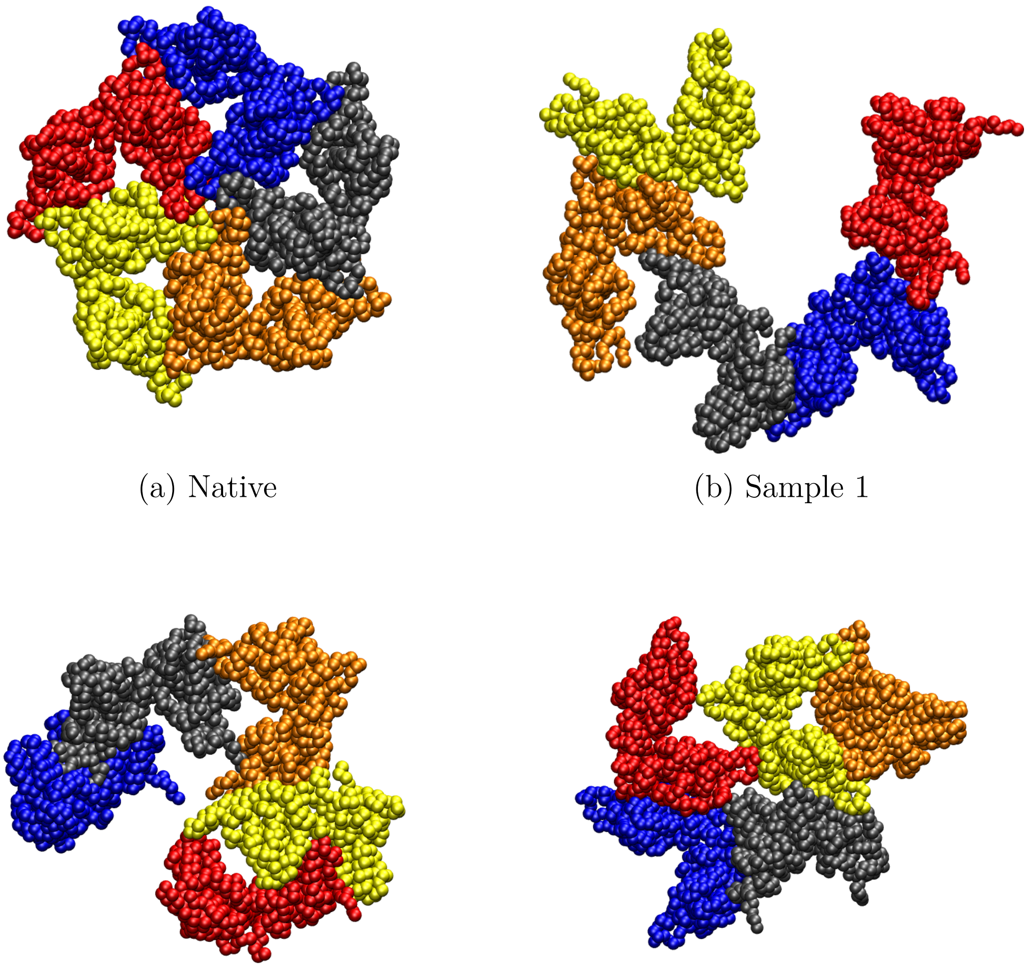


Figure 9.12: Capsid Hood Configurations
(a) The native capsid hood with an interaction potential of -48.1 kcal/mol. (b-d) Simulation samples from simulating the assembly of the viral capsid hood. These samples range in interaction potential from -43.1 kcal/mol to -58.4 kcal/mol.

construction of a capsid hood (Figure 9.12a) cannot occur without some modification. Figure 9.12 illustrates the effect of the frustration on multiple proteins. In figure 9.12, proteins occurring frustrated orientations result in the misinformation of the capsid hood. The interaction potential of the native configuration is -48.1 kcal/mol. From these configurations, it is clear that using the interaction potential as a measure of native state is insufficient as the simulated structures, all with major configurational faults, have interaction potentials of -43.1 kcal/mol (Figure 9.12b), -51.0 kcal/mol (figure 9.12c) and -58.4 kcal/mol (figure 9.12d). Hagan et al. note that the strength of the interaction potential is critical in the formation of the capsid. If the interactions between the capsomers is too strong, simulations are unable to escape the potential traps formed by non-native complexes [186].

To facilitate the formation of native viral complexes, the use of either a weaker potential or higher temperature may be required. If the electrostatics of the potential cause the frustration, increased ionic strengths may improve the simulations ability to find the native structure [180].

If this model is used for larger simulations, it is likely that effects similar to those experienced by Hagan et al. [186] will occur. In their model, sub-capsids assemble, but form frustrated, incomplete, malformed or unclosed capsids due to the orientations and assembly order of the viral fragments [186].

Our results clearly show that Kim and Hummer's simulation and model [12] will discover the native structure between the viral subunits, but lacks the ability to maintain the structure necessary for assembling the viral proteins, such that all of them occupy near native poses simultaneously. This feature is required if an entire capsid is to be assembled by Monte-Carlo simulation.

Chapter 10

Conclusions

In this work, we document a successful parallel CPU-GPU CUDA implementation of the Kim and Hummer coarse-grained model for replica exchange Monte Carlo (REMC) protein simulations. This software is designed specifically to exploit current heterogeneous GPU-CPU computing architectures and shows excellent speed-up and scalability compared to a single processor implementation. This type of relatively low-cost parallel architecture has great relevance for researchers in developing countries such as South Africa, where High Performance Computing centres are generally not available.

We have described in detail the approach and optimisations required to achieve maximum performance for our parallel code. The final optimal heterogeneous parallel implementation employs multithreading for the concurrent Monte Carlo simulations required by the replica exchange protocol. Each simulation thread performs asynchronous calls to the potential evaluation kernel running on the GPU. Optimisations focussed on this interaction potential calculation, as our detailed code profiling shows that the pairwise force calculations consume over 90% of the run time, even for the smallest systems.

On the GPU, we found that the latency of global memory accesses was the chief bottleneck. Performance on the GPU was further degraded by the random access pattern of the interaction potential lookups for the short range potential. We achieved optimal performance through adherence to the tabled CUDA best practices: maximising parallel execution (kernel occupancy) and optimizing memory and instruction usage. Of these, alleviation of latency through the use of shared memory had the most impact on GPU runtime. However, as the random contact potential lookups do not match the perfect GPU programming model, counter-intuitive memory configurations also performed surprisingly well. Generally, we found that the use of texture memory minimised the random access penalty, resulting in a 10 to 20% improvement over other types of memory. However, the impact of kernel occupancy is greatest: we found that once occupancy is raised to 37.5% or more, the latency of contact potential lookups is effectively hidden and even the relative performance advantage of texture memory over shared memory for storage of the residue data is removed. Further, we found that optimisation of the thread block size is important for small simulations. Simulations of fewer than 100 residues perform best with a thread block size of 32 threads because this maximises block-level parallelism and the work efficiency of thread warps. For larger simulations, the use of 64 threads per block shows the benefits of greater block-level parallelism than 128 threads, highlighting that, once simulations saturate GPU resources, high occupancy becomes increasingly important.

Benchmarks of simulations between 100 and 7668 residues experience in between 10 and 1400 times speed-up on a single GPU-CPU pairing depending on the configuration. By utilizing multiple GPUs and CPU cores, this speed-up scales the Monte-Carlo simulations near linearly. Application studies have shown, informally, that our simulations scale with 99% and 94% efficiency to 2 and 3 GPU's respectively due to the separability and low PCI-E bus dependence of our simulations. Optimal performance is achieved though concurrent computation on the GPU and CPU, overlapping Monte-Carlo moves on the CPU and interaction potential calculations on the GPU using asynchronous GPU calls. For a single GPU and CPU thread, this results in almost double the performance of sharing the GPU between multiple CPU threads using synchronous GPU calls.

In terms of utility, a dual-core machine with one GTX280 card is capable of up to 9 million Monte-Carlo iterations per day when performing a simulation containing 7668 residues using GPU-accelerated code. By contrast, a CPU, fully utilizing both cores, manages only 12 000 iterations per day. Practically, this means a 10 million iteration simulation needs just over a day using a GPU but over 800 days if it only uses the CPU. In nearly all cases, GPU acceleration decreases simulation runtime by two orders of magnitude.

Verification and validation of our GPU code shows that it is capable of reproducing known results and that the effect of the inaccuracies of GPU mathematics hardware are manageable by delegation of appropriate operations to either the CPU or GPU.

Three case study applications were successfully performed using the implementation. We find that inclusion of the ubiquitin C-terminus tail increases both the binding affinity and the specificity of binding for the UIM/Ub system, even with a rigid model. This sees proportion of native to inverse configurations at the binding site rising from 80:20 to 86:14.

The most significant application performed using our implementation is the macromolecular crowding of cytochrome *c* and cytochrome *c* peroxidase by CspA. Using a repulsive crowder model, we find our simulations to produce results in agreement with scaled particle theory and Kim et al. [185]. We find that crowded simulations increase both the binding affinity and the specificity of binding as the packing fraction of these simulations increases. Preliminary investigation into fully repulsive crowding is promising, but reveals that the complex interactions with fully attractive and repulsive residues in all proteins will alter the behaviour of the simulation because of the blocking effect caused by crowder complexes in otherwise native binding pockets.

These crowding simulations are also an excellent illustration of the utility of a fast simulation system. Refinement of the repulsive crowder model and the detection and correction of any human errors in performing the simulations is also accelerated. A complete representative set of simulations can be performed in less than 2 weeks on a single GPU enabled desktop. To perform the same simulations a cluster containing 48 cores would require over a month of continuous, exclusive use. Over the course of our studies, these simulations were performed a total of five times.

Our final simulation of the docking of two HBV capsid components proves that the model can locate the native complex formed by a pair of these viral proteins. However, the energetic frustration experienced by fragments composed of multiple viral proteins denotes that refinement of the model is required before complete capsids or sub-capsids can be assembled using Kim's method. Accomplishment of this process will give new insights into molecular self-assembly and may yield insights useful in the development of therapeutic drugs.

Finally, we note that, although we have only considered a specific model for protein-protein interactions, our implementation could easily be generalized to other types of interaction functions, as well as to other types of coarse-grained macromolecules (e.g. DNA). The effective parallelisation approach developed in this work is generally applicable to N-body problems that require similar random access to lookup tables, where aspects of the interaction between bodies are dependent on their type or state. More immediate improvements to our docking implementation are in the form of flexible linkers. This addition to our simulation will enable the refinement of protein docking poses for non-rigid molecules, e.g. the ubiquitin tail, allowing for the discovery of structures even closer to experimentally observed native structures. The addition of these linkers is of little additional computational cost, merely an altered Monte-Carlo mutation and the linear time complexity bonded force evaluations on either the CPU or GPU.

Flexibility and refinement can also be incorporated by the inclusion of a molecular dynamics simulation based on the Kim-Hummer potential, vectorising and integrating the potentials to derive updated positions on the GPU as opposed to performing an interaction potential reduction. Such a simulation will be amenable to greater speed-ups than the existing code as more tasks can be performed in parallel on the GPU.

References

[1] A. Tramontano, *The Ten Most Wanted Solutions in Protein Bioinformatics (Chapman & Hall/CRC Mathematical & Computational Biology).* Chapman & Hall, 1 ed., May 2005.

[2] N. A. Pierce and E. Winfree, “Protein Design is NP-hard,” *Protein Eng.*, vol. 15, pp. 779–782, October 2002.

[3] I. Halperin, B. Ma, H. Wolfson, and R. Nussinov, “Principles of docking: An overview of search algorithms and a guide to scoring functions.,” *Proteins*, vol. 47, pp. 409–443, June 2002.

[4] M. Levitt, “A simplified representation of protein conformations for rapid simulation of protein folding.,” *Journal of Molecular Biology*, vol. 104, pp. 59–107, June 1976.

[5] M. S. Friedrichs and P. G. Wolynes, “Toward protein tertiary structure recognition by means of associative memory hamiltonians.,” *Science (New York, N.Y.)*, vol. 246, pp. 371–373, October 1989.

[6] E. I. Shakhnovich and A. M. Gutin, “Implications of thermodynamics of protein folding for evolution of primary sequences,” *Nature*, vol. 346, pp. 773–775, August 1990.

[7] J. Skolnick and A. Kolinski, “Simulations of the Folding of a Globular Protein,” *Science*, vol. 250, pp. 1121–1125, November 1990.

[8] J. Karanicolas and C. L. Brooks, “The origins of asymmetry in the folding transition states of protein L and protein G.,” *Protein Sci*, vol. 11, pp. 2351–2361, October 2002.

[9] N.-V. Buchete, J. E. Straub, and D. Thirumalai, “Anisotropic coarse-grained statistical potentials improve the ability to identify nativelike protein structures,” *The Journal of Chemical Physics*, vol. 118, no. 16, pp. 7658–7671, 2003.

[10] V. Tozzini, “Coarse-grained models for proteins,” *Current Opinion in Structural Biology*, vol. 15, pp. 144–150, April 2005.

[11] S. J. Marrink, H. J. Risselada, S. Yefimov, D. P. Tieleman, and A. H. de Vries, “The MARTINI Force Field: A Coarse Grained Model for Biomolecular Simulations,” *The Journal of Physical Chemistry B*, vol. 111, pp. 7812–7824, July 2007.

[12] Y. C. Kim and G. Hummer, "Coarse-grained Models for Simulations of Multiprotein Complexes: Application to Ubiquitin Binding," *Journal of Molecular Biology*, vol. 375, pp. 1416–1433, February 2008.

[13] D. Kirk and W.-M. Hwu, *Programming massively parallel processors : a hands-on approach*. Morgan Kaufmann Publishers, 2010.

[14] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, "Dense linear algebra solvers for multicore with GPU accelerators," in *Proceedings of IPDPS 2010: 24th IEEE International Parallel and Distributed Processing Symposium*, pp. 1–8, April 2010.

[15] L. Nyland, M. Harris, and J. Prins, "Fast N-Body Simulation with CUDA," in *GPU Gems 3* (H. Nguyen, ed.), ch. 31, Addison Wesley Professional, August 2007.

[16] R. G. Belleman, J. Bédorf, and S. F. Portegies Zwart, "High performance direct gravitational N-body simulations on graphics processing units II: An implementation in CUDA," *New Astronomy*, vol. 13, pp. 103–112, February 2008.

[17] M. S. Friedrichs, P. Eastman, V. Vaidyanathan, M. Houston, S. Legrand, A. L. Beberg, D. L. Ensign, C. M. Bruns, and V. S. Pande, "Accelerating molecular dynamic simulation on graphics processing units," *Journal of Computational Chemistry*, vol. 30, pp. 864–872, April 2009.

[18] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten, "Accelerating molecular modeling applications with graphics processors," *Journal of Computational Chemistry*, vol. 28, pp. 2618–2640, September 2007.

[19] C. I. Rodrigues, D. J. Hardy, J. E. Stone, K. Schulten, and W. Mei, "GPU acceleration of cutoff pair potentials for molecular modeling applications," in *CF '08: Proceedings of the 5th conference on Computing frontiers*, (New York, NY, USA), pp. 273–282, ACM, 2008.

[20] D. J. Hardy, J. E. Stone, and K. Schulten, "Multilevel summation of electrostatic potentials using graphics processing units," *Parallel Computing*, vol. 35, pp. 164–177, March 2009.

[21] N. Schmid, M. Bötschi, and W. F. Van Gunsteren, "A GPU solvent-solvent interaction calculation accelerator for biomolecular simulations using the GROMOS software," *J. Comput. Chem.*, vol. 31, no. 8, pp. 1636–1643, 2010.

[22] NVIDIA, "CUDA Programming Guide 2.3." http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf Last accessed: 2009-09-08, 2009.

[23] NVIDIA, "CUDA Best Practices Guide 2.3." http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide_2.3.pdf Last accessed: 2009-09-08, 2009.

[24] A. D. Mackerell, “Empirical force fields for biological macromolecules: Overview and issues,” *J. Comput. Chem.*, vol. 25, pp. 1584–1604, October 2004.

[25] J. Anderson, C. Lorenz, and A. Travesset, “General purpose molecular dynamics simulations fully implemented on graphics processing units,” *Journal of Computational Physics*, vol. 227, pp. 5342–5359, May 2008.

[26] J. A. van Meel, A. Arnold, D. Frenkel, O. Portegies, and R. G. Belleman, “Harvesting graphics power for MD simulations,” *Molecular Simulation*, vol. 34, no. 3, pp. 259–266, 2008.

[27] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus, “CHARMM: A program for macromolecular energy, minimization, and dynamics calculations,” *J. Comput. Chem.*, vol. 4, pp. 187–217, February 1983.

[28] H. Pelletier and J. Kraut, “Crystal structure of a complex between electron transfer partners, cytochrome c peroxidase and cytochrome c.,” *Science (New York, N.Y.)*, vol. 258, pp. 1748–1755, December 1992.

[29] nVIDIA, “GeForce 256.” http://www.nvidia.com/page/geforce256.html Last accessed: 2010-10-04.

[30] nVIDIA, “GeForce GTX 480.” http://www.nvidia.com/object/product_geforce_gtx_480_us.html Last accessed: 2009-10-04.

[31] ATI, “ATI Radeon HD 5870 Graphics.” http://www.amd.com/us/products/desktop/graphics/ati-radeon-hd-5000/hd-5870 Last accessed: 2009-10-04.

[32] J. Hensley, “AMD CTM overview,” in *SIGGRAPH ’07: ACM SIGGRAPH 2007 courses*, (New York, NY, USA), p. 7, ACM, 2007.

[33] M. Olano and A. Lastra, “A shading language on graphics hardware: the pixelflow shading system,” in *SIGGRAPH ’98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, (New York, NY, USA), pp. 159–168, ACM, 1998.

[34] M. S. Peercy, M. Olano, J. Airey, and P. J. Ungar, “Interactive multi-pass programmable shading,” in *SIGGRAPH ’00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, (New York, NY, USA), pp. 425–432, ACM Press/Addison-Wesley Publishing Co., 2000.

[35] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, “Brook for GPUs: stream computing on graphics hardware,” in *SIGGRAPH ’04: ACM SIGGRAPH 2004 Papers*, (New York, NY, USA), pp. 777–786, ACM Press, 2004.

[36] R. Fernando and M. J. Kilgard, *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics.* Addison-Wesley Professional, March 2003.

[37] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. Lefohn, and T. J. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[38] M. J. Harris, G. Coombe, T. Scheuermann, and A. Lastra, "Physically-based visual simulation on graphics hardware," in *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pp. 109–118, Eurographics Association, 2002.

[39] W. Li, X. Wei, and A. E. Kaufman, "Implementing Lattice Boltzmann Computation on Graphics Hardware," *The Visual Computer*, vol. 19, pp. 444–456, 2003.

[40] J. Krüger and R. Westermann, "Linear algebra operators for GPU implementation of numerical algorithms," in *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, (New York, NY, USA), p. 234, ACM, 2005.

[41] M. Harris, "Fast fluid dynamics simulation on the GPU," in *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, (New York, NY, USA), p. 220, ACM, 2005.

[42] A. Kolb and N. Cuntz, "Dynamic particle coupling for GPU-based fluid simulation," in *In Proc. of the 18th Symposium on Simulation Technique*, pp. 722–727, 2005.

[43] H. Nguyen, *GPU Gems 3*. Addison-Wesley Professional, 2007.

[44] R. Strzodka and C. Garbe, "Real-Time Motion Estimation and Visualization on Graphics Cards," in *VIS '04: Proceedings of the conference on Visualization '04*, (Washington, DC, USA), pp. 545–552, IEEE Computer Society, 2004.

[45] M. Y. Ansari, "Video Image Processing Using Shaders," 2003.

[46] A. E. Lefohn, J. M. Kniss, C. D. Hansen, and R. T. Whitaker, "Interactive Deformation and Visualization of Level Set Surfaces Using Graphics Hardware," in *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, (Washington, DC, USA), p. 11, IEEE Computer Society, 2003.

[47] J.-P. Farrugia, P. Horain, E. Guehenneux, and Y. Alusse, "GPUCV: A Framework for Image Processing Acceleration with Graphics Processors," in *2006 IEEE International Conference on Multimedia and Expo*, pp. 585–588, IEEE, December 2006.

[48] K. Moreland and E. Angel, "The FFT on a GPU," in *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pp. 112–119, Eurographics Association, 2003.

[49] M. Hopf and T. Ertl, "Hardware Accelerated Wavelet Transformations," in *Proc. TCVG Symposium on Visualization*, pp. 93–103, 2000.

[50] E. S. Larsen and D. Mcallister, "Fast matrix multiplies using graphics hardware," in *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, p. 55, ACM Press, 2001.

[51] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder, "Sparse matrix solvers on the GPU: conjugate gradients and multigrid," in *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, (New York, NY, USA), pp. 917–924, ACM, 2003.

[52] S. Popov, J. Günther, H. P. Seidel, and P. Slusallek, "Stackless KD-Tree Traversal for High Performance GPU Ray Tracing," *Computer Graphics Forum*, vol. 26, pp. 415–424, September 2007.

[53] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan, "Photon mapping on programmable graphics hardware," in *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pp. 41–50, Eurographics Association, 2003.

[54] J. Hable and J. Rossignac, "Blister: GPU-based rendering of Boolean combinations of free-form triangulated shapes," in *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, (New York, NY, USA), pp. 1024–1031, ACM, 2005.

[55] L. Bavoil, S. P. Callahan, A. Lefohn, Ao, and C. T. Silva, "Multi-fragment effects on the GPU using the k-buffer," in *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, (New York, NY, USA), pp. 97–104, ACM, 2007.

[56] M. Olano, B. Kuehne, and M. Simmons, "Automatic shader level of detail," in *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pp. 7–14, Eurographics Association, 2003.

[57] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha, "Fast Computation of Database Operations Using Graphics Processors," in *Proc. of ACM SIGMOD*, pp. 215–226, 2004.

[58] "OpenCL overview." http://www.khronos.org/opencl Last accessed: 2009-10-04.

[59] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: a many-core x86 architecture for visual computing," *ACM Trans. Graph.*, vol. 27, pp. 1–15, August 2008.

[60] "nVIDIA's Next Generation CUDA Compute Architecture: Fermi." http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf Last accessed: 2010-08-23.

[61] J. E. Stone, D. J. Hardy, I. S. Ufimtsev, and K. Schulten, "GPU-accelerated molecular modeling coming of age," *Journal of Molecular Graphics and Modelling*, vol. 29, pp. 116–125, September 2010.

[62] P. Eastman and V. S. Pande, "Efficient nonbonded interactions for molecular dynamics on a graphics processing unit," *J. Comput. Chem.*, vol. 31, pp. 1268–1272, October 2010.

[63] M. J. Harvey, G. Giupponi, and G. D. Fabritiis, "ACEMD: Accelerating Biomolecular Dynamics in the Microsecond Time Scale," *Journal of Chemical Theory and Computation*, vol. 5, pp. 1632–1639, June 2009.

[64] J. P. Walters, V. Balu, S. Kompalli, and V. Chaudhary, "Evaluating the use of GPUs in liver image segmentation and HMMER database searches," in *IEEE International Symposium on Parallel & Distributed Processing (IPDPS'09)*, pp. 1–12, 2009.

[65] H. Shi, B. Schmidt, W. Liu, and W. Mueller-Wittig, "Accelerating Error Correction in High-Throughput Short-Read DNA Sequencing Data with CUDA," in *IEEE International Workshop on High Performance Computational Biology (HiCOMB 2009)*, 2009.

[66] D. Komatitsch, D. Michéa, and G. Erlebacher, "Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA," *Journal of Parallel and Distributed Computing*, vol. 69, pp. 451–460, May 2009.

[67] T. Preis, P. Virnau, W. Paul, and J. J. Schneider, "GPU accelerated Monte Carlo simulation of the 2D and 3D Ising model," *Journal of Computational Physics*, vol. 228, pp. 4468–4477, July 2009.

[68] S. S. Stone, J. P. Haldar, S. C. Tsao, Hwu, B. P. Sutton, and Z. P. Liang, "Accelerating advanced MRI reconstructions on GPUs," *J. Parallel Distrib. Comput.*, vol. 68, no. 10, pp. 1307–1318, 2008.

[69] C. Men, X. Gu, D. Choi, A. Majumdar, Z. Zheng, K. Mueller, and S. B. Jiang, "GPU-based ultrafast IMRT plan optimization," *Physics in Medicine and Biology*, vol. 54, pp. 6565–6573, November 2009.

[70] B. Zhang, X. Yang, F. Yang, X. Yang, C. Qin, D. Han, X. Ma, K. Liu, and J. Tian, "The CUBLAS and CULA based GPU acceleration of adaptive finite element framework for bioluminescence tomography," *Opt. Express*, vol. 18, pp. 20201–20214, September 2010.

[71] N. Singla, M. Hall, B. Shands, and R. D. Chamberlain, "Financial Monte Carlo simulation on architecturally diverse systems," in *2008 Workshop on High Performance Computational Finance*, pp. 1–7, IEEE, November 2008.

[72] V. Simek, R. Dvorak, F. Zboril, and J. Kunovsky, "Towards Accelerated Computation of Atmospheric Equations Using CUDA," *Computer Modeling and Simulation, International Conference on*, vol. 0, pp. 449–454, 2009.

[73] R. Kelly, "GPU Computing for Atmospheric Modeling," *Computing in Science and Engineering*, vol. 12, no. 4, pp. 26–33, 2010.

[74] M. W. Govett, J. Middlecoff, and T. Henderson, “Running the NIM Next-Generation Weather Model on GPUs,” *Cluster Computing and the Grid, IEEE International Symposium on*, vol. 0, pp. 792–796, 2010.

[75] Y. Luo and R. Duraiswami, “Canny edge detection on NVIDIA CUDA,” in *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pp. 1–8, IEEE, June 2008.

[76] X. H. H. Wang and W. F. Good, “Real-time stereographic rendering and display of medical images with programmable GPUs.,” *Computerized medical imaging and graphics : the official journal of the Computerized Medical Imaging Society*, vol. 32, pp. 118–123, March 2008.

[77] J. Fung and S. Mann, “Using graphics devices in reverse: GPU-based Image Processing and Computer Vision,” in *2008 IEEE International Conference on Multimedia and Expo*, pp. 9–12, IEEE, June 2008.

[78] B. Pieters, D. V. Rijsselbergen, W. D. Neve, and R. V. de Walle, “Performance evaluation of H.264/AVC decoding and visualization using the GPU,” *Applications of Digital Image Processing XXX*, vol. 6696, no. 1, pp. 606–629, 2007.

[79] Y. Allusse, P. Horain, A. Agarwal, and C. Saipriyadarshan, “GpuCV: A GPU-Accelerated Framework for Image Processing and Computer Vision,” in *Advances in Visual Computing* (G. Bebis, R. Boyle, B. Parvin, D. Koracin, P. Remagnino, F. Porikli, J. A. Peters, J. Klosowski, L. Arns, Y. Chun, T.-M. Rhyne, and L. Monroe, eds.), vol. 5359 of *Lecture Notes in Computer Science*, ch. 42, pp. 430–439–439, Berlin, Heidelberg: Springer Berlin/Heidelberg, 2008.

[80] J. Owens, “Data-parallel algorithms and data structures,” in *SIGGRAPH ’07: ACM SIGGRAPH 2007 courses*, (New York, NY, USA), p. 3, ACM, 2007.

[81] W. Kahan, “Pracniques: further remarks on reducing truncation errors,” *Commun. ACM*, vol. 8, no. 1, p. 40, 1965.

[82] M. Harris, “Optimizing Parallel Reduction in CUDA.” http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf Last accessed: 2009-09-08.

[83] J. Yang, Y. Wang, and Y. Chen, “GPU accelerated molecular dynamics simulation of thermal conductivities,” *Journal of Computational Physics*, vol. 221, pp. 799–804, February 2007.

[84] G. R. Smith and M. J. Sternberg, “Prediction of protein-protein interactions by docking methods.,” *Current Opinion in Structural Biology*, vol. 12, pp. 28–35, February 2002.

[85] A. Tozeren and S. W. Byers, *New Biology for Engineers and Computer Scientists*. Prentice Hall, May 2003.

[86] G. Marshall and I. Vakser, "Protein-Protein Docking Methods," in *Proteomics and Protein-Protein Interactions* (G. Waksman, ed.), vol. 3 of *Protein Reviews*, ch. 6, pp. 115–146–146, Boston, MA: Springer US, 2005.

[87] W. Humphrey, A. Dalke, and K. Schulten, "VMD – Visual Molecular Dynamics," *Journal of Molecular Graphics*, vol. 14, pp. 33–38, 1996.

[88] C. J. Tsai, S. Kumar, B. Ma, and R. Nussinov, "Folding funnels, binding funnels, and protein function.," *Protein Science*, vol. 8, pp. 1181–1190, June 1999.

[89] K. Lee, "Computational study for protein-protein docking using global optimization and empirical potentials.," *International Journal of Molecular Sciences*, vol. 9, pp. 65–77, January 2008.

[90] A. Bonvin, "Flexible protein-protein docking," *Current Opinion in Structural Biology*, vol. 16, pp. 194–200, April 2006.

[91] L. P. Ehrlich, M. Nilges, and R. C. Wade, "The impact of protein flexibility on protein-protein docking," *Proteins*, vol. 58, pp. 126–133, January 2005.

[92] A. Solernou and J. Fernández-Recio, "Refinement of rigid-body protein-protein docking using backbone," *Open Access Bioinformatics*, vol. 2010.2, pp. 19–27, April 2010.

[93] L. Li, R. Chen, and Z. Weng, "RDOCK: refinement of rigid-body protein docking predictions.," *Proteins*, vol. 53, pp. 693–707, November 2003.

[94] A. Tovchigrechko and I. A. Vakser, "How common is the funnel-like energy landscape in protein-protein interactions?," *Protein Science*, vol. 10, pp. 1572–1583, August 2001.

[95] J. E. Jones, "On the Determination of Molecular Fields. II. From the Equation of State of a Gas," *Proceedings of the Royal Society of London. Series A*, vol. 106, pp. 463–477, October 1924.

[96] S. Wodak, "Computer analysis of protein-protein interaction," *Journal of Molecular Biology*, vol. 124, pp. 323–342, September 1978.

[97] E. Katchalski-Katzir, I. Shariv, M. Eisenstein, A. A. Friesem, C. Aflalo, and I. A. Vakser, "Molecular surface recognition: determination of geometric fit between proteins and their ligands by correlation techniques," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 89, pp. 2195–2199, March 1992.

[98] H. A. Gabb, R. M. Jackson, and M. J. Sternberg, "Modelling protein docking using shape complementarity, electrostatics and biochemical information.," *Journal of Molecular Biology*, vol. 272, pp. 106–120, September 1997.

[99] P. Burkhard, P. Taylor, and M. D. Walkinshaw, "An example of a protein ligand found by database mining: description of the docking method and its verification by a 2.3 A

X-ray structure of a thrombin-ligand complex.,” *Journal of Molecular Biology*, vol. 277, pp. 449–466, March 1998.

[100] V. Sobolev, R. C. Wade, G. Vriend, and M. Edelman, “Molecular docking using surface complementarity.,” *Proteins*, vol. 25, pp. 120–129, May 1996.

[101] R. Chen, L. Li, and Z. Weng, “ZDOCK: an initial-stage protein-docking algorithm.,” *Proteins*, vol. 52, pp. 80–87, July 2003.

[102] F. Jiang and S. H. Kim, “"Soft Docking": matching of molecular surface cubes.,” *Journal of Molecular Biology*, vol. 219, pp. 79–102, May 1991.

[103] R. D. Taylor, P. J. Jewsbury, and J. W. Essex, “A review of protein-small molecule docking methods,” *Journal of Computer-Aided Molecular Design*, vol. 16, pp. 151–166, March 2002.

[104] Y.-P. P. Pang, E. Perola, K. Xu, and F. G. G. Prendergast, “EUDOC: a computer program for identification of drug interaction sites in macromolecules and drug leads from chemical databases.,” *Journal of Computational Chemistry*, vol. 22, pp. 1750–1771, November 2001.

[105] Y. P. Pang, T. J. Mullins, B. A. Swartz, J. S. McAllister, B. E. Smith, C. J. Archer, R. G. Musselman, A. E. Peters, B. P. Wallenfelt, and K. W. Pinnow, “EUDOC on the IBM Blue Gene/L system: Accelerating the transfer of drug discoveries from laboratory to patient,” *IBM Journal of Research and Development*, vol. 52, pp. 69–81, January 2008.

[106] G. Jones, “Development and validation of a genetic algorithm for flexible docking,” *Journal of Molecular Biology*, vol. 267, pp. 727–748, April 1997.

[107] G. M. Morris, D. S. Goodsell, R. S. Halliday, R. Huey, W. E. Hart, R. K. Belew, and A. J. Olson, “Automated docking using a Lamarckian genetic algorithm and an empirical binding free energy function,” *J. Comput. Chem.*, vol. 19, pp. 1639–1662, January 1998.

[108] C. M. Oshiro, I. D. Kuntz, and J. S. Dixon, “Flexible ligand docking using a genetic algorithm.,” *Journal of computer-aided molecular design*, vol. 9, pp. 113–130, April 1995.

[109] C. D. Rosin, R. S. Halliday, W. E. Hart, and R. K. Belew, “A Comparison of Global and Local Search Methods in Drug Docking,” in *In Proceedings of the Seventh International Conference on Genetic Algorithms*, pp. 221–228, 1997.

[110] D. K. Gehlhaar, G. M. Verkhivker, P. A. Rejto, C. J. Sherman, D. B. Fogel, L. J. Fogel, and S. T. Freer, “Molecular recognition of the inhibitor AG-1343 by HIV-1 protease: conformationally flexible docking by evolutionary programming.,” *Chemistry & biology*, vol. 2, pp. 317–324, May 1995.

[111] P. Khodade, R. Prabhu, N. Chandra, S. Raha, and R. Govindarajan, “Parallel implementation of *AutoDock*,” *Journal of Applied Crystallography*, vol. 40, pp. 598–599, Jun 2007.

[112] V. E. Lamberti, L. D. Fosdick, E. R. Jessup, and C. J. C. Schauble, "A Hands-On Introduction to Molecular Dynamics," *Journal of Chemical Education*, vol. 79, p. 601, May 2002.

[113] C. Sagui and T. A. Darden, "Molecular dynamics simulations of biomolecules: long-range electrostatic effects.," *Annual review of biophysics and biomolecular structure*, vol. 28, no. 1, pp. 155–179, 1999.

[114] K. Wiehe, M. W. Peterson, B. Pierce, J. Mintseris, and Z. Weng, "Protein-protein docking: overview and performance analysis.," *Methods in Molecular Biology*, vol. 413, pp. 283–314, 2008.

[115] J. A. McCammon, B. R. Gelin, and M. Karplus, "Dynamics of folded proteins.," *Nature*, vol. 267, pp. 585–590, June 1977.

[116] P. L. Freddolino, A. S. Arkhipov, S. B. Larson, A. McPherson, and K. Schulten, "Molecular Dynamics Simulations of the Complete Satellite Tobacco Mosaic Virus," *Structure*, vol. 14, pp. 437–449, March 2006.

[117] D. L. Ensign, P. M. Kasson, and V. S. Pande, "Heterogeneity Even at the Speed Limit of Folding: Large-scale Molecular Dynamics Study of a Fast-folding Variant of the Villin Headpiece," *Journal of Molecular Biology*, vol. 374, pp. 806–816, November 2007.

[118] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kalé, and K. Schulten, "Scalable molecular dynamics with NAMD," *Journal of Computational Chemistry*, vol. 26, pp. 1781–1802, December 2005.

[119] D. A. Case, T. E. Cheatham, T. Darden, H. Gohlke, R. Luo, K. M. Merz, A. Onufriev, C. Simmerling, B. Wang, and R. J. Woods, "The AMBER biomolecular simulation programs," *Journal of Computatioanl Chemistry*, vol. 26, pp. 1668–1688, December 2005.

[120] E. Lindahl, B. Hess, and D. van der Spoel, "GROMACS 3.0: a package for molecular simulation and trajectory analysis," *Journal of Molecular Modeling*, vol. 7, pp. 306–317, August 2001.

[121] S. Plimpton, "Fast Parallel Algorithms for Short-Range Molecular Dynamics," *Journal of Computational Physics*, vol. 117, pp. 1–19, March 1995.

[122] L. Kalé, R. Skeel, M. Bh, R. Brunner, A. Gursoy, N. Krawetz, J. Phillips, A. Shinozaki, K. Varadarajan, and K. Schulten, "NAMD2: Greater scalability for parallel molecular dynamics," *Journal of Computational Physics*, vol. 151, pp. 283–312, 1999.

[123] "Amber (PMEMD) NVIDIA GPU Support Benchmarks." http://ambermd.org/gpus/benchmarks.htm Last Accessed: 2010-09-21.

[124] Z. Yao, "Improved neighbor list algorithm in molecular simulations using cell decomposition and data sorting method," *Computer Physics Communications*, vol. 161, pp. 27–35, August 2004.

[125] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, “Equation of State Calculations by Fast Computing Machines,” *The Journal of Chemical Physics*, vol. 21, no. 6, pp. 1087–1092, 1953.

[126] W. K. Hastings, “Monte Carlo Sampling Methods Using Markov Chains and Their Applications,” *Biometrika*, vol. 57, no. 1, pp. 97–109, 1970.

[127] B. Widom, *Statistical Mechanics: A Concise Introduction for Chemists.* Cambridge University Press, 1 ed., May 2002.

[128] P. Hellekalek, “Don’t trust parallel Monte Carlo!,” *SIGSIM Simul. Dig.*, vol. 28, pp. 82–89, July 1998.

[129] M. Matsumoto and T. Nishimura, “Mersenne Twister: a 623-dimensionally equidistributed uniform pseudo-random number generator,” *ACM Transactions on Modeling and Computer Simulation*, vol. 8, pp. 3–30, January 1998.

[130] D. J. Earl and M. W. Deem, “Parallel tempering: Theory, applications, and new perspectives,” *Physical Chemistry Chemical Physics*, vol. 7, no. 23, pp. 3910–3916, 2005.

[131] Y. Li, M. Mascagni, and A. Gorin, “A decentralized parallel implementation for parallel tempering algorithm,” *Parallel Computing*, vol. 35, pp. 269–283, December 2008.

[132] M. Eleftheriou, A. Rayshubski, J. W. Pitera, B. G. Fitch, R. Zhou, and R. S. Germain, “Parallel implementation of the replica exchange molecular dynamics algorithm on Blue Gene/L,” *International Parallel and Distributed Processing Symposium*, vol. 0, p. 281, 2006.

[133] H. Kokubo and Y. Okamoto, “Prediction of membrane protein structures by replica-exchange Monte Carlo simulations: Case of two helices,” *Journal of Chemical Physics*, vol. 120, pp. 10837–10847, June 2004.

[134] M. W. Maddox and M. L. Longo, “A Monte Carlo study of peptide insertion into lipid bilayers: equilibrium conformations and insertion mechanisms.,” *Biophysical Journal*, vol. 82, pp. 244–263, January 2002.

[135] W. Im and C. L. Brooks, “Interfacial folding and membrane insertion of designed peptides studied by molecular dynamics simulations.,” *Proceedings of the National Academy of Sciences of the United States of America*, vol. 102, pp. 6771–6776, May 2005.

[136] W. Y. Y. Yang, J. W. Pitera, W. C. Swope, and M. Gruebele, “Heterogeneous folding of the trpzip hairpin: full atom simulation and experiment.,” *Journal of Molecular Biology*, vol. 336, pp. 241–251, February 2004.

[137] D. Kihara, H. Lu, A. Kolinski, and J. Skolnick, “TOUCHSTONE: An *ab initio* protein structure prediction method that uses threading-based tertiary restraints,” *Proceedings of the National Academy of Sciences of the United States of America*, vol. 98, pp. 10125–10130, August 2001.

[138] A. E. García and K. Y. Sanbonmatsu, "Exploring the energy landscape of a β hairpin in explicit solvent," *Proteins*, vol. 42, no. 3, pp. 345–354, 2001.

[139] J. Nilmeier and M. P. Jacobson, "Monte Carlo Sampling with Hierarchical Move Sets: POSH Monte Carlo," *Journal of Chemical Theory and Computation*, vol. 5, pp. 1968–1984, August 2009.

[140] A. E. Roitberg, A. Okur, and C. Simmerling, "Coupling of Replica Exchange Simulations to a Non-Boltzmann Structure Reservoir," *The Journal of Physical Chemistry B*, vol. 111, pp. 2415–2418, March 2007.

[141] M.-H. Hao and H. A. Scheraga, "Monte Carlo Simulation of a First-Order Transition for Protein Folding," *The Journal of Physical Chemistry*, vol. 98, pp. 4940–4948, May 1994.

[142] J. W. Pitera and W. Swope, "Understanding folding and design: Replica-exchange simulations of "Trp-cage" miniproteins," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 100, pp. 7587–7592, June 2003.

[143] N. Rathore and J. J. de Pablo, "Monte Carlo simulation of proteins through a random walk in energy space," *The Journal of Chemical Physics*, vol. 116, no. 16, pp. 7225–7230, 2002.

[144] E. Shakhnovich, "Monte-Carlo Methods in Studies of Protein Folding and Evolution," in *Computer Simulations in Condensed Matter Systems: From Materials to Chemical Biology Volume 2* (M. Ferrario, G. Ciccotti, and K. Binder, eds.), vol. 704 of *Lecture Notes in Physics*, ch. 21, pp. 563–593–593, Springer Berlin / Heidelberg, 2006.

[145] S. Lorenzen and Y. Zhang, "Monte Carlo refinement of rigid-body protein docking structures with backbone displacement and side-chain optimization," *Protein Science*, vol. 16, pp. 2716–2725, December 2007.

[146] J. J. Gray, S. Moughon, C. Wang, O. Schueler-Furman, B. Kuhlman, C. A. Rohl, and D. Baker, "Protein-protein docking with simultaneous optimization of rigid-body displacement and side-chain conformations.," *Journal of Molecular Biology*, vol. 331, pp. 281–299, August 2003.

[147] K. Tai, "Conformational sampling for the impatient," *Biophysical Chemistry*, vol. 107, pp. 213–220, February 2004.

[148] S. Doniach and P. Eastman, "Protein dynamics simulations from nanoseconds to microseconds," *Current Opinion in Structural Biology*, vol. 9, pp. 157–163, April 1999.

[149] W. Shinoda and M. Mikami, "Self-guided molecular dynamics in the isothermal?isobaric ensemble," *Chemical Physics Letters*, vol. 335, pp. 265–272, February 2001.

[150] X. Wu and B. R. Brooks, "Self-guided Langevin Dynamics simulation method," *Chemical Physics Letters*, vol. 381, pp. 512–518, November 2003.

[151] S. C. Phillips, J. W. Essex, and C. M. Edge, “Digitally filtered molecular dynamics: The frequency specific control of molecular dynamics simulations,” *The Journal of Chemical Physics*, vol. 112, no. 6, pp. 2586–2597, 2000.

[152] A. Kolinski and J. Skolnick, “Monte Carlo simulations of protein folding. I. Lattice model and interaction scheme.,” *Proteins*, vol. 18, pp. 338–352, April 1994.

[153] Z. N. Gerek and S. B. Ozkan, “A flexible docking scheme to explore the binding selectivity of PDZ domains,” *Protein Science*, vol. 19, no. 5, pp. 914–928, 2010.

[154] A. May and M. Zacharias, “Protein-Ligand Docking Accounting for Receptor Side Chain and Global Flexibility in Normal Modes: Evaluation on Kinase Inhibitor Cross Docking,” *Journal of Medicinal Chemistry*, vol. 51, pp. 3499–3506, June 2008.

[155] F. Tama, O. Miyashita, and C. L. Brooks, “Normal mode based flexible fitting of high-resolution structure into low-resolution experimental data from cryo-EM.,” *J Struct Biol*, vol. 147, pp. 315–326, September 2004.

[156] Y. Levy, P. G. Wolynes, and J. N. Onuchic, “Protein topology determines binding mechanism,” *Proceedings of the National Academy of Sciences of the United States of America*, vol. 101, pp. 511–516, January 2004.

[157] J. Skolnick, L. Jaroszewski, A. Kolinski, and A. Godzik, “Derivation and testing of pair potentials for protein folding. When is the quasichemical approximation correct?,” *Protein Science*, vol. 6, pp. 676–688, March 1997.

[158] S. Miyazawa and R. Jernigan, “Residue-Residue Potentials with a Favorable Contact Pair Term and an Unfavorable High Packing Density Term for Simulation and Threading,” *Journal of Molecular Biology*, vol. 256, pp. 623–644, 1996.

[159] M. Zacharias, “Protein-protein docking with a reduced protein model accounting for side-chain flexibility.,” *Protein Science*, vol. 12, pp. 1271–1282, June 2003.

[160] A. Liwo, S. Ołdziej, C. Czaplewski, D. S. Kleinerman, P. Blood, and H. A. Scheraga, “Implementation of molecular dynamics and its extensions with the coarse-grained UNRES force field on massively parallel systems; towards millisecond-scale simulations of protein structure, dynamics, and thermodynamics.,” *Journal of Chemical Theory and Computation*, vol. 6, pp. 890–909, March 2010.

[161] A. Voegler Smith and C. K. Hall, “Alpha-helix formation: discontinuous molecular dynamics on an intermediate-resolution protein model,” *Proteins*, vol. 44, pp. 344–360, August 2001.

[162] A. Irbäck, F. Sjunnesson, and S. Wallin, “Three-helix-bundle protein in a Ramachandran model,” *Proceedings of the National Academy of Sciences of the United States of America*, vol. 97, pp. 13614–13618, December 2000.

[163] G. Wei, N. Mousseau, and P. Derreumaux, “Complex folding pathways in a simple β-hairpin,” *Proteins*, vol. 56, no. 3, pp. 464–474, 2004.

[164] C. Clementi, “Coarse-grained models of protein folding: toy models or predictive tools,” *Current Opinion in Structural Biology*, December 2007.

[165] C. Chen, R. Saxena, and G.-W. W. Wei, “A multiscale model for virus capsid dynamics.,” *International Journal of Biomedical Imaging*, vol. 2010, 2010.

[166] R. Das and D. Baker, “Macromolecular modeling with Rosetta.,” *Annual Review of Biochemistry*, vol. 77, no. 1, pp. 363–382, 2008.

[167] *Rosetta@home http://boinc.bakerlab.org/rosetta/ Last Accessed: 2010-09-30.*

[168] A. Zhmurov, R. I. Dima, Y. Kholodov, and V. Barsegov, “Sop-GPU: Accelerating biomolecular simulations in the centisecond timescale using graphics processors,” *Proteins*, vol. 78, pp. 2984–2999, July 2010.

[169] S. Miyazawa and R. Jernigan, “Self-consistent Estimation of Inter-residue Protein Contact Energies Based on an Equilibrium Mixture Approximation of Residues,” *Proteins: Structures, Function, and Genetics*, vol. 34, pp. 49–68, 1999.

[170] M. R. Betancourt and D. Thirumalai, “Pair potentials for protein folding: choice of reference states and sensitivity of predicted native states to variations in the interaction schemes,” *Protein Sci*, vol. 8, pp. 361–369, February 1999.

[171] C. Thachuk, A. Shmygelska, and H. H. Hoos, “A replica exchange Monte Carlo algorithm for protein folding in the HP model.,” *BMC bioinformatics*, vol. 8, p. 342, September 2007.

[172] M. Saito and M. Matsumoto, “SIMD-Oriented Fast Mersenne Twister: a 128-bit Pseudorandom Number Generator,” *Monte Carlo and Quasi-Monte Carlo Methods 2006*, pp. 607–622, 2006.

[173] D. Goldberg, “What every computer scientist should know about floating-point arithmetic,” *ACM Comput. Surv.*, vol. 23, pp. 5–48, March 1991.

[174] N. J. Higham, “The accuracy of floating point summation,” *SIAM J. Sci. Comput.*, vol. 14, no. 4, pp. 783–799, 1993.

[175] M. Tasche and H. Zeuner, *Handbook of Analytic Computational Methods in Applied Mathematics.* Chapman and Hall/CRC, 1 ed., June 2000.

[176] K. G. Troitzsch, “Validating Simulation Models,” in *Proceedings of 18th European Simulation Multiconference on Networked Simulation and Simulation Networks, SCS Publishing House*, pp. 265–270, 2004.

[177] K. A. Swanson, R. S. Kang, S. D. Stamenova, L. Hicke, and I. Radhakrishnan, “Solution structure of Vps27 UIM-ubiquitin complex important for endosomal sorting and receptor downregulation,” *The EMBO Journal*, vol. 22, pp. 4597–4606, September 2003.

[178] J. D. Foley, A. Van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics: Principles and Practice.* Addison-Wesley, 2nd ed., June 1990.

[179] J. E. Mebius, "Derivation of the Euler-Rodrigues formula for three-dimensional rotations from the general formula for four-dimensional rotations," *arXiv General Mathematics, arXiv:math/0701759v1*, Jan 2007.

[180] R. B. Best, "Personal Communications," 2008-2010.

[181] R. Ellis, "Macromolecular Crowding: Obvious But Underappreciated," *Trends in Biochemical Sciences*, vol. 26, pp. 597–604, October 2001.

[182] A. Minton, "Models for Excluded Volume Interaction between an Unfolded Protein and Rigid Macromolecular Cosolutes: Macromolecular Crowding and Protein Stability Revisited," *Biophysical Journal*, vol. 88, pp. 971–985, February 2005.

[183] H. X. Zhou, G. Rivas, and A. P. Minton, "Macromolecular Crowding and Confinement: Biochemical, Biophysical, and Potential Physiological Consequences," *Annual Review of Biophysics*, vol. 37, pp. 375–397, June 2008.

[184] J. Batra, K. Xu, and H.-X. Zhou, "Nonadditive effects of mixed crowding on protein stability," *Proteins: Structure, Function, and Bioinformatics*, vol. 77, pp. 133–138, October 2009.

[185] Y. C. Kim, R. B. Best, and J. Mittal, "Macromolecular crowding effects on protein–protein binding affinity and specificity," *The Journal of Chemical Physics*, vol. 133, no. 20, p. 205101, 2010.

[186] M. Hagan and D. Chandler, "Dynamic Pathways for Viral Capsid Assembly," *Biophysical Journal*, vol. 91, pp. 42–54, July 2006.

Appendix A

Supplementary Data

Table A.1: Amino Acid Parameters. *The van der Waals radius and electrostatic charge characteristics of each amino acid residue [12]. Electrostatic charge q refers to the elementary charge, approximately* 1.6×10^{-19} *coulombs.*

Amino Acid	**Abbreviation**	**van der Waals (Å)**	**Charge (q)**
Alanine	ALA	5.0	0
Argnine	ARG	6.6	1
Asparagine	ASN	5.7	-1
Aspartic acid	ASP	5.6	-1
Cysteine	CYS	5.5	0
Glutamic acid	GLU	5.9	-1
Glutamine	GLN	6.0	0
Glycine	GLY	4.5	0
Histine	HIS	6.1	0.5
Isoleucine	ILE	6.2	0
Leucine	LEU	6.2	0
Lysine	LYS	6.4	1
Methionine	MET	6.2	0
Phenylalanine	PHE	6.4	0
Proline	PRO	5.6	0
Serine	SER	5.2	0
Threonine	THR	5.6	0
Tryptophan	TRP	6.8	0
Tyrosine	TYR	6.5	0
Valine	VAL	5.9	0

Table A.2: Contact Energies in RT units. Values are attained from Miyazawa and Jernigan [158]

Amino Acid	Cys	Met	Phe	Ile	Leu	Val	Trp	Tyr	Ala	Gly	Thr	Ser	Asn	Gln	Asp	Glu	His	Arg	Lys	Pro
Cys	-5.44	-4.99	-5.80	-5.50	-5.83	-4.96	-4.95	-4.16	-3.57	-3.16	-3.11	-2.86	-2.59	-2.85	-2.41	-2.27	-3.60	-2.57	-1.95	-3.07
Met		-5.46	-6.56	-6.02	-6.41	-5.32	-5.55	-4.91	-3.94	-3.39	-3.51	-3.03	-2.95	-3.30	-2.57	-2.89	-3.98	-3.12	-2.48	-3.45
Phe			-7.26	-6.84	-7.28	-6.29	-6.16	-5.66	-4.81	-4.13	-4.28	-4.02	-3.75	-4.10	-3.48	-3.56	-4.77	-3.98	-3.36	-4.25
Ile				-6.54	-7.04	-6.05	-5.78	-5.25	-4.58	-3.78	-4.03	-3.52	-3.24	-3.67	-3.17	-3.27	-4.14	-3.63	-3.01	-3.76
Leu					-7.37	-6.48	-6.14	-5.67	-4.91	-4.16	-4.34	-3.92	-3.74	-4.04	-3.40	-3.59	-4.54	-4.03	-3.37	-4.20
Val						-5.52	-5.18	-4.62	-4.04	-3.38	-3.46	-3.05	-2.83	-3.07	-2.48	-2.67	-3.58	-3.07	-2.49	-3.32
Trp							-5.06	-4.66	-3.82	-3.42	-3.22	-2.99	-3.07	-3.11	-2.84	-2.99	-3.98	-3.41	-2.69	-3.73
Tyr								-4.17	-3.36	-3.01	-3.01	-2.78	-2.76	-2.97	-2.76	-2.79	-3.52	-3.16	-2.60	-3.19
Ala									-2.72	-2.31	-2.32	-2.01	-1.84	-1.89	-1.70	-1.51	-2.41	-1.83	-1.31	-2.03
Gly										-2.24	-2.08	-1.82	-1.74	-1.66	-1.59	-1.22	-2.15	-1.72	-1.15	-1.87
Thr											-2.12	-1.96	-1.88	-1.90	-1.80	-1.74	-2.42	-1.90	-1.31	-1.90
Ser												-1.67	-1.58	-1.49	-1.63	-1.48	-2.11	-1.62	-1.05	-1.57
Asn													-1.68	-1.71	-1.68	-1.51	-2.08	-1.64	-1.21	-1.53
Gln														-1.54	-1.46	-1.42	-1.98	-1.80	-1.29	-1.73
Asp															-1.21	-1.02	-2.32	-2.29	-1.68	-1.33
Glu																-0.91	-2.15	-2.27	-1.80	-1.26
His																	-3.05	-2.16	-1.35	-2.25
Arg																		-1.55	-0.59	-1.70
Lys																			-0.12	-0.97
Pro																				-1.75

Table A.3: List of Variables

Symbol	**Value**	**Quantity/Units**
K_b	$1.380650424 \times 10^{-23}$	Boltzmann Constant (J/K)
E_{charge}	$1.602176487 \times 10^{-19}$	Elementary Charge (Coulombs)
Å	10^{-10}	Angstrom (Metres)
D	80	Dialectric constant of water
N_A	$6.02214179 \times 10^{23}$	Avogadro's Constant
R_{gas}	8.314472	Gas Constant
λ	0.159	Scaling parameter
e_0	-2.27	Offset parameter (K_BT)
ξ	10	Debye screening length (Å)
	$(294 \times R_{gas} \times 4184)^{-1}$	Conversion from K_BT to $kcal.mol^{-1}$

Appendix B

Benchmarking Configuration

Table B.1: System Configuration *T*he hardware and software configuration is used for all benchmarks in Chapters 7

CPU	Intel Core2Duo 3 GHz (E8400)
Memory	4GB DDR 2 800, 2 channel configuration
Mainboard	Asus P5N-T (nforce 780i chip set)
OS	Ubuntu 9.10 64-bit
GPU	NVIDIA GTX 280 (Asus, hardware rev a1)
NVIDIA Driver	185.18
CUDA	2.21
GSL	1.12

Table B.2: Simulation Molecular Data *B*enchmarking simulation data is attained from one of three complexes.

UIM/Ub	Vsp27/Ubiquitin (100 residues, 24 and 76 respectively)
Cc/Ccp	Yeast cytochrome *c*/cytochrome *c* peroxidase (402 residues, 108 and 294 respectively)
2g33	Hepatitis B virus (284 residues per capsid piece)

Appendix C

Performance

Table C.1: CPU Simulation Benchmark Scalability

	CPU 1 Thread	CPU 2 Threads		CPU 4 Threads	
Residues	Time (s)	Time (s)	Speedup	Time (s)	Speedup
100	18	9	1.98×	9	1.92×
402	309	158	1.95×	156	1.98×
568	786	411	1.91×	400	1.97×
1136	4702	2387	1.97×	2374	1.98×
1704	11735	5968	1.97×	5943	1.97×
2272	21913	11099	1.97×	11070	1.98×
3408	51725	26099	1.98×	26077	1.98×
4544	94054	47903	1.96×	47750	1.97×
5680	148695	75670	1.97×	75466	1.97×
6816	216140	110090	1.96×	109643	1.97×
7668	275042	139773	1.97×	138966	1.98×

Table C.2: Synchronous GPU Simulation Benchmark Performance

	CPU	GPU Sync. 1 Thread		GPU Sync. 2 Threads		GPU Sync. 4 Threads	
Residues	Time (s)	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup
100	18	2.0	8.9×	7.7	2.3×	3.0	5.9×
402	309	2.8	109.4×	17.2	17.9×	4.8	63.8×
568	786	3.6	217.4×	20.2	38.9×	5.1	153.9×
1136	4702	11.1	424.7×	33.8	139.1×	16.1	291.8×
1704	11735	23.7	495.5×	41.2	284.8×	27.0	434.8×
2272	21913	42.5	516.0×	54.3	403.4×	45.6	480.1×
3408	51725	90.9	569.1×	84.9	609.4×	87.4	591.7×
4544	94054	158.7	592.6×	130.0	723.2×	130.3	721.9×
5680	148695	256.2	580.3×	200.7	740.8×	201.0	739.9×
6816	216140	355.6	607.8×	263.0	822.0×	265.0	815.6×
7668	275042	455.6	603.7×	332.1	828.1×	332.5	827.2×

Table C.3: Asynchronous GPU Simulation Benchmark Performance
Times are quoted with standard deviations and are averages for all streams for a particular number of pthreads.

	CPU	**GPU Async. 1 Thread**		**GPU Async. 2 Threads**	
Residues	Time (s)	Time (s)	Speedup	Time (s)	Speedup
100	18	2147 (±33)	8.3×	12024 (±330)	1.5×
402	309	2260 (±31)	136.6×	12296 (±58)	25.1×
568	786	2727 (±31)	288.4×	12504 (±130)	62.9×
1136	4702	6496 (±209)	724.5×	16611 (±82)	283.1×
1704	11735	10537 (±155)	1113.9×	20203 (±110)	580.9×
2272	21913	17676 (±76)	1239.8×	27525 (±294)	796.1×
3408	51725	37126 (±120)	1393.2×	46754 (±362)	1106.3×
4544	94054	64312 (±184)	1462.5×	73657 (±456)	1276.9×
5680	148695	103356 (±436)	1438.7×	112728 (±654)	1319.1×
6816	216140	139113 (±455)	1553.7×	147872 (±670)	1461.7×
7668	275042	191747 (±544)	1434.4×	200376 (±863)	1372.6×