Parallel algorithms for querying spatial properties in the Protein Data Bank

Joshua Selvan

A research report submitted to the Faculty of Engineering and the Built Environment, University of the Witwatersrand, Johannesburg, in partial fulfilment of the requirements for the degree of Master of Science in Engineering.

Johannesburg, December 2019

Declaration

I declare that this research report is my own, unaided work, except where otherwise acknowledged. It is being submitted for the degree of Master of Science in Engineering to the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination to any other university.

Signed this ____ day of _____ 20__

_____

Joshua Selvan

Contents

Glossary

absolute run time The wall-clock time which a code takes to run from start to finish..

amino acid Molecules which make up proteins.

binary tree A simple tree structure where each data node can have up to two child nodes.

brute force A programming approach where problems are solved by dedicating excessive computing resources to it.

C++ A programming language.

CUDA A programming API for NVIDIA GPUs.

distributed computing Programs which run over multiple separate machines or processing nodes.

ESBTL Easy Structural Biology Template Library.

GPGPU A "General Purpose Graphical Processing Unit", a newer GPU which can accept custom code to run in its shader processors.

GPU Graphical Processing Unit: A specialised piece of processing hardware usually used for displaying graphics.

hashed index A key based structure used for fast, direct look up of specified elements in data storage objects such as vectors or tables..

kd-tree A k-dimensional spatial data structure tree in which the data points placed into the tree are sorted by alternating dimensions at each level of the tree.

Moore's law A prediction about the speed at which integrated circuits on computer chips improve.

MPI A "Message Passing Interface" standard for distributed memory parallel processing.

multi-threading Code which has purposefully been designed to run multiple concurrent processes at once.

openMP A programming API for multi-threaded processing in shared memory.

parallelization The process of converting a sequential piece of code to one which makes use of parallel resources.

PDB "Protein data bank" a file structure used by the world wide protein data bank to store protein structures.

pre-processing To format data before it is used in an algorithm.

Protein A biomolecule consisting of a chain of amino acid residues.

Spatial data structures Data storage structures which divide data into subsets based on their spatial layout.

thread A contained process being run by a processing unit.

Abstract

Searching large protein databases for proteins with certain structural properties is expensive. This research explored the use of GPGPUs (General Purpose Graphical Processing Units) in speeding up such structural queries. Brute force and kd-tree spatial data structure algorithms were compared and benchmarked against non-GPU parallel algorithms to assess the effectiveness of using GPGPUs.

This was done with the aim of increasing the speed at which queries against large protein databases can be completed to help mitigate the effect of increasing data set sizes of current protein databases [57].

A set of parallel variations of range search algorithms were developed and implemented in the GPU programming language CUDA and their performances times in completing batch range search jobs were compared against other parallel approach types such as multi-threading and message passing to see if the GPU approaches completed notably faster or slower than more traditional parallelised approaches.

The results showed GPGPUs can construct kd-trees far faster than other parallelised implementations can achieve and that in most scenarios (excluding specific cases such as very low or zero result searches) the GPGPU approaches either matched or performed far better than the other parallelised approaches.

While comparing different GPU algorithms, the complex GPU based kd-tree algorithm performed similarly to a simple GPU brute force range search. This highlighted the benefits of writing code which made the most of the GPU's parallel architecture as opposed to modifying efficient (recursive) algorithms to adequately fit into those same GPU architectures. This implied that even though spatial data structures are effective ways of dealing with protein data, there are better returns on effort in writing code specifically for the GPU's inherently parallel architecture for initiatives which require algorithms to be developed from scratch.

Chapter 1

Introduction

1.1 Overview and motivation

Proteins are large biomolecules which are involved in or are solely responsible for most of the processes which occur within cells (such as DNA replication and catalysing metabolic reactions). Each protein is composed of long amino acid chains. The biophysical properties of the amino acids determine the three dimensional (3D) structure of the chain and it is this shape of the protein which is a key factor in the behaviour and function of the protein. Proteins often occur in complexes with other proteins, and neighbouring proteins will affect a protein's shape.

There are many different types of applications which use or determine the physical structure of proteins. Practical examples include areas such as drug design where finding a protein segment with a specific shape might be useful [14], predicting the behaviour of the secondary and tertiary structures of proteins [32], or predicting the actual functions of said proteins [15].

This research thus considers the following specific problem: given a large database of protein structures, find all proteins in that database that have amino acids of a specific type with atoms that are within a required range of each other within protein structure. For example, a very simple query might be to find all proteins in the database where there is an oxygen atom on an alanine amino acid that is within a certain distance range to a sulphur atom on a cysteine amino acid within the same protein chain.

This task of locating amino acids that are close to each other in 3D space but may be far from each other in the protein chain is useful because protein chains can be

contorted into a 3D shapes by the way the amino acid molecules within the protein naturally bend as well as outside forces such as other nearby protein structures. By locating proteins with certain molecules within a required distance of each other (even though those atoms may be far apart on the primary sequence) one is able to search for or confirm that certain amino acid combinations exhibit forces on each other or check for implied connective forces within protein records which contain multiple separate proteins. When one is able to confirm sites of interaction within or between proteins, one can then move onto more complex queries such as the implications of those interactions. For example, in *Cerutti, et al.* [14] molecules were sought which would interact with other molecules under certain conditions. A valid experiment would be to search 100000 known protein structures for amino acid pairs which are close to each other in 3D space despite being far apart on the primary structure – molecule pairs meeting this condition would be viable for further study to see if they were amenable and effective for planned engineering efforts.

As to the size of the protein data sets being searched, the statistics page of the World Wide Protein Data Bank [57] states that the number of structures (the main storage format for protein data) that are deposited each year almost doubled between 2008 (7000) and 2017 (13 049) with a current total of 149 536 stored protein structures.

Given this situation there were two general approaches to improving performance. One could either seek to utilise information inherent to the protein data being parsed (such as indexing the spatial positioning of each protein's molecules) to more quickly locate sought relationships, or one could attempt to utilise parallel programming architectures to try to scale the implemented algorithms to cope with larger data sets in lower time frames. The approach taken in this research was to try to utilise both approaches at the same time.

1.2 Research Objectives

The primary objective of this research was to test the performance characteristics of GPUs at performing range searches against openMP and MPI on large protein databases.

The way this was implemented was to provide a scalable approach to executing amino acid range searches (a technique for finding spatial data points within a required distance of each other) by optimising both a brute force search and a spatial data structure known as a kd-tree to be run in the parallel environment of a GPU.

Spatial data structures are indexing structures which group data points together based on the spatial area they inhabit with regards to other data points within the same set [61]. This emphasis on position provides an efficient way to identify proximities between data points in 3D spaces – as the atoms in a protein are small enough that they can be represented as points in space for the purpose of these searches. The challenge for this research however came in finding a good way to implement the construction and searching of a popular spatial structure known as a kd-tree in a parallel GPU environment as aspects of the kd-tree approach are naturally taken care of in sequential environments via recursion, while those aspects have to be intentionally planned around in the GPU's non-sequential architecture.

1.3 Overview of approach

These challenges resulted in both a GPU based kd-tree construction algorithm and a kd-tree range search algorithm being designed which reworked the recursive relationships of sequential kd-tree algorithms into formats which were more compatible with GPU hardware.

For the GPU kd-tree construction algorithm, a complex pre-sorting algorithm was developed which allowed for all the spatial ordering data of each spatial dimension to be generated in a single sorting run which could then be re-used without having to re-sort the remaining data points at each level of the kd-tree. Additionally this construction method resulted in the remaining data subsets for each position in each given level of the kd-tree being distinct which allowed for each consecutive level of the kd-tree to be computed in parallel without any race conditions or contested resources. This proved to be a very efficient algorithm for constructing kd-trees and performed much better than the comparison algorithms which will be detailed a bit later.

For the GPU kd-tree range search algorithm, an approach was developed which started with a single process being run from the root of the kd-tree, but then instead of searching each additional viable path down the kd-tree encountered via recursion, a backlog of required kd-tree descent paths were stored in a list and at regular intervals all backlogged search paths were added to the kd-tree paths being run in parallel – enabling a kd-tree range search process which is recursive in sequential architectures to be run in lockstep (a term used when multiple processors need to execute the exact same command simultaneously) in the GPU's lockstep centric environment. While this GPU based range search approach did perform

competitively compared to other parallel approaches, the GPU kd-tree range search performed similarly to a normal GPU brute force range search in most cases - and GPU brute force approaches require less memory space to run and are far simpler to set up.

The pursuit of scalable protein data bank searching is not done in a vacuum however. There have been many papers on scalable (parallel) searching techniques and quite a few implementations are provided for free usage online [67, 59, 6]. Thus, to properly display the gains of utilising GPU based parallelism, the GPU inclusive algorithms developed were compared against more traditional forms of parallelization – namely multi-threading and distributed computing. To realise these comparisons, brute force and kd-tree range search algorithms were implemented for execution with multi-threading, for execution across a distributed computation environment and for execution with a GPU environment. The GPU oriented algorithms were then run against the multi-threaded and distributed computing algorithms with different load sizes and search requirements to ascertain how the GPU implementation's performance differed from the more traditional parallel approaches.

These trials were done with the use of a dedicated server in the Wits Core Research Cluster which featured an NVIDIA GK110GL [Tesla K20Xm] GPU while utilizing a data set of 100 000 PDB files from the worldwide protein databank (which amounted to over 76 gigabytes of data). What the results showed was that GPU approaches were usually comparable or superior to shared memory multi-threaded approaches and distributed computing approaches in high compute tasks. That said, the processing resources upon GPUs are finite and extra resources (such as extra processors or memory chips) cannot be added into existing GPU models – so while the GPU provided substantial gains to processing speed in high compute tasks, the more distributed processing nodes one added to a compute cluster, the closer that traditional approach came to equalizing with a single high cost GPU setup - making distributed computing a more scalable approach for extremely large datasets unless one were to obtain multiple GPUs to spread out across a similarly distributed network at high monetary and effort cost.

1.4 Structure of Report

The rest of this research report is dedicated to explaining the various knowledge fields needed for background, the objective method by which the experiment was run, the specifics of physical hardware and developed algorithms used, and finally

the results and conclusions drawn from the parallelized algorithm comparisons.

In the background section an introduction will be given to proteins, what they are and how they behave. Following that a brief section covers the expansion rate of data and the expected deceleration of sequential computing speed improvements. An overview of several types of parallelization will be given, a brief section on mathematically comparing parallel speedup and then a mathematical section on tree structures and kd-trees. In the subsequent objectives chapter the experiments to be run will be stated and the comparison criteria used to judge the results will be defined. The fourth chapter will list all the resources utilized and constructed to execute this research, while the final two chapters will list the comparison results, provide some graphs for clarity and elaborate on which scenarios the GPU oriented algorithms provided the best absolute processing times for the biomolecular range searches.

Chapter 2

Background

2.1 Overview

The research in this report explores the use of GPUs in speeding up the runtime of range search queries across large protein databases. To explain the basis of this research, it is necessary to outline the nature of the proteins being searched, the mathematical algorithms used in range searches, the relationship between the rate at which available computing power increases and data set size, and an overview of some ways in which one can parallelize algorithms with available parallelization platforms.

To address these an introduction will be given into what proteins are, how they interact, and a description of what primary, secondary and tertiary structure refers to.

Following this a section is dedicated to explaining tree structures and their use in enabling spatial searches – with an emphasis on kd-trees, their expected search times as well as examples of other spatial algorithm endeavours in the biological field.

The third section will deal with the rate at which sequential computing devices have improved over the years, and a brief overview of a prediction known as Moor's law.

The fourth section will look into the terminology of comparing parallel systems, while the fifth and final section provides a brief history and an explanation of how several parallelization platforms work - including openMP for multi-threading, MPI for shared memory multi-nodal parallel processing, and CUDA for GPU processing. In the GPU section some additional detail will be given as to the sorts of algorithmic

issues which can cause bottlenecks in its parallel architecture.

2.2 Proteins

2.2.1 The roles and composition of proteins in cells

Proteins are a type of molecule that are either involved in or responsible for most of the functions within living cells [17]. Proteins can act as enzymes speeding up the rate at which chemical reactions occur within cells; they are the constructs responsible for the transport of energy throughout cells, a means of delivering the raw material of which cells are built, and they underlie sensors and cells' ability to transport information [36].

Proteins are constructed from smaller molecules called amino acids and the properties of each protein are determined not only by the sequence of amino acids which compose the protein but also the shape that the proteins take on at any given time [49]. Proteins bend into these native 3-dimensional shapes automatically in an expeditious and reproducible manner (a behaviour known as protein folding) and are usually biologically functional.

While non-bonded interactions always occur between molecules, strong non-bonded interactions only occur between certain configurations of amino acids – or occasionally amino acid combinations which appear similar to the correct amino acid combinations which mimic the interactions of the interactive configurations [19]. Examples of such configurations which allow proteins to interact include weak hydrogen bonds, ionic interactions, disulphide bridges, aromatic interactions and hydrophobic interactions [67]. This means that the shape a given protein folds into is dictated by the positions of the amino acids which compose the protein and this resulting shape in turn dictates how the protein will be able to interact with other molecules based on its physical layout or spatial orientation.

2.2.2 Describing protein structures in four levels

The primary structure

Proteins are constructed from organic compounds known as amino acids. All amino acids are made up of an amine (-NH2) and carboxyl (-COOH) functional group as

well as a side chain (R group) which is the distinguishing feature of each type of amino acid. There are about 500 naturally occurring amino acids but only 20 serve as the building blocks of protein.

Each amino acid is capable of forming up to two peptide bonds – which is a covalent link which can occur between two amino acids with the elimination of a water molecule. This allows the carboxyl group of one amino acid to bond to the amino group of the next. This limit of two covalent bonds is why amino acids form sequential polypeptide chains – with the amino acids which compose the chains being called amino acid residues due to the water molecule lost between each pair (see figure 2.2.2). These sequential polypeptide chains are the protein's primary structure [7].

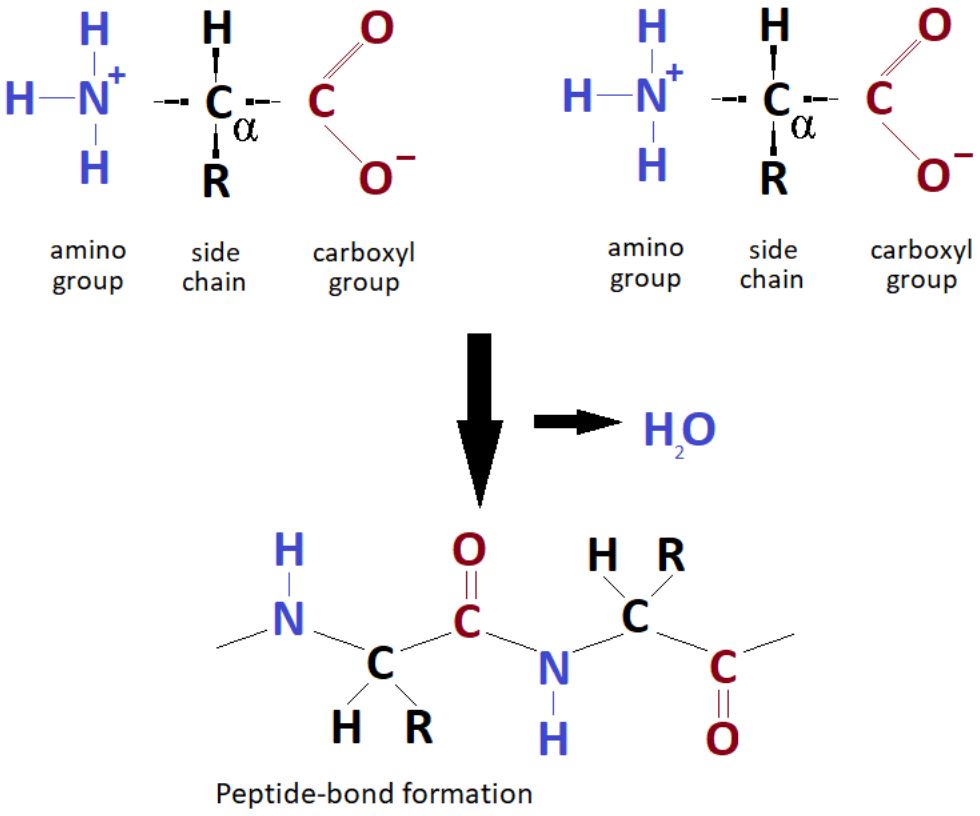


Figure 2.1: With the loss of two hydrogen atoms from the amino group and an oxygen atom from the carboxyl group, the C of the carboxyl group forms a bond with the N of the amino group

The secondary structure

While polypeptide chains are just a sequence of amino acid residues, this sequence does not naturally form a straight line. The bonds which hold the three component groups of amino acid residues together are sources of torsion which cause the polypeptide chains to fold in predictable manners when viewed in small segments.

While these sources of torsion result in predictable shapes locally, they still leave the polypeptide chains with degrees of freedom. As an example, while an NH group joined to a side chain may impose a specific bondage angle between the two groups, one still lands up with relatively free torsional rotation so long as those groups maintain the required angle of bondage – which means that long polypeptide chains can be rotated into countless valid configurations with the constraints of their bonds still met.

The shapes which secondary structures conform to without any outside forces are α helices, β sheets, β turns and ω loops [7].

Figure 2.2: Above is depicted two examples of resulting secondary structure shapes which polypeptide chains can be bent into by torsional forces

The tertiary structure

The tertiary structure of a polypeptide chain refers to the usually compact, three dimensionally folded arrangement that the polypeptide chain adopts under physiological conditions. Segments of the chain may be α helices or β strands or a number of less regular conformations such as turns or loops between secondary-structure elements that allow these elements to pack tightly against each other [7].

Sometimes proteins can naturally occur in multiple tertiary structures, and at other times secondary molecules within cells will force polypeptide chains into different shape configurations which cause the containing protein to behave differently [36].

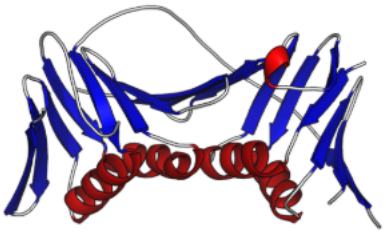


Figure 2.3: Above is depicted a tertiary structure consisting of β sheets and α helices. As opposed to the individual β sheets and α helices shown in Figure 2.2.2, the secondary structures in this image have been compacted and contorted by secondary forces.

Quaternary structure

While not required, there are instances where a protein will be composed of multiple polypeptide chains. Examples of such proteins include haemoglobin, DNA polymerase, and ion channels. In these instances the protein will be said to have a quaternary structure.

The polypeptide chains which compose a quaternary structure are referred to as subunits – and all the subunits of a quaternary structure can either be identical polypeptide chains or there can be different types of subunit chains.

Of importance is that while a polypeptide chain's secondary and tertiary structure can be unaffected by a protein's quaternary structure, there are many instances where a polypeptide chain's secondary and tertiary structure will change when part of a quaternary structure or may only acquire those structures once joined to a quaternary structure [7].

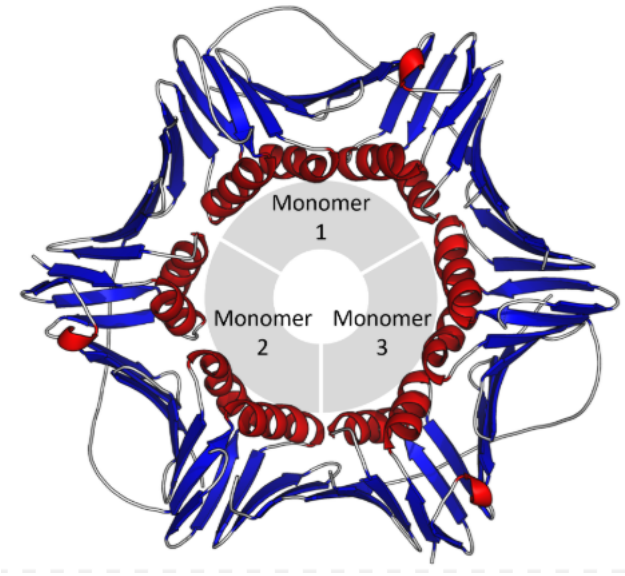


Figure 2.4: A quaternary protein structure

Drawing relationships from highly dynamic structures

Analysing how a protein behaves in a single configuration is a simplification of the protein's behaviour. Not only does one wish to know the various shapes that proteins can take on, but also which amino acids come close to each other when the protein strands start to loop back over themselves – as the proximity of different amino acids (even when not directly joined) can cause the protein to react differently to other molecules within cells.

While basing amino acid interactions on positions alone remains a simplification, identifying amino acid proximities is often an important base component of more complex bio-molecular analysis techniques.

To compare different algorithmic approaches for processing proteins, we compared the time for calculating amino-acid distances.

2.2.3 Examples of bio-molecular research featuring spatial data

Aromatic bond interactions

In organic chemistry, aromaticity is used to describe flat, cyclic molecule rings consisting of resonance bonds which exhibit more stability than other geometric or

connective arrangements which can be achieved with the same set of atoms. Aromatic molecules thus have low reactivity and so do not break apart easily to react with other substances.

The paper by ***Chourasia et al. (2011)*** [15] focuses on the amino acids phenylalanine, tyrosine, tryptophan and histidine which contain aromatic rings and describes interactions between these aromatic rings.

This was done by using the X-ray crystal structures of proteins from PDB to map the aromatic-aromatic ($\Pi-\Pi$) network of proteins in aromatic rings and then investigate how the separate aromatic rings were connected to each other.

Amongst other noted relationships was the trend that the angle between the planes of proximal aromatic rings would result in different types of interactions – with greater angles (resulting in the centroid of the two aromatic rings being above 5Å) resulting in C-H$\cdots\Pi$ interactions being more prevalent while ($\Pi-\Pi$) interactions were more prevalent at lesser centroid distances.

Higher order residue interactions

The inside derivation of a three-dimensional structure of a protein from its primary sequence is controlled by a set of principles referred to as the folding code. The principles which dictate folding code however are both complex and largely unknown. One proven principle which has been shown to play a crucial role in attaining stable conformations in protein structures is the concept of higher order residue interactions (*HORI*) [65].

HORI is based on the concept that if one approximates amino acid residues to spheres centred on their location, then it is not possible for more than four closely packed spheres to be in mutual contact with each other at the same time – meaning that *HORI* interactions can only occur as pair wise, triplet or quadruplet interactions. The concept of *HORI* interactions has been successfully employed in structure analysis and fold recognition by different groups [29, 72], while earlier work also showed that the higher order interactions can be used to improve accuracy of fold recognition and generic structure analysis [62, 41].

In ***Sundaramurthy et al.*** [65], a web server is described which facilitates the searching of 3 categories of *HORI* relationships (residue number based computation, complete structure based computation and residue-type based computation) within

user specified distance cut-off ranges and amino acid residue structure types (C-α or C-β)

Location of required bond types for drug design

Cerutti et al. [14] dealt with the development of potential viral attachment inhibitors and immunotherapeutic agents to hinder the interactions that the HIV-1 virus utilises to enter cells.

The research hypothesised that proteins with specific disulphide bonds would more readily interact with HIV-1 than the cells that HIV-1 seeks to enter – and that after such bondings take place, the ability of the pre-bonded HIV-1 to enter such cells would be greatly inhibited. This demonstrates the relevance of being able to identify sought amino acid residue properties for commercial and medical relevant purposes.

Generalised location and identification of interaction sites

Robillard [59] describes a web available service known as SpeeDB which allows for the rapid searching of over 80 000 protein structures (the entirety of the worldwide protein database at the time of publication) for relationships such as the aromatic and *HORI* type interactions mentioned previously (or other relationships which can be expressed by proximities of specific amino acid residue proximities) in a matter of seconds.

The benchmark tests, run on a moderately complex Sulphur–Aromatic query, showed that the web service was able to provide linear speedup with the number of compute cores used. This was achieved via a range of technical innovations such as permanent in-memory spatial data structures for all the PDB files available for querying (which allowed PDB structures to be searched in sublinear time), highly optimised memory layout, as well as a multi-threaded query engine.

Molecular Dynamics

Molecular dynamics (*MD*) is a field of algorithms which models a variety of molecular properties for solids, liquids and molecules [1, 4]. For molecules this works by treating each atom in the constructed simulations as points of mass and then running Newton's equations on each atom to calculate their energy coefficients in very

small time steps [56]. This allows for an approximation of how each atom moves over time and when summed up, one can generate a general energy function for the entire molecule over time.

Molecular dynamics simulations are not compute intensive for individual atoms, rather the intensiveness of *MD* simulations comes from the number of atoms being processed and the number of time steps that calculations are made at to make molecular level simulations accurate – which often means that simulations need to be calculated in femtosecond intervals, which means hundreds of thousands of time steps need be calculated to generate accurate behaviour for a few picoseconds of real time [10].

As a result of the inherently parallel nature of *MD* simulations (as in a number of simple, repetitive calculations run over hundreds to hundreds of thousands of point masses repetitively [56]), *MD* simulations are prime candidates for parallelised computing, and extensive initiatives have been undertaken to utilise them in parallel systems [2, 58, 64].

An example of an *MD* program is CHARMM [11]. CHARMM is a code set that calculates a set of alchemical/empirical energy functions. These functions, while accurate, are not based on the physics of the individual atoms but rather observed approximate free energies associated with atoms or atom groups in certain orientations, distances and angles. The three operations which CHARMM calculates with this information are minimization (wherein CHARMM calculates the movements a system would make to minimize its energy), Dynamics (which produces a trajectory of a system), and Normal mode analysis which provides an orthonormal basis for the harmonics vibrations of a system about a particular configuration. On the topic of spatial input, CHARMM uses a list of non-bonded pair interactions to reduce the number of calculations which need to be redone at each timeframe. Instead of calculating the energy between every pair of atoms in the system, CHARMM instead keeps a list of all the relevant pairs which are within a certain distance of each other and while the positions of atoms in these systems usually don't radically change in relation to each other, the list does need to be periodically updated [11].

Two examples of GPU oriented MD projects will be given at the end of the parallel platforms section after some key GPU terminology has been provided.

2.2.4 Protein Data Bank (PDB) files

The Protein Data Bank format (PDB) was created in 1970 to provide a standardized representation for macromolecular structure data derived from X-ray diffraction and NMR studies [71] and is currently the most commonly used format for storing protein data.

PDB files store atomic coordinates, crystallographic structure factors and NMR experimental data. Aside from the coordinates, each file also includes the names of molecules, primary and secondary structure information, sequence database references (where appropriate), ligand and biological assembly information, details about data collection and structure solution and bibliographic citations [71]. Within PDB files this data is split up under the headers of:

- Introduction
- Title Section
- Primary Structure Section
- Heterogen Section
- Secondary Structure Section
- Connectivity Annotation Section
- Miscellaneous Feature Section
- Crystallographic and Coordinate Transformation Section
- Coordinate Section
- Connectivity Section
- Bookkeeping Section

For the purposes of categorising amino acid chains, only the data contained within the "Coordinate" section of PDB files are required. The PDB "Coordinate" section lists the atoms within the PDB's proteins in the order they appear within the protein's amino acid sequences. Besides giving the order that atoms within the amino acid chain appear, each atom entry within the "Coordinate" section also includes:

- The atom serial number

- Atom name
- Residue name
- Chain identifier
- Residue sequence number
- X coordinate in Ångstroms
- Y coordinate in Ångstroms
- Z coordinate in Ångstroms
- Temperature factor
- Element symbol
- Charge on the atom

2.3 Spatial data structures

2.3.1 Spatial indexing

When data contains clearly defined characteristics, data structures can be used to organize and manage that data by indexing those characteristics. The use of spatial data to order data structures is known as spatial indexing [61]. Spatial indexing techniques sort data with respect to the overlapping spaces that the data occupies by means of recursive decomposition into continually smaller spatial subsets.

Spatial indexing is best utilized on data types which can be geometrically represented or geometrically decomposed [61]. Thus any data sets that have attributes which can be represented as points, lines, polygons, regions or volumes are favourable candidates for use with spatial data structures. Straight-forward examples of such datasets include Geographic Information Systems (GIS) [13]. The locations of landmarks in cities, the positions and connections within road systems, or even topographical layouts, have geometric relationships which can be exploited by hierarchical structures for faster lookups. Other examples of data sets which work well with hierarchical structures include linear networks, computer graphics, solid modelling and the storage and representation of protein chains – as protein chains exhibit complex behaviours which can be understood by studying their physical 3D layout [61]. Using data structures to store these types of information in compact forms can often

save space as well as reduce the time taken to perform searches and modifications to the data in question [8].

While spatial data structures outperform unstructured databases with small databases, the worth of such structured methods become more apparent when dealing with databases of massive size. Without structure, using brute force to search for collisions in a 2D data sets could take (in the worst case) $O(n^2)$ time to complete. Such quadratic search times are expensive in data sets with thousands of entries, in truly massive databases (take the complete records of a country-wide retail store for example) the quadratic cost of such a simple search is inhibiting. With the use of simple tree structures, data structures can reduce the worst case of such 2D databases from $O(n^2)$ down to $O(n \log(n))$ or even $O(n)$

2.3.2 Binary trees

Binary trees are tree data structures in which each node has at most 2 child nodes – one known as the left child node and the other as the right child node. Binary trees always start at a single node known as the root node and nodes which have no child nodes are known as leaf nodes. Sub-trees refer to smaller trees within the main binary tree which are composed of all nodes joined to by a non-root node. In binary trees, all sub-trees will also be binary trees – although this is a trait that does not hold true for all tree structures [61].

When used to store data, binary trees will recursively split the remaining data load into their two child nodes, with each child node forming the start of a sub-tree which will then split the remaining data load via the same principle. While data is occasionally stored within every node within a binary tree, most often the actual data points that the binary tree stores will only be kept in the tree's leaf nodes. The tree's stored data is organized by splitting the data load via some median value at every non-leaf node in the tree. When searching a binary tree for a sought value, at each node of the binary tree the user will descend to the child node whose value range could still contain the value which the user seeks – until the user either finds its sought value in a leaf node or discovers that the tree does not contain the sought value

As an example, consider a binary tree storing the names and heights of all the students in a class. The root node may split the data so that all the students who are less than 1.7 meters tall are stored in the left sub-tree while all students who

are taller than 1.7 metres would be stored in the right sub-tree. Looking at the left sub-tree, all students who are shorter than 1.3 metres may be stored in the left sub-tree's left sub-tree while all students who are taller than 1.3 metres but shorter than 1.7 metres would be stored in the left sub-tree's right sub-tree, etc. Whether the names and heights of individual students are stored only in the leaf nodes of the binary tree or within the interior nodes of the tree as well is a trivial choice of implementation.

2.3.3 Kd-trees

kd-trees share many similarities with binary trees. Each node in a kd-tree has two child nodes and like binary trees, kd-trees split data loads at every level of nodes based on a dividing attribute value [9]. The order in which the nodes of kd-trees are placed however is designed to deal with data loads which need to be sorted by many dimensions of attributes.

Where a binary tree sorts data based on different markers of a single attribute only, kd-trees alternate between different attributes at every level of the tree. This is to say that if you were storing the coordinates of points in 3D space within a kd-tree, the root node may split the data points into sub-trees based on an x-axis coordinate while the next layer of nodes would split the data points into sub-trees based on a y-axis coordinate, followed by a z-axis coordinate, followed again by an x-axis coordinate, and so on. This approach effectively splits up the dimensional space of the data being processed into recursively smaller d-dimensional cells. Conceptually this means that every time a user descends a branch in a kd-tree, the user has navigated to a smaller sub-space within the total space of the entire data set – an approach which can be efficiently manipulated for discerning distances between different data points.

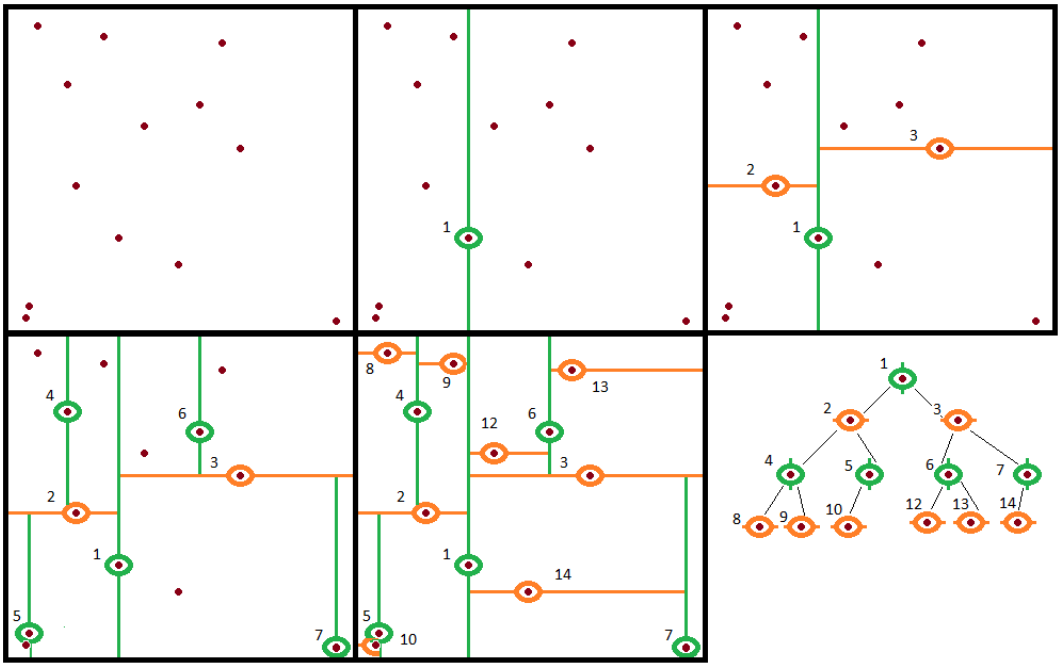


Figure 2.5: In the above picture one can see 13 points being allocated positions in a kd-tree which alternates between placing with respect to the middle most x-coordinate (green lines and circles) and y-coordinate (orange lines and circles). As can be seen from the 3rd box and forward, each consecutive middle most point is chosen from reduced subsets of space divided up by previous inserts into the tree. Also note that in the final level of the tree, there is no point allocated to the 11 slot in the tree – depending on how you split even subsets of points at each level of the tree, empty positions in the final level of the tree can appear in different places.

The reason that kd-trees are popular when dealing with multi-dimensional attribute trees is because their width expands slower than approaches such as quad-trees (with the width of kd-trees doubling every level as opposed to the quadrupling width of quad trees) – this said, kd-trees do lead to much deeper trees and unlike quad trees where each set of 4 child nodes are set, the order chosen to cycle through dimensions in a kd-tree affects the data layout of the entire tree.

While the kd-tree approach causes fan-out problems, and allows for any number of attribute splits at each level of the kd-tree, the structure of kd-trees cause their own problems. As kd-trees sort through attribute space sequentially, the sequence in which different attributes are searched for affects the layout of data within the kd-tree – as opposed to quad-trees where the ordering of attributes is of no consequence as all attributes are dealt with simultaneously. Additionally, kd-trees tend to lead to far deeper trees than quad-trees due to the separated sequences in which attributes are tested.

2.3.4 Range searching with kd-trees

Range searching refers to the process of finding data points within a certain proximity of a selected data point or type of data point. Range searches can be performed across any number of dimensions and while one could search for proximities between points in the x, y or x, y and z dimensions, the dimensions being looked at could be anything – for example *salary*, *age*, *years in tertiary institutions* and *total career length*, are a valid set of dimensions to perform a range search across.

Empirically, Exact match range searches (where every sought proximity between two data point types) in kd-trees were shown to have an average running time of $O(\log(n))$ [9] while the worst case has been worked out to be $O(n^{(1-\frac{1}{k})})$ [42].

2.3.5 Other types of spatial structures

There are several other types of spatial structures used for parallel processing. In this research the decision was taken to use kd-trees due to successes with this structure in previous work in the faculty [59]. Three other types of spatial structures types are listed here for completeness:

Quad-trees

Quad-trees more closely resemble a two dimensional binary tree than kd-trees – as quad-trees implement the same sorting criteria at each level of the tree where as kd-trees alternate between sorting criteria. Where binary trees confine each node in the tree to having only two child nodes (good for splitting data with a single dimension), each node in a quad-trees has 4 child nodes – meaning that a quad-tree representing two dimensional data will split the data into all four dimensional quadrants at each level of the tree. Quad-trees thus fan out much faster than kd-trees but are not as deep and unlike kd-trees, the fact that one does not need to keep track of which dimension is being sorted at each level of the tree is also a plus. The three dimensional version of a quad-tree is known as an octree which has eight child nodes per node.

R-trees

An R-tree is a tree structure which works by bounding sets of data points into the tightest fitting bounding box which can hold them and then splitting the bounding box into two bounding boxes if the selected box receives more data points than the boxes are allowed to hold – which each become child nodes of the node which held the original bounding box (but no longer holds any data points after they are divided and transferred into the two child nodes. In R-trees, the leaf nodes contain data points and references to which bounding box they belong to [8].

Searching R-trees works by identifying the bounding boxes which meet the searches criteria and then analysing the entries within those bounding boxes for relevant data points – as opposed to the previous tree structures listed which search through data points (in their own nodes) individually one at a time.

K-D-B-tree

A K-D-B-tree uses the dimensional splitting criteria of the kd-tree but stores the data points of the data set the tree represents in buckets like the R-tree in the above subsection [60].

Grid files

Grid files are a spatial lookup structure which works by representing k dimensional data records as points within a k dimensional grid which is then compressed down into small buckets of non-empty entries – with each data point in the buckets having a reference address of the actual data entry in the database the grid file is tied to.

Grid files are formed by creating a single bucket which can hold n entries and when that bucket overflows the amount of values it can hold, the bucket is split into two buckets divided by a median value of one of their attributes. When inserting further entries, the entries will be placed in the bucket (of the two available) which meets its dimensional criteria. When either of these two buckets overflow, both buckets will be divided in half again over a median value of a different attribute of the data – increasing the grid from two buckets next to each other to a square of buckets, ect. As this insertion method continues buckets will be created which represent smaller sections of k dimensional space with 1 to n entries represented in each bucket.

2.4 Approaches to increasing compute power vs increasing data load sizes

In 1965, Gordon E. Moore, a former CEO of Intel, made a claim in an article that the complexity of computer chips (specifically referring to the factors of increased silicon die size, reduction in feature size, and "device and circuit cleverness") would double every year – a claim E. Moore later adjusted to eighteen months instead of one year [50]. When talking about Moore's law however, one is actually referring to a comment by Intel executive David House who said that the combination of the effect of more transistors and the transistors being faster would lead to a doubling of computer performance every 18 months [40].

The standard of doubling computing power every 18 months became an industry norm by which manufacturers set their targets to meet expectations [33] and while the original technique of halving transistor sizes began to become more difficult, more complex arrangements of chip components allowed for David House's comment to remain accurate [16].

The size to which transistors have shrunk has however started to cause issues which makes chip improvement progressively harder. For example, due to quantum mechanics, transistors are now of a small enough scale where minute amounts of electric charge bypass materials which are meant to act as resistors, and misfire into neighbouring transistors [28, 26] – plus the difficulty in controlling electrical charges at this scale has also prevented manufacturers from increasing system clock speeds [33]. Moreover, with transistors measuring 5 nanometres across with a 1 nanometre gate [20], transistors are nearing the limit of how small they can be made with atomic particles.

While the rate at which processing power increases is decreasing, the size of data sets has been outpacing processing speed before the decline in processing speed increase even began [3, 69]. As current examples, the Australian Square Kilometre Array Pathfinder project (a 36 identical parabolic antenna radio telescope array located at Murchison Radio-astronomy Observatory in the Australian Mid-West [21]) acquired 7.5 terabytes per second of sample image data in 2012, a rate which is projected to increase 100-fold to 750 terabytes per second (25 zettabytes per year) by 2025 [37, 51] while the total number of protein structures in the worldwide protein data bank almost doubled between 2008 (7000) and 2017 (13 049) [57].

To deal with such increasing data loads, processing initiatives are increasingly relying

on powerful co-processors and distributed computer infrastructures to generate the additional compute, memory, networking, and storage resources which individual processing nodes are unable to supply on their own. This type of approach where multiple compute nodes, with varying degrees of private resources, base functionality and shared memory are referred to as parallel computing.

Parallel computing comes in many forms and they can be categorised into fine-grained, course-grained and embarrassingly parallel tasks based on how often subtasks within the computing being done need to communicate. Embarrassingly parallel tasks are ones which rarely (or never) need to communicate with each other - embarrassingly parallel tasks are thus a good fit for distributed computing approaches as large data batches do not often need to be sent back and forth between the separate computing nodes (which was the approach taken with this research's MPI algorithms). Course-grained parallelism refers to subtasks which do not need to communicate multiple times per second - these types of subtasks require more handling then embarrassingly parallel tasks but generally fit well into distributed memory architectures as the availability of shared memory resources between computing processors helps alleviate the fast paced communication requirements (and was thus the approach taken with this research's openMP algorithms). Lastly, fine-grained parallelism refers to tasks which need to communicate multiple times per second. This sort of parallelism requires careful coding and can be fairly difficult to get right. There are however types of powerful co-processors which are physically built to accommodate large amounts of parallel processing (and require large mounts of parallel processing to be fully utilised) which are ideal for fine-grained parallel tasks and often can not handle embarrassingly parallel tasks due to lacking the memory or processing freedom to queue up multiple large embarrassingly parallel tasks at once. One such co-processor which can be utilised to execute fine-grained parallelism is known as a graphical processing unit (or GPGPU as the last few generations of GPUs have mostly been general purpose graphical processing units) because of the inherently parallel nature of their architectures which require high degrees of parallelism for their architectures to be efficiently utilised.

While many exaggerated claims about the superiority of GPU inclusive processing over purely CPU based processing are frequently circulated [43], GPUs have been shown to provide superior processing speeds in certain applications under fairly specific circumstances. Despite the possible gains of implementing GPU inclusive coding however, the difficulty in programming with GPUs and predicting the effective changes in processing speed when GPU coding is implemented, have led to the pursuit having a fairly mixed acceptance [47].

2.5 Measuring performance gains in parallel systems

Depending on whether one is comparing sequential algorithms to parallel algorithms or parallel algorithms to other parallel algorithms, there are different metrics which are more appropriate.

In the case of comparing sequential algorithms to parallel algorithms, one can make note of how scalable the parallel platform being used is and what percentage of the code being run can actually be parallelized. These will be discussed with reference to Amdahl's law [5] and Gustafson's law [31]. While comparing sequential algorithms to parallel algorithms is relevant for certain sections of this research (such as the side objective of comparing hashed index and kd-tree construction times), the main experiments are between parallel algorithms exclusively – which meant that the metric of speedup, while relevant, is not a precise fit.

This section will give a brief description of the types of parallel architectures looked at with reference to Flynn's taxonomy [24], and then definitions of speedup, Amdahl's law and Gustafson's law, will be provided. Note however that the metric of the main experiment is absolute completion time and these metrics are added here solely to give a background to related fields.

2.5.1 Flynn's taxonomy

Flynn's taxonomy is a classification of computer structures proposed in 1966 and serves as a tool for designing and describing processors [25]. A brief description of the project relevant Flynn classifications are given below.

Single instruction stream, single data stream

Abbreviated as SISD, single instruction stream, single data stream, is a computer architecture in which a single uni-core processor executes a single instruction stream to operate on data stored in a single memory. This is a good description of how a basic, non-parallel, desktop computer would operate - although most modern computers do have low level parallelism built into them.

Single instruction stream, multiple data streams

Abbreviated as SIMD, this describes a computer architecture where multiple processing elements each process different data elements at the same time utilising data level parallelism.

Single instruction, multiple thread

Abbreviated SIMT, Single instruction, multiple thread is an extension of Flynn's taxonomy and is an execution model used in parallel computing.

The term SMT refers to the ability of a GPU to allow multiple program threads to be run at the same time. SIMT is thus a computing architecture which meets both the requirements of SIMD and SMT.

Multiple-instruction, multiple-data

Abbreviated MIMD, multiple-instruction multiple-data refers to an architecture design were there are a number of processors that function asynchronously and independently. In these architectures different processors may be executing different instructions on different pieces of data.

2.5.2 Parallel Speedup

Speedup is a measurement of improvement in computer science. While it can have wider ambiguous use, its original purpose was to compare the performance of a sequential system with the performance of a sequential system which had been enhanced by parallel processing [35]. The original definition of speedup is:

$$S_p(n) = \frac{Tseq}{Tpar(n)} \tag{2.1}$$

where:

- $Tseq$ represents the best sequential execution time that could be implemented
- $Tpar(n)$ is the parallel execution time with parallelism degree equal to n

If the speedup factor is n, then one says an n-fold speedup has been achieved. For example, if a sequential algorithm takes 15 minutes and the corresponding parallel algorithm only takes 3 minutes then one would say a 5-fold speedup was achieved.

2.5.3 Parallel Efficiency

Parallel efficiency is a measure of the parallel speedup gained against the amount of parallelism (or processors) utilised to achieve it, ie:

$$E_p = \frac{S_p(n)}{p} \tag{2.2}$$

Where:

- $S_p(n)$ represents the parallel speedup achieved with a parallel algorithm
- p is the number of processors (or number of parallel resources) utilised in the run of the parallel algorithm.

This metric is useful when determining how much speedup one gains from each unit of parallelism added to a parallel implementation as there are usually diminishing returns for each parallel resource added to computing a task of fixed size.

2.5.4 Amdahl's law

Amdahl's law reads:

$$MaximumSpeedup = \frac{1}{r_s + \frac{r_p}{n}} \tag{2.3}$$

where:

- $r_s + r_p = 1$ represents the ratio of the sequential portion in one program (r_s) to the parallel portion (r_p)
- n is the number of processors

This equation expresses the theoretical limits of attainable compute speed improvements which can be made in programs which contain both sequential and parallel components [5]. The conclusion being that even if the parallelized portion of the

code is reduced to near zero execution time through a very high number of processors, the total processing time of the code cannot be reduced to less than the sequential component; ie.

$$Speedup_{n->\infty} = \frac{1}{r_s} \tag{2.4}$$

On top of that however, Amdahl expressed the view that even in extremely parallel tasks, there remained components and bottlenecks which were not actually parallel, despite their setting. This view emphasised that things like increasing communication requirements between distributed computing and the indivisibility of certain tasks places a limit on the improvements of parallel systems before the architectures which they were built on caused detrimental overheads.

Gustafson's Law

$$Speedup = n + (1 - n)s \tag{2.5}$$

where:

- n represents the number of processors
- s is the serial fraction of the process which does not benefit from the parallelization

Gustafson's law was in response to Amdahl's law and the assumption that there was finite improvement which could be achieved through parallel processing. This difference came from the perspective that in most real world scenarios the size of the parallel portion of the code is not independent of the number of processors being used but rather that the parallel portion is confined by total run time – meaning that using more processors would allow a larger parallel portion of code to be executed while still meeting maximum run time required by different scenarios.

This was demonstrated in a paper showing that a 1024 processor setup had successfully scaled the parallel portion of 3 different codes by a factor of just over 1023 despite Amdahl's law predicting that a performance increase of a 100 fold increase should be about the limit [31].

2.6 Parallelization Platforms

2.6.1 OpenMP

Architectural basis: shared memory architectures

OpenMP works with scalable shared memory multi-processor architectures (MIMD) which are systems in which the native programming model is shared memory access. Shared memory refers to (usually large) blocks of random access memory (RAM) which can be accessed by different processing units.

OpenMP is a shared-memory multiprocessing application program inference (API) for easy development of shared memory parallel programs [53]. It provides a set of compiler directives to create threads, synchronize the operations and manage the shared memory on top of pthreads.

The layout of openMP code is a block-structured approach in which sections of the code which are to be run in parallel are marked as such at which point the program will split from a single thread of execution to multiple threads of execution – which is followed by a new sequential thread when the parallel block has been completed.

When implementing openMP code the compiler takes care of transforming sequential code into parallel code according to its directives. When executing openMP code, openMP maintains a thread pool to deal with any blocks of code that have been marked for parallel execution [22].

Of significance is the way threads work in openMP is that openMP will create a thread pool and instantiate multiple threads off the main program thread until such a time as those threads are terminated. When using multiple threads in openMP, each thread has full access to all the resources being used by all the threads – which means race conditions or memory locks could occur if the code causes such overlaps in threads and does not handle them.

2.6.2 MPI

Architectural basis: distributed memory systems

MPI is a message passing library specification which defines an extended message passing model for parallel, distributed programming on distributed computing environment [30]. Distributed memory systems are multiprocessor computer systems in which each processor has its own private memory and each processor can only operate on its local data – which necessitates communication approaches such as MPI if processors need to exchange their local data.

As a specification, MPI is not an implementation of the message passing model it defines but several implementations do exist such as the openMPI implementation used in this research. In the MPI model, each process has its own address space and communicates with other processes to access others address space. Developers are responsible for partitioning workload and mapping tasks about which tasks are to be computed.

MPI provides point-to-point, collective, one-sided, and parallel I/O communication models [30]. Point-to-point communications enable exchanging data between two matched processes. Collective communication is a setup where all members need to wait for (or at) a synchronization point while a selected member performs broadcast, gather and scatter operations to the other members of the set. One-sided communications facilitate remote memory access without matched process on the remote node.

How OpenMPI works

OpenMPI allows users to instantiate multiple OpenMPI processes. OpenMPI processes are contained environments which means that the data within the OpenMPI environments cannot be shared (or further data retrieved) except through message passing APIs built into OpenMPI.

While one can instantiate as many OpenMPI processes as one wants, unless there is enough processing power to run each OpenMPI process, having too many OpenMPI processes may start to slow the processes they run down instead of speeding them up.

OpenMPI processes are usually mapped to processor cores. As an example, this research was run on a CPU chip which had two processors each with six processing cores on them – and thus the largest set of OpenMPI processes which could be run on that chip before negative impacts started to appear happened to be 12.

2.6.3 General purpose graphical processor unit computing

Architectural basis: single instruction multiple data

Modern GPUs implement a category of computing known as SIMT which is a combination of single instruction, multiple data (SIMD) and simultaneous multi-threading (SMT). This implies two sets of behaviours and each causes different types of constraints.

The SMT aspect of SIMT architectures simply refers to the ability of a GPU to allow multiple program threads to be run at the same time. In GPUs this is enacted by the high number of arithmetic logic units (ALUs) which GPUs contain of which each can work on their own thread (and if need be, each ALU can work on multiple threads through temporal multi-threading which just consists of swapping between different threads really fast).

The SIMD aspect of SIMT refers to a behaviour in which a group of processors are all forced to perform the same set of instructions at the same time but each processor is capable of performing those identical instructions on different sets of input data – in NVIDIA architectures, this constraint is manifest in the ALUs which are bound in groups of eight to a single execution context cache. Because each processor in a SIMD setup has to execute the exact same instruction, major performance penalties occur if some processors have to execute a different logical branch of the instruction being run through – which makes avoiding branch divergence when coding for SIMD or SIMT systems a high priority. There are physical advantages to forcing multiple processors to compute the same instruction in sync however. SIMD architectures allow for each in-sync processor to share the same execution context cache. This is fairly important for parallel processors as when placing hundreds to thousands of processing cores onto a single card one needs to utilize space as efficiently as possible to fit all those processors in. In the context of NVIDIA, the NVIDIA example, eight ALUs will each share a single (physical) execution context – so while each ALU will still need a small private memory cache to operate, each grouping of eight ALUs is able to cut out the need for seven additional execution context caches and still

functions appropriately.

The combination of SMT and SIMT is what causes GPUs to be so difficult to program with. Hosting progressively larger amounts of processing nodes requires instituting efficient coordination between the nodes as well as cutting down on the space that each nodes requires on the chip. The result of these trade-offs however is architectural resources which are hard to efficiently utilize on tasks which the architectures were not innately designed for.

The History of GPUs

The precursor to GPUs came into being in the 1970s and were known as video controllers. Video controllers were hard coded to output certain visuals related to the computer games they were part of [12].

In the 1980s discrete GPUs started to develop which were the first devices which would accept offloaded commands from the CPU and render graphics on their own. A notable tipping point occurred in 1987 with the release of the IBM 8514/A which was the first GPU which could draw lines on computer screens faster than a CPU could handle [12].

In 1992 the first open application programming interfaces (APIs) started to appear, with openGL launching in 1992 and directX in 1995 [18]. These standardised programming platforms made GPGPU programming far more accessible to non-experts for the first time.

For the purposes of this research the final milestone of note was the 2006 release of the GEFORCE 8800. This GPU had 681 million transistors (a large amount for the time), a unified shader and could operate at faster clock cycles than the CPU. This model featured a number of stream processors which allowed for graphics tasks to be run in parallel and helped open up the availability of general purpose graphical processing unit computing [48] (GPGPU computing) for use in activities in scientific research or bit coin mining.

The transition from GPUs to GPGPUs

While there are now a range of dedicated GPGPUs available for purchase (such as the NVIDIA Tesla K20Xm model utilized in this research) which are dedicated

to computational use and cannot actually render viewable graphics like their GPU predecessors, all modern GPUs are actually GPGPUs as well.

GPUs use programs called *shaders* to generate the appropriate amount of shading (levels of light, dark and colour) onto the images they visualise. In 2003 the NVIDIA GeForce 3 was released which was the first GPU to feature floating point operations and, more importantly, programmable shaders. While the GeForce 3 implemented these shaders as a way for computer games to provide their own optimised shaders, this enhancement allowed programmers to submit their own shader code to be executed on the GPU hardware – allowing the processing power of the previously dedicated graphics devices to be directed to computational objectives not related to graphics at all.

Early efforts at using GPUs as GPGPUs required users to reformulate computational problems in terms of graphics primitives, an end which was supported by the APIs known as OpenGL and DirectX. These cumbersome translations were improved by newer general-purpose programming languages and APIs such as Sh/RapidMind, Brook and Accelerator which were in turn followed by platforms such as NVIDIA's CUDA, which allowed programmers to ignore the underlying graphical concepts in favour of more common high-performance computing concepts. Other modern APIs which allow abstraction of the underlying graphical concepts include Microsoft's DirectCompute and Apple/Khronos Group's OpenCL.

An introduction to general GPU architectures

GPU architectures have changed and become more powerful with each consecutive generation of models [46]. Despite the changes to design and continuous innovation there are three general GPU roles which tend to remain constant across consecutive devices: the GPUs on chip global memory, streaming multiprocessors, and stream processors. To explain the roles that these devices perform it is best to start at the smallest component and work upwards.

A streaming processor (also known as CUDA cores or texture shaders in the NVIDIA architecture) are the components in GPUs which carry out computations [68]. Each streaming multiprocessor will have its own arithmetic logic unit (ALU) to perform computations and a small private memory cache with only 16, 32 or 48 KB of storage space.

In GPUs, streaming processors are grouped together in sets of 8, 16 or 32 inside

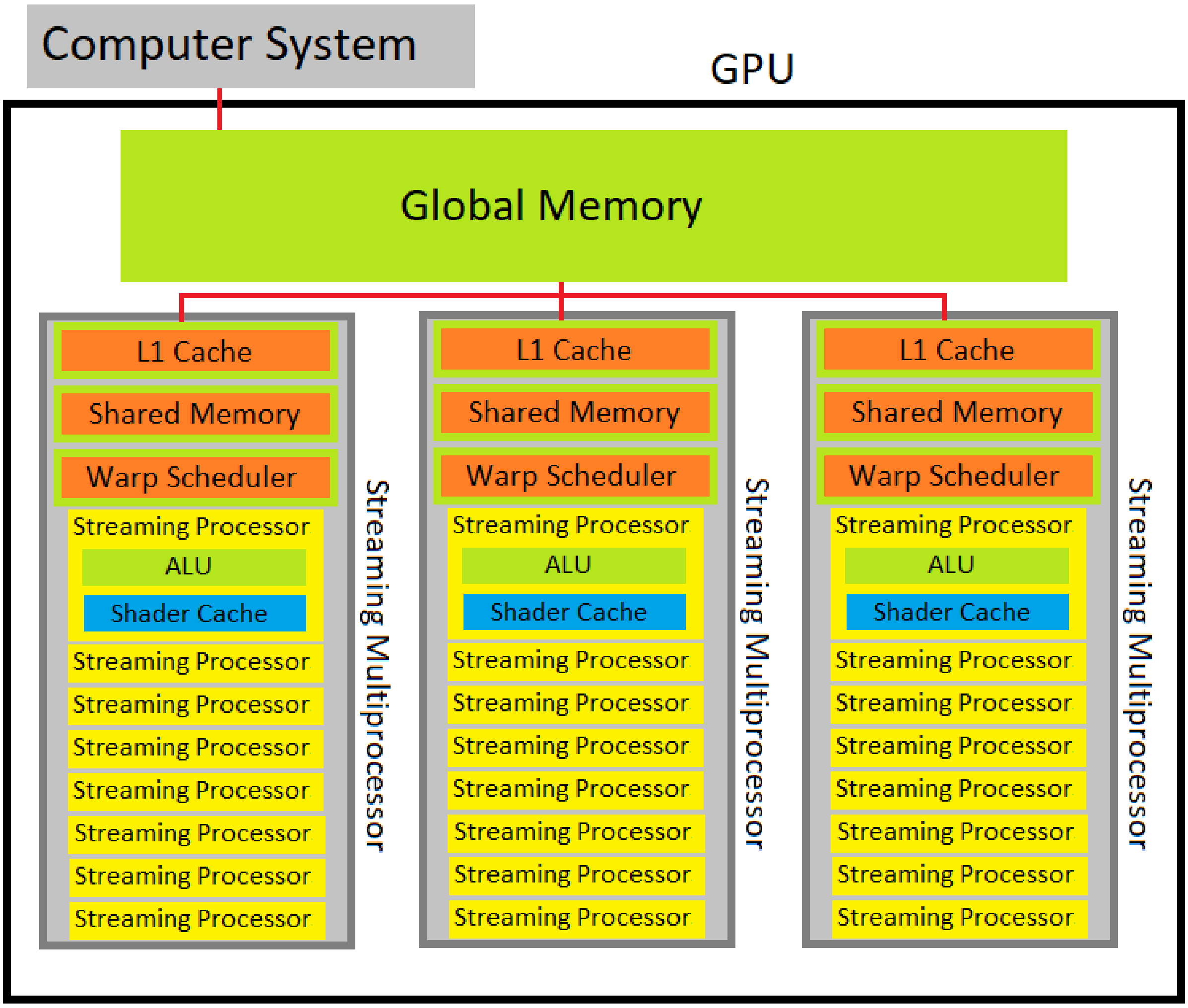


Figure 2.6: Illustration of the GPGPU memory pipeline whereby data is passed from the computer system down through global and local memory and into the processing cores private memory caches

constructs called streaming multiprocessors. Streaming multiprocessors will have a single execution context cache (called a warp scheduler in NVIDIA architectures) which dictates the code step that all of the streaming multiprocessor's streaming processors execute at the same time – which is what causes the SIMT constraint of forcing each processing core to work in sync. Each streaming multiprocessor will have a shared memory cache which all the streaming processors can access (albeit only one can access it at a time) which is larger than streaming processors' private caches. Each Streaming multiprocessor will also have a L1 Cache which is a fast access cache used to reduce latency between the streaming multiprocessors shared memory and the GPU's Global memory [70]. A final note is that while every streaming processor within a multiprocessor must execute the same command at the same time, different streaming multiprocessors are allowed to be on different steps

of the same code or run different codes at the same time.

Streaming multiprocessors are then connected to the GPU's global memory which is a very large memory cache used for marshalling data coming in from the outside computer system, grouping data to be sent back to the outside computer system as well as handling the distribution of data to the GPU's streaming multiprocessors. Unlike other parts of the GPU, global memory can only be accessed sequentially by GPU components – meaning that it can only pass data to one specialised GPU component at a time. While the GPU is busy processing compute jobs however, the GPU can still pass data to and from the computer system it is attached to. This is an important feature as it allows the GPU to continually fetch data for processing before the GPU wishes to use it – which helps prevent load time bottlenecks to the GPU's multiprocessors.

2.6.4 Architecture of the Tesla K20Xm GPU

This research used a Tesla k20Xm GPU for its parallelized kd-tree pre-processing and range searching algorithms. For that reason, the devices relevant specs are listed here:

- GPU Chip: Kepler GK110
- Memory bandwidth: 250 GB/s
- Global Memory: 6GB GDDR5 memory
- Number of CUDA cores 2496
- Compute Capability (physical feature release): 3.5

2.6.5 CUDA

Introduction

CUDA is a parallel programming language and programming model which is developed and maintained by NVIDIA and primarily designed to operate with NVIDIA GPUs. Originally released in June 2007, CUDA was built as an extension to the C programming language, CUDA libraries allow compilers to compile sections of C, C++ and Fortran code into parallel thread execution (PTX) code after which

NVIDIA graphics drivers use a standard include compiler to convert the PTX code into binary code which can be natively run within NVIDIA graphics cards. This utility thus allows GPUs (or more optimally, NVIDIA GPUs) to be exploited as compute devices for parallel data applications. While writing code to be executed on GPUs has previously been possible, only with the release of parallel computing platforms such as CUDA, CTM and OpenCL, has the pursuit become simple enough to be attempted by non-specialists.

CUDA does not require programmers to manually specify how individual tasks should be managed on GPUs. Rather, CUDA provides an abstraction to allow programmers to express how they want their programs to be executed on a GPU when run. To achieve this, a CUDA inclusive script will generate a PTX code when compiled. When the CUDA inclusive script is run, the compiler will then use that PTX code to compile code specific to the implementation's target GPU. Optimising factors such as the number of registers used or allocated shared memory is possible but catered for by the abstraction if ignored.

To achieve this level of abstraction, CUDA implemented the open industry OpenACC 2.0 standard. The OpenACC standard (developed by PGI, Cray and NVIDIA) outlines simple compiler directives and hints which programmers can use to specify which sections of code should be passed onto a GPU. Under this standard, CUDA specific commands can be inserted into normal C/C++ code which will be ignored if run without a CUDA compiler (as in such situations the CUDA code will be treated as comments) but will actively assign marked sections of code for GPU processing if run through a CUDA compiler.

This setup means that CUDA is easily compatible with existing C/C++ code as CUDA scripts behave like normal C/C++ code as default – only sending commands to a CUDA kernel where the code expressly commands the compiler to do so. As CUDA codes usually include a CPU and GPU working in tandem, this allows for simple processes to be run on the CPU exclusively while sending only the computationally heavy sections of code to a GPU for processing.

CUDA Programming basics

General purpose GPU programming operates around the concepts of kernels, threads, warps, blocks and multiprocessors. Kernels are scripted functions which can be run on GPUs. The code written in kernels is near identical to the code written for CPU

scripts, the main difference being that kernel functions can only be run on GPUs unless the kernels are explicitly stated to be available on both the host and the device. This GPU-exclusive identification is noted by a keyword before functions to indicate they are meant to be run by the GPU. In CUDA, the keyword `__Device__` indicates that a function can only be run by a GPU while the keyword `__Global__` will make a function available to scripts running on both the CPU and GPU. The keyword `__Host__` ensures a function can only be run by the CPU but functions default to this setting if no other keyword is specified.

Kernels are submitted to GPUs with the intention of being executed in parallel. This means that multiple instances of at least portions of the scripts which compose the kernel can be executed at the same time. These multiple instances which need to be executed in parallel are known as threads. While CPUs can handle multiple threads (for multi-threaded scripts) by quickly switching between processing different threads, GPUs handle multiple threads by passing them to streaming multiprocessors – chips featuring multiple processing cores (known as streaming processors or CUDA cores) which allows for up to 32 threads to be executed at the same time (in parallel) on each streaming processor.

When executing groups of threads on streaming multiprocessors, there are two terms used to refer to the thread groups. A warp is a group of 32 threads which is the maximum number of threads that can be executed in parallel by each streaming processor in a streaming multiprocessor. A block refers to a group of threads which have been assigned to a streaming processor for processing – which can be any number from 1 thread to 32 threads. All threads being simultaneously run in a block must execute the exact same code script. While each thread in a block runs the same code script, each thread retains its own block/thread ID which acts as a starting point for obtaining different results from the same code script. When submitting code to a GPU for execution, the user specifies the number of blocks to be run and the number of threads to be instantiated per block. Which multiprocessors are assigned to blocks is a job handled by CUDA and not one which can be controlled by programmers.

When blocks are passed to a multiprocessor, the multiprocessor takes groups of threads from their assigned blocks and executes them in parallel. The number of threads a multiprocessor can execute at once depends on the containing GPU's architecture. When programming on an NVIDIA GPU with the CUDA programming platform, the maximum number of threads a multiprocessor can handle at a time is 32; and so 32 threads are referred to as a warp. While multiprocessors can only

operate on 32 threads at a time, this does not mean that a block can only contain 32 threads. When a block contains more than 32 threads, the multiprocessor will extract 32 threads for processing and when any of those 32 threads complete, the multiprocessor will fetch an unprocessed thread from the block to take its place. The ideal number of threads that blocks should contain is situation dependant and hard to determine without experimentation.

NVIDIA multiprocessors employ a SIMT architecture which allows for a unique pipeline technique for similar processes being run in parallel. Utilising shared memory, each thread in a warp will start at the same programming address. However, each thread maintains its own instruction address counter and register state. This allows each thread to branch and execute freely even when the entire warp is operating off the same programming address based command.

The last constraint that needs to be acknowledged is the amount of memory available to individual threads running on multiprocessors. Each processing core on a GPU has a small amount of personal memory and while each core can access their memory cache extremely fast, transferring data between these caches and the GPU's general memory is much slower by comparison – and while data is being transferred between a core's cache and the GPU's general memory, both that core and the general GPU memory can perform no other operations. Because of these memory constraints and slow data movement speeds, cores need to be programmed in such a way that they make communications with GPU general memory as seldom as possible.

Avoiding data races in GPU kernel scripts

A data race is a situation where two kernels try to access (or overwrite) the same value in memory. As an example, imagine a GPU kernel which counts the number of positive integers in a list – incrementing a counter in general GPU memory every time a positive integer is found. This code will run fine if only a single thread is initiated but if 512 threads of this kernel are run, they may start overwriting each other's results. As one thread reads the current counter from global memory, 12 other threads may read the exact same counter value before the first thread has a chance to increment the value of that counter – so where the counter should have increased from the value of 0 to 13, it will instead be increased to the value of 1, 13 times. Worse, GPUs offer no guarantee of the order in which threads will have access to general GPU memory. It is completely possible that after the first 13 threads access the general memory counter, only 12 will write back the value of 1

before 200 more threads read the counter value of 1, each of the 200 threads may then increment the value to 2 and then the 13th thread from the initial batch could decrease the counter back down to 1 again.

There are three general approaches to dealing with such issues. The first approach is the forced thread synchronisation command. Thread synchronisation (initiated in CUDA with a `__syncthread()` command) forces all threads in a block to halt at that position in the script until every thread has reached that position, at which point all the threads may continue with the kernel script. While forcing halts in this way can slow down the runtime of a kernel, it allows for certain operations to be safely carried out: such as forcing every thread to finish reading a value from a singular memory location before allowing any thread to write to it again.

The second approach revolved around the use of atomic operations. Atomic operations can perform some basic mathematical operations on values in general GPU memory with the guarantee that only a single instance of an atomic operation can be carried out at a given time. This means that when a thread atomically increments a value in general GPU memory, it is guaranteed to be returned the value of the number before incrementation and also guaranteed that no other thread will be able to access or modify that value until the atomic operation is done. Atomic operations have dedicated physical circuitry which allows them to execute faster than normal private-cache-to-general-GPU-data-transfers and provide, amongst other functionalities, a sound basis for noting how many instances of results have been found across different threads and writing each result to a unique memory address.

Handling data transfer efficiency

In the order of least to most expensive, data transfer is: [shared memory to thread cache], [thread cache to shared memory], [general GPU memory to thread cache], [thread cache to general GPU memory] and finally [CPU memory to/from general GPU memory]. This highlights the importance of sending data from the CPU to the GPU (and from the general GPU memory to threads) as seldom as possible. That noted however, a thread, or bunch of threads, fetching a value from general GPU memory is far faster than a thread writing a value to general GPU memory.

There are several techniques that help alleviate these bottlenecks in instances where they cannot be avoided. For sending data from private caches to general GPU memory, it is better to send large chunks of data as opposed to individual values.

This means that if a thread or block of threads would amass multiple results during their execution, it is far more efficient to save up all the results and send them to general GPU memory at once. The same applies in reverse as well. Data can be sent from general GPU memory to threads much faster if the data being sent is coalesced – and speed up can thus be attained if the programmer manages to organise the data in large chunks to be sent back and forth.

The other technique for speeding up memory transfers between private caches and general GPU memory has to do with the shared memory of each multiprocessor. Shared memory is a type of memory physically built into multiprocessors which each individual streaming processor within the multiprocessor can access. Accessing shared memory is much faster than accessing general GPU memory but while shared memory is larger than each streaming processor's private memory cache, shared memory is still much smaller than general GPU memory. Still, if large chunks of coalesced memory are to be utilized by each thread within a warp, then that data can be loaded once into shared memory and then accessed by each individual thread at a much faster pace. Likewise, shared memory can be used to accumulate the results of every thread in a block and then send the batch of all the completed results to general GPU memory once the entire block of threads have completed their operations.

For sending data between the CPU and the GPU, there are two basic techniques to alleviate bottlenecks. The first is known as memory pinning. Memory pinning maps a section of normal memory so that the GPU knows exactly what that memory contains – making pinned memory an extension of GPU memory with a much slower fetch time. Once CPU side memory is pinned, editing the contents of that memory from the CPU becomes very slow, so the most practical implementation is to first marshal all the data to be sent to the GPU into a coalesced memory location, pin the memory and then send it to the GPU to be processed.

The other technique for speeding up data transfer to the GPU has to do with the fact that GPUs can process internally stored data and send/receive data from the CPU at the same time. This means that instead of loading an entire data set to the GPU, processing the entire data set and sending all the data back, a user can instead keep sending segments of a data set to the GPU, overlapping the transfers and data processing stages of the code to more efficiently utilize the GPU's processing time.

2.6.6 Examples of spatial data structures and protein data being applied in GPU settings

NAMD molecular dynamics library

NAMD is a molecular dynamics simulation library used to compute the trajectories of atoms in very short time steps by applying empirical force fields to each atom in the system instead of calculating the exact forces acting on each atom in the system [55]. While using empirical force fields instead of calculating the specific force fields acting on each atom does have a slight accuracy cost, the savings in compute power achieved with these empirical fields allows for larger and more complex molecular systems to be simulated than exact force field calculations would allow on the same set of hardware.

The NAMD library also has scalable support for parallelization across multiple (to hundreds) of processors on parallel platforms, distributed parallelization across clusters of separate compute nodes [55] as well as support for execution upon GPGPU environments [63]. NAMD is able to scale in this way by using a technique called atom decomposition where in atoms which are close to each other are grouped into sets and then sets are assigned to different processors to divide the atom calculations required cleanly across multiple processing nodes [39]. Examples of the use of GPU based NAMD implementations include research into generalized born/solvent-accessible surface area implicit solvent calculations [66] and free energy simulations with the AMOEBA polarizable force field [54, 73].

Computing non-bonded interactions for molecular dynamics

Eastman et al.[23] provides a competitive and scalable GPU implementation for calculating non-bonded interactions. Instead of using an algorithm optimised for CPUs (which has a run time of $O(n)$), this research started with an $O(n^2)$ algorithm which was already optimised for GPU processing [27] and made improvements to reduce its computational complexity.

The algorithm operates by loading 32 atoms into each GPU processing block (to match the CUDA 32 warp size) while loading the positions and force field parameters for the atoms into the blocks shared memory – for fast access by each warp in the block and to greatly decrease the amount of data transfer between shared memory and the GPU's main memory (as the shared memory cache uses results in only one

atom being loaded per 16 interactions evaluated, compared to two atoms for each interaction when using a more traditional neighbour list).

In addition to this base approach, the axis aligned bounding box of each block of atoms is pre-computed to identify blocks which are too far away to interact with the atoms of other blocks and these blocks are not computed at all (achieving the same benefits as a neighbour list) and the atoms are pre-sorted to help ensure that the atoms in each bounding box will be close to each other – which was noted as being easy to achieve in proteins as the sequential chains are always relatively close to each other.

This approach resulted in a runtime which experienced linear completion time growth with increasing atom loads and was 28 times faster than NAMD (an award winning parallel molecular dynamics code) as opposed to the algorithm it was based off [27] which was only 5 times faster than NAMD

Generating histograms for molecular dynamics trajectories

Radial distribution functions (RDF) define the probability of finding a particle at a given distance from another tagged particle. RDF is a compute expensive function within molecular dynamics trajectory processes and thus a bottleneck to other, faster parts of those functions. In ***Levine et al.*** [44], a methodology is presented for generating histograms of atom pair distance values (for RDF) which is run on and can be scaled across multiple GPUs. While the calculations for the data points which are summed in the histograms are straight forward, compiling the results from multiple GPUs is not. The paper discusses the many software (CUDA) and GPU hardware (atomic operations) which are applicable to increasing the efficiency of such tasks under different situations. The final version of the algorithm presented, running in parallel on four NVIDIA GeForce GTX 480 (Fermi) GPUs, was found to be 92 times faster than a multithreaded implementation running on an Intel Xeon 5550 CPU.

Implementing geospatial algorithms on GPUs and scaling to clusters

The paper by ***Zhang et al. (2015)*** [74] describes implementations for process geospatial data on GPUs as well as approaches for integrating those GPU based geospatial algorithms into processing clusters such as Amazon's EC2 – while noting that the former algorithms are new, novel approaches and that most popular

processing cluster implementations do not have support for GPU integration at this time.

The first half of the paper details several self-developed GPU-based spatial data indexing techniques which use data presented in either grid-file, quad-tree or R-tree format as well as developed techniques to join various indexed spatial data sets on a GPU.

The second half of the paper provides two approaches for including GPU processors in distributed computing solutions. As most big data systems do not support spatial data processing, the approach taken was to use the single-node GPU-based techniques to improve processing time per node and reduce the amount of inter-node communication required to complete work loads.

The first approach was to utilize the MPI parallelization stack available on the Oak Ridge National Lab (ORNL) Titan Supercomputer (the ORNL Titan having an equal number of CPU and GPU processing nodes). The approach involved pairing up country MBRs (minimum bounding rectangles) with raster tiles (rasterisation being the process whereby GPUs convert shape vectors into the flat pixel configurations displayed on screens) and then sending each paired tile to a Titan computing node through the MPI API. Once in the titan nodes, the elevation data stored within the then divided raster tiles (as well as polygon overlap verification) was efficiently computed with GPUs. This approach was able to generate elevation histograms for 3000+ US counties over 20+ billion raster cells in about 10 seconds using 8 titan nodes.

The second approach extended Cloudera Impala (an interactive SQL query platform for dealing with Apache Hadoop data stored in Apache HDFS or Apache HBase data distributions) with modules which allow the Impala's existing infrastructure to be used for spatial query processing – in the demonstrated case this was used to process taxi trips.

The additions made to Cloudera Impala were front end support for inputting spatial query syntax (so users could pass in spatial queries), and back end support for storing spatial data as strings and improvements to the extraction speed of spatial data from sets.

Using 6 Amazon Elastic cloud (EC2) instances which each contained 10 g2.2xlarge instances (each with 8 Intel Sandy Bridge 2.6 GHZ vCPUs, 15 GB memory and an NVIDIA GPU with 4 GB graphics memory and 1,536 CUDA cores) allowed for

processing 170 million taxi trip pickup locations (6.9 GB of data) in about 30 seconds while increasing from 6 to 10 EC2 instances decreased the taxi query run time from 30 seconds to 21 seconds – showing reasonable scalability.

Chapter 3

Research Objective

The primary objective of this research was to test the performance characteristics of GPUs against openMP and MPI at performing range searches on large protein databases.

To be thorough in this objective, the run times of both brute force and kd-tree based GPU range search algorithms were compared to the run times of equivalent brute force and kd-tree based range searches utilising the openMP and MPI parallel platforms.

To cover the details of this process, this chapter is split into 3 sections. The first section provides a summary of the test scenarios which all the algorithms were run through and what each set was run to accomplish. In the second section the metrics by which the algorithms were compared will be outlined. Finally, the third section lists the GPU algorithms implemented with a brief description of their phases, followed by a list of all the other algorithms they were compared against from the other parallel platforms. At the end of the third section there will be brief descriptions of the different approaches compared for generating the hashed indexes and kd-trees used in most of the algorithms.

Following this chapter will be a chapter dedicated to explaining the implementation details of the algorithms noted in this chapter with emphasis put on the developed GPU based kd-tree algorithms which proved, while efficient, to be very complex.

3.1 Planned Comparisons

This research compared the time that GPU algorithms took to range search large protein data sets to the time that other parallel algorithms took to range search the same data sets. Due to the difficulty of accurately determining the performance of parallel implementations through theory, this research took the experimental approach of running all the implemented algorithms through 4 sets of data for each experiment – with every experiment being run 4 times to get an averaged result.

The main focus of this comparison was the time required to complete the actual range search phase of the algorithms while the time taken to pre-process data for the range searches (to construct reusable resources such as kd-trees and hashed indexes) was a secondary interest.

As the parallel platforms of openMP and MPI are highly scalable depending on how many resources your compute setup can provide them, the Results section contains a segment showing the most competitive results which could be attained to compare against the GPGPU algorithms in the main results section.

Table of algorithms implemented for range searching:			
	Brute Force	With Hash-Index	With Kd-tree
CUDA	✓		✓
OpenMP	✓	✓	✓
MPI	✓	✓	✓
CPU - Only	✓	✓	✓
Hybrid CPU + CUDA	✓		✓

Table 3.1: In the GPU environment, the cost of brute force locating sought molecules in parallel was so low that efficiently implementing hash-index usage on the GPU (or loading lists of those atoms' positions onto the GPU was deemed unnecessary

3.1.1 Main comparisons

These comparisons were split into two stages. The first stage was to only determine if GPU range search algorithms (implemented with CUDA) were competitive with other parallel range search algorithms. This was done by comparing whether the completion time of the CUDA algorithms were about the same, significantly lower

or significantly higher in performing range searches on unordered sets of PDB files than other parallel algorithms.

To test how the CUDA algorithms compared against the other parallel algorithms, three range search queries were run on a very large set of randomly chosen PDB files. These range searches sought to find all instances where the two sought atom types were within 4 Ångstroms of each other and the run times for these searches were noted on 12500, 25000, 37500 and 50000 files respectively.

The first search query was run on an atom pair which had a high occurrence rate. The second search query was run on an atom pair which had a sporadic occurrence rate, and the third search query on a set of atoms which had a zero occurrence rate.

The second stage of comparisons was to look into what sort of data loads GPU processed better or worse when compared to the other parallel algorithms. This inquiry, which was deemed worthwhile due to the GPU's reputation of requiring high levels of repetitive processing to be fully utilised, which meant running batches of range search queries which would be progressively more compute-intensive for individual protein structures.

To do this, the available PDB files were divided into groups by total chain lengths (ie. PDB files of less than 1024 atoms length, files of less than 2048 atoms length, etc.) and 5000 PDB files were chosen randomly from each group of similar length PDB files. Then range searches which featured a low frequency of results and range searches which featured a high frequency of results, were run on the file groups of less than 1024, 2048, 4096, 8192 and 16384 atom lengths respectively, upon all the parallel algorithms.

3.1.2 Pre-processing comparisons

The secondary objective of this study was to look at how well GPU inclusive code handled the pre-processing steps required to construct the data lists which contained the protein data that the range searches operated on. This consisted of 3 steps:

1. Extracting protein structure data from PDB files to RAM
2. Constructing hashed Indexes (which utilised separate chaining)
3. Constructing kd-trees

While these steps can be directly parallelised with openMP and openMPI, there are several combinations of CPU and GPU code which one can use to parallelize these steps and all implemented combinations are listed in this chapter.

Of note is that GPUs cannot be efficiently used to extract protein data from PDB files and this step is included for completeness.

Table of algorithms implemented for pre-processing:				
	CPU	CUDA inclusive Variants	openMP	MPI
Extracting protein structure data from PDB files	✓		✓	✓
Constructing hash-indexes	✓		✓	✓
Constructing kd-trees	✓	✓	✓	✓

Table 3.2: While multiple variants of CPU + GPU inclusive codes were tested for constructing kd-trees, extracting data from PDB files on the GPU was not viable and constructing hash-indexes was not compute intense enough to warrant GPU inclusion.

3.1.3 Optimal openMP and openMPI comparisons

Experimentation was done to determine the optimal parallelism settings to use on the available hardware to obtain the best openMP and openMPI results (varying thread count in openMP and processors in openMPI) to compare against the CUDA results. Only the best performing settings were used for comparison against the CUDA results. A summary of the run times obtained with different parallelism settings for openMP and openMPI are provided in the Results section.

An important note with the openMPI settings however is that while openMPI has impressive scaling potential, the main objective of including openMPI was to give fair comparison to the GPU algorithms – meaning that the number of processing nodes utilized with openMPI was limited to those available on the cluster machine this research utilized. As the openMPI algorithms were implemented in an embarrassingly parallel manner, and the gains in compute time were linear with the number of processing nodes added to the openMPI runs, the openMPI algorithm would have likely continued to scale in this manner had more processing nodes been

made available to it. As is, the 12 nodes utilised were enough to show when and where distributed computing would overtake the other parallel algorithms and by what degree.

3.2 Algorithms compared

All the range searches implemented for this research were designed to find all instances within the protein structures being passed to them where two specified types of atoms were within a specified distance (in angstroms) of each other. These will be referred to as atom A and atom B in these descriptions.

While the brute force approaches differ in their method, the general method of all the kd-tree range search implementations is to receive a list of all instances of atom A within a protein structure and to descend down a kd-tree built for that protein structure for each unique atom A to check for proximities with instances of atom B.

3.2.1 The GPU based range search algorithms

GPU brute force range search

Two algorithms were implemented for the GPU brute force range search. The first was a pure brute force range search where unprocessed coordinate lists of the protein structures being processed were passed onto the GPU in batches and the raw processing power of the GPU was used to find all instances of both sought atom types, and then comparing the proximity of each possible atom pair in a highly parallel way.

The second GPU brute force approach incorporated a CPU hashed index which held atom counts of each atom type within each protein structure. In this approach, the CPU performed a cheap check to see if at least one instance of atom A was present in each protein structure before passing those protein structures to the GPU for processing – an approach which greatly improved the runtime of the GPU brute force range search over large data sets at the expense of a long pre-processing window to construct the CPU based indexes for the entire dataset.

GPU kd-tree range search

This algorithm worked by loading batches of kd-trees for protein structures onto the GPU and then for each kd-tree loaded onto the GPU, located all instances of atom *A*s for individual proteins with a CPU-end hashed index and then passed the list of atom *A*s to the GPU which would then run through the kd-tree of that protein for each atom A in a highly parallel way (which shall be described in detail in chapter 4).

3.2.2 The other parallel algorithms run for comparisons

CPU brute force range search

This approach just parsed through unprocessed protein structures with normal sequential CPU coding to build a list of all instances of atom A and atom B and then did a brute force comparison of the distances between all combinations of atom *A*s and atom *B*s. This approach has a runtime of $O(n^2)$

CPU hashed index range search

This approach used pre-constructed order preserving hashed indexes which utilized separate chaining to store the counts and location references to all the types of atoms found in each protein record (this exact hashed index is the one used by all hash index utilising algorithms in the paper). These hashed indexes were then used to construct lists of the coordinates of both atom As and atom Bs and then brute force comparisons of the distances between all combinations of atom As and atom Bs were carried out.

CPU kd-tree range search

This approach used the pre-constructed hashed indexes containing the counts and references to all the types of atoms found in each protein to construct lists of the coordinates of atom As in each protein and then checked for proximities with atom *B*s in each protein by parsing the lists of atom As through pre-constructed kd-trees associated with each protein.

OpenMPI hashed index range search

This approach implemented an exact copy of the *CPU hashed index range search* mentioned above but did so across a set of separate processing nodes – with each processing node receiving an equal sized portion of the total PDB file set to process and then recompiling all the results together once all the processing nodes were done.

OpenMPI kd-tree range search

This approach implemented an exact copy of the *CPU kd-tree range search* mentioned above but did so across a set of separate processing nodes – with each processing node receiving an equal sized portion of the total PDB file set to process and then recompiling all the results together once all the processing nodes were done.

OpenMP hashed index range search

This approach implemented an almost exact copy of the *CPU hashed index range search* mentioned above but used multi-threading to range search multiple PDB structures with this method at the same time.

OpenMP kd-tree range search

This approach implemented an almost exact copy of the *CPU kd-tree range search* mentioned above but used multi-threading to range search multiple PDB structures with this method at the same time.

3.2.3 Pre-processing algorithms

Extracting protein structure data from PDB files with ESBTL

The pre-processing index construction steps required protein structure data to be loaded into RAM from PDB files before the hashed index and kd-tree index construction could begin. The research thus used an open source library called ESBTL

[45]. ESBTL (Easy Structural Biology Template Library) is a lightweight C++ library that allows the handling of PDB data and provides a data structure suitable for geometric analysis and advanced constructions.

The implemented versions of this pre-processing step were:

- A CPU implementation
- An openMPI implementation which ran instances of the CPU implementation on multiple nodes
- An openMP implementation which multi-threaded the loops which appeared in the CPU implementations interactions with ESBTL

Constructing Hashed indexes

In this research, hashed indexes were used to locate all instances of select atom types within proteins being range searched. In kd-tree range searches only, all instances of one of the two atom types would be located with hashed indexes while in every brute force algorithm except the GPU brute force approach, hashed indexes would be used to get lists of both atom types. The implemented versions of this construction step were:

- A CPU hashed index construction algorithm
- An OpenMPI implementation which ran instances of the CPU implementation on multiple nodes
- An openMP implementation which multi-threaded some portions of the CPU implementation

Constructing kd-trees

The kd-tree construction method utilised for this research was a complex algorithm developed to match the criteria of kd-tree construction algorithms outlined in [61] by ***H. Samet***.

It was developed from scratch and is easily the most complex algorithm utilised in this research. The reason for designing the algorithm from scratch was to ensure

that the approach would be fully compliant with GPU architecture constraints by design when it came to parallelizing it.

The approach adopted was to mirror the construction process of a binary tree (essentially a 1 dimensional kd-tree) but to pre-sort the 3D data set into 3 connected sorted lists which would automatically remove placed data points from the other two lists when it was consumed in any of the three lists. This behaviour achieved three efficient results. The first was that the 3 sorted lists never needed to be re-sorted, the second was that the simple binary tree construction method could be utilised with the lists with very few modifications, and the third was that the placement of each data point in a given level of the kd-tree worked off completely separate data points – meaning that the construction of every level of the tree could be fully parallelized without causing bottlenecks or race conditions.

The algorithm is cleanly separable into 2 parts. The first is the construction of 3 sorted lists (one for each dimension that 3D data could be sorted by) and the second is a construction method which consumes those lists to generate a kd-tree.

Chapter 4 will explain at length how this algorithm was implemented. As to parallel implementations of the algorithm, the 5 following versions were compared:

- A CPU sorted lists construction method and a CPU sorted lists consumption method
- A GPU bitonic based sorted lists construction method and a GPU sorted lists consumption method.
- A CPU sorted lists construction method and a GPU sorted lists consumption method.
- A GPU sorted lists construction method and a CPU sorted lists consumption method.
- An openMPI implementation which ran instances of the pure CPU implementation across multiple nodes.
- An openMP implementation which multi-threaded large sections of the pure CPU implementation.

3.3 Results metrics

Measuring the results of the comparisons between the different parallel algorithms in this research mainly came down to the difference in absolute run time for each algorithm to complete the same range search request. This is because comparing completely different parallel architectures is, while a worthwhile pursuit, a non-standard comparison. OpenMP and OpenMPI are scaled in different ways and while the results presented hold true for the hardware utilised in this research, newer hardware or more extensive processing clusters could increase the performance these two parallel systems could provide. The same goes for using a more powerful GPU model should one be acquired.

As a side view to this, graphs were also provided to give a more visual view of what the total run time numbers represented.

Chapter 4

Resources and implementation details

This chapter lists the hardware, data, software packages and implemented algorithms which were gathered to execute the comparisons written in Chapter 3.

To cover these aspects of the project the relevant information has been split into 5 sections. The first section details the physical resources which were used to run these algorithms – which includes details on the compute cluster, the individual node this research was run on, and the specifications of the K20 Tesla GPU which was used to run the GPU algorithms. The second section lists the third party software which was used in this research. The third section provides details of the protein data which the algorithms were run upon, a brief description of the range search queries chosen and will provide links to the git repository where a copy of the full PDB file list utilised (as well as where all copies are the scripted algorithms) can be found. The fourth section details the specifics of how the GPU kd-tree construction algorithm works (in detail), and then details the pseudo-code and behaviour of all the other algorithms implemented. Finally the fifth section briefly describes the marshalling program which was built to standardise testing.

4.1 Hardware utilized for benchmarking the algorithms

Due to the size of the data sets being compared, the hardware requirements were put far above the capabilities of an average desktop computer – not only because of the limited processing power that standard desktops possess but also because standard desktops do not have sufficient RAM to hold the full data sets which would be used during benchmarking. These requirements were met by resources made available on the Wits Core Research Cluster.

For the duration of the trials being performed, a machine in the research cluster containing an NVIDIA GK110GL [Tesla K20Xm] was fully reserved. The machine has 32 GB of RAM, two Intel Xeon E5 2620 v2 @ 2.10GHz processors, each of which has 15MB L2 cache and 6 physical cores/12 hyper-threaded cores. The NVIDIA Tesla K20Xm has 2496 CUDA cores, 5GB GDDR5 memory and 208GB/sec bandwidth [52]. As the machine was not utilized by other members of the cluster for the duration of the trials, the performance of the Tesla GPU was not split when trial running this research's algorithms.

4.2 Software utilized during benchmarking

The CUDA programming platform was utilized to interface with the NVIDIA k20Xm Tesla GPU, and C++ was chosen as the projects main programming language due to its compatibility with CUDA. For the parallel CPU processing algorithms, the OpenMP standard was used to execute multi-threaded codes (and was compiled through the MPICH compiler) while the OpenMPI framework was used to execute distributed computing across several CPU cores on the cluster. In addition to these, an open source C++ library called ESBTL was utilised to extract protein chain coordinate data from the utilised PDB files.

The versions of these resources were as follows:

- CUDA: V 10.1.243
- G++: V 4.8.5
- mpiCC (MPICH): V 1.10.7
- OpenMP: V 3.1
- ESBTL: V 1.0 Beta

4.3 Data and query specifications

4.3.1 Selecting and obtaining extensive amino acid data sets from PDB files

The chosen source of protein data was an extensive cache of PDB files from the WorldWide protein database organisation which were stored in the Wits Core Research Cluster for general use. The list of the PDB files utilised can be found online at: `https://github.com/JoshuaSelvan/pdbProcessingResources`.

For constructing PDB structure data, only the names of the atoms and the Ångstroms coordinates of each atom are needed. While the other listed attributes within PDB files are not needed for the amino acid oriented algorithms which were enacted in this research, the level of detail described by these sections is useful to explain why secondary data sets often need to be constructed to be used in place of PDB libraries.

As PDB files contain such a wide selection of information on their housed protein structures, sorting through the wealth of irrelevant data within PDB files would add a needless processing overhead to any process being run on that data. For that reason, users often need to create secondary data structures storing only the data relevant to their experiments. Such secondary data structures reduce the memory and processing requirements imposed by the data sets they work with, making the extraction of data from PDB files an important aspect of many large protein database inquiries.

For the algorithms executed in this research, a sample consisting of 100000 PDB files (which amounted to over 76 gigabytes of data) was utilized. The experiments run dealt with protein chains up to 16192 atoms in length which encompassed 91000 of the 100000 PDB used. To load data from the PDB files into RAM during the execution of the C++/CUDA programs, an open source C++ library called ESBTL was used to extract protein coordinate data from the PDB files.

While PDB files usually hold a single protein structure, there are instances when they hold multiple protein structures and/or other structures. For this reason the research will not talk about processing protein structures but rather PDB structures – as those are what are loaded from PDB files.

4.3.2 Selecting atom pairs for the protein range searches

In the experiments run there were three atom pairs which were used to test range searches which resulted in low, medium and high quantities of results (and a fourth search for non-existent atoms which resulted in no results). The way these atom pairs were chosen was by experimentally searching for atoms which appeared in PDB structures in low, medium and large numbers. The atom pairs chosen were C1'-C2', OG-N and CB-CG – representing low occurrence, medium occurrence and high occurrence atom pairs respectively. While the presence of individual atoms varies greatly from file to file, Table 4.1 shows the average occurrences of these combinations in random file samples (containing protein chains of length 512 to 16384) and random samples of medium length chains (containing protein chains of length 4096 to 8192). In the experiments performed, atom pairs were considered successful matches if both required atoms were present and within a 4 Ångstroms range of each other.

Average occurrence count of atoms in randomly selected PDB structures			
PDB structure length range:	C1'- C2'	OG - N	CB - CG
512 to 16384 atoms long	5 : 5	32 : 601	525 : 350
4096 to 8192 atoms long	6 : 6	43 : 717	601 : 490

Table 4.1: The above table gives a visualisation of the average appearance counts of atoms in randomly selected PDB structures. The averages were drawn from 4 samples of 5000 randomly selected PDB files each. Using batches of larger files resulted in more instances of the atoms being present per file.

4.4 Algorithm implementations

This section has two halves. The first half is dedicated to explaining how the developed kd-tree construction method (in both CPU and GPU environments) functions, as well as how the kd-tree range searches work (in both the CPU and GPU environments). This was deemed necessary due to the complexity of the two GPU algorithms, and viewing the CPU version first, helps with the explanations.

The second half is dedicated to providing pseudo-codes of all the algorithms implemented in this research (both range searches and pre-processing phases)

The actual C++ implementations of these algorithms are available online in a github repository at the address: `https://github.com/JoshuaSelvan/pdbProcessingResources`.

4.5 Kd-tree Construction Algorithm

This section explains how the kd-tree construction algorithm functions, and how the algorithm is parallelized in a GPU environment, and then gives reasons why several GPU performance issues are not incurred by this algorithm.

The approach for showing how the kd-tree construction algorithm works will be to provide a simple implementation of a binary-tree construction algorithm and then make 2 consecutive modifications to that binary-tree construction algorithm to convert it to a 3 dimensional kd-tree construction algorithm - these changes can be used to extend the kd-tree construction method to any number of dimensions.

4.5.1 Constructing a binary-tree

Binary trees split data with a single dimension into a tree structure. Here three pseudo-codes will be provided to construct such a tree and then a small visual demonstration will be displayed.

The first two algorithms form a quick sort function which sorts a list of elements in ascending order. The third algorithm is a pseudo-code which takes in a sorted list of data points and places the middle most datapoint as the current node in the binary tree. A quick note here is that there is a step in the binary node algorithm where the nodes are counted and then divided by 2 instead of just selecting the median of the list. This is done intentionally as it helps explain how the kd-tree construction

algorithm works later.

Algorithm 1: Partition

Input: data point list A, lowerbound hi, upperbound hi

Result: orders elements in A and returns i

```
pivot := A[hi]; i := lo;
for j := lo to hi - 1 do
    if A[j] < pivot then then
        if i != j then then
            swap A[i] with A[j];
        i := i + 1;
    swap A[i] with A[hi]; return i;
```

Algorithm 2: Quicksort

Input: data point list A, lowerbound p, upperbound r

Result: Once fully completed, data points will be ordered

```
if (p < r) then
    q ← Partition(A,p,r);
    Quicksort(A,p,q);
    Quicksort(A,q+1,r);
```

Algorithm 3: Binary-tree node placing algorithm

Input: List of remaining viable data points: *currentList*

Result: Placed binary tree node and up to two recursive calls back to this algoritm if child nodes are required

```
counter → 0;
for each data point in currentList do
    counter → counter + 1;
NewNode → currentList[counter/2];
Place NewNode in binary Tree;
if data points left of currentList[counter/2] > 0 then
    Execute algorithm 3 on data points left of currentList[counter/2]
if data points right of currentList[counter/2] > 0 then
    Execute algorithm 3 on data points right of currentList[counter/2]
```

As an example to the above, Figure 4.1 shows how the binary tree forms from its 1 dimensional data point list. From the visual one can see how trivial the process of sorting 1 dimensional data is.

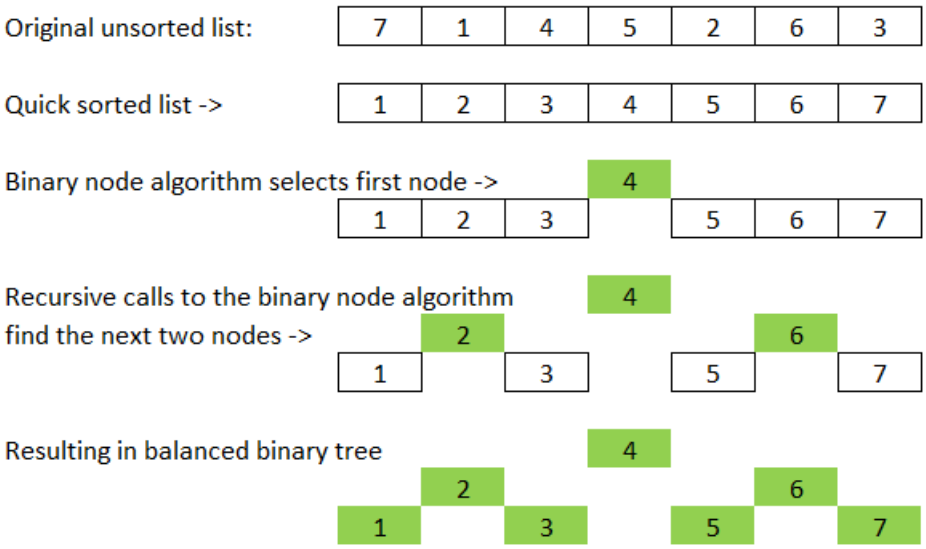


Figure 4.1: A standard binary tree construction

4.5.2 Converting the binary-tree construction algorithms to kd-tree construction algorithms

Updating the two algorithms which make up the quick sort process

We will now update the partition algorithm (seen above as Algorithm 1) to sort a 3-dimensional data list with respect to one of its three dimensions (which will be the values in one of the list's three columns). We will also update this partition algorithm to sort a numeric list containing the numbers 1 to n in the same order which the list containing the dimensional data is being sorted. The updated partition algorithm can be seen below in Algorithm 4.

This updated partition algorithm (Algorithm 4) is then called with the updated quick sort algorithm seen below as Algorithm 5. This updated algorithm takes in the additional parameters of an ordered numeric list of numbers 1 to n as well as a dimension number to dictate which of the data list's columns the data will be sorted by.

These modifications achieve two ends. Firstly the algorithm can now sort data sets which contain more than one coordinate per entry. Secondly, the sorted numeric list outputted by the combination of Algorithm 4 and Algorithm 5 is a reference of where each data point was originally from in the original unsorted list. For example, the original numeric list value in position 2 was the value 2 and seeing as this value

Algorithm 4: Partition which also sorts second numeric array

Input: 3 dimensional data point list A, ordered list B ([1,2... ,n]), lowerbound lo, upperbound hi, currentDimension d (which will be a constant from 0 to 2 depending on which of the 3 dimensions the algorithm is set to sort by)

Result: Orders elements in List A with respect to the dimension d (which will be the entries in one of A's columns),the movements made in List A are mirrored by the elements in the same positions in list B and return the value i

```
pivot := A[d][hi]; i := lo;
for j := lo to (hi - 1) do
    if A[d][j] < pivot then then
        if i != j then then
            swap A[d][i] with A[d][j];
            swap B[i] with B[j];
        i := i + 1;
    swap A[d][i] with A[d][hi];
    swap B[i] with B[hi];
    return i;
```

Algorithm 5: Quicksort which also sorts second numeric array

Input: data point list A, ordered list B, lowerbound p , upperbound r, currentDimension d

Result: Once fully completed, data points will be ordered in list A and list B will show where all of list A's elements originated from

```
if (p < r) then
    q ← Partition(A,p,r,d);
    Quicksort(A,B,p,q,d);
    Quicksort(A,B,q+1,r,d);
```

was moved around in the exact same steps as the data points in the data point list, whatever position the data point being sorted landed up in, if the value in the same position in the numeric list is 2, the data point originated from position 2 originally – this relationship allows us to locate the original positions of all the data points.

The sorted numeric list is an artefact that will be used to construct the kd-tree, the ordered list of data points are not actually used during construction. We will refer to this set of numeric lists as *Originated From* indexes as the data they represent is where each sorted data element originated from in the original unsorted list. A demonstration of this algorithm's effect can be seen in Figure 4.2.

Original Coords list

	x	y	z
Pos: 1	2	6	8
Pos: 2	5	9	9
Pos: 3	8	2	10
Pos: 4	3	5	7
Pos: 5	6	10	3
Pos: 6	9	7	4
Pos: 7	0	8	6

Ascending numeric list

1
2
3
4
5
6
7

The two arrays after being sorted with respect to List A's x coordinates

	x	y	z
Pos: 1	0	8	6
Pos: 2	2	6	8
Pos: 3	3	5	7
Pos: 4	5	9	9
Pos: 5	6	10	3
Pos: 6	8	2	10
Pos: 7	9	7	4

XO
7
1
4
2
5
3
6

Figure 4.2: A list of 3 dimensional data sorted by Algorithm 5 with respect to the x dimension

Constructing artefacts to handle delete operations across multiple *Originated From* indexes

Using Algorithm 5, we construct an *Originated From* index with respect to how the data list would be sorted with respect to the x (as seen in Figure 4.2), y and z dimensions of the data list. The three *Originated From* indexes produced (and the sorted lists they correspond to) can be seen below in Figure 4.3.

Coords array sorted by the x,y and z dimension with OriginatedFrom indexes moved in the same ways

	Sorted by X			XO	Sorted by Y			YO	Sorted by Z			ZO
Pos: 1	0	8	6	7	8	2	10	3	6	10	3	5
Pos: 2	2	6	8	1	3	5	7	4	9	7	4	6
Pos: 3	3	5	7	4	2	6	8	1	0	8	6	7
Pos: 4	5	9	9	2	9	7	4	6	3	5	7	4
Pos: 5	6	10	3	5	0	8	6	7	2	6	8	1
Pos: 6	8	2	10	3	5	9	9	2	5	9	9	2
Pos: 7	9	7	4	6	6	10	3	5	8	2	10	3

Figure 4.3: The original list sorted with respect to each 3 dimensions with the numeric lists next to them showing where each of the sorted elements came from in the original list.

To use these three separate *Originated From* indexes together we need a way to delete the same data point being represented in all three lists when a data point is deleted in one of them. For example, if a data point is selected from the x-*Originated From* index to be placed in the kd-tree, we will remove that data point from the x-*Originated From* index to signify it has been consumed – that same data point will still be present in the y-*Originated From* and z-*Originated From* indexes unless we create a way to locate that same data point in the other indexes however.

The information needed to create a location directory is already present in the three *Originated From* indexes. As the *Originated From* indexes point to where data points originated from before being sorted, we can just swap the positions and values of every entry in an *Originated From* index to get an index which now shows where each entry in the original unsorted list will go when sorted with respect to that dimension. We will refer to these forward directing indexes as *Moved To* indexes. Using the *Originated From* index constructed in Figure 4.2, this process is shown in Figure 4.4.

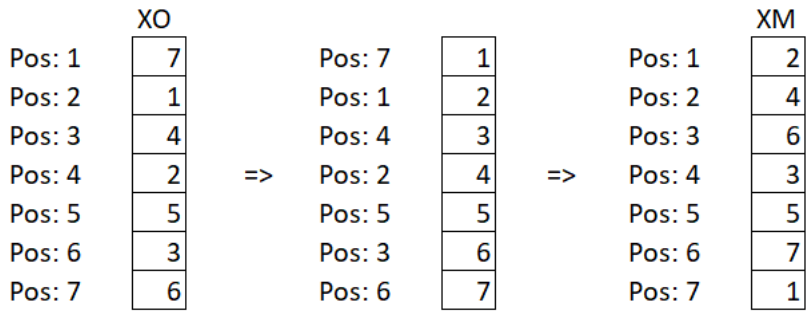
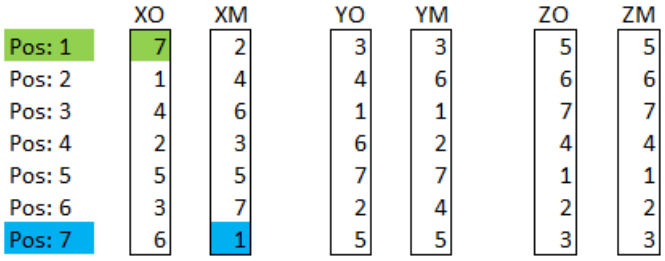


Figure 4.4: The positions and values of the *Originated From* index are swapped and then the pairs are reordered so the positions are ascending again. This results in an index which shows where the unsorted entries of the 3 dimensional list will land up if sorted with respect to the chosen dimension. We will refer to these forward directing indexes as *Moved To* indexes.

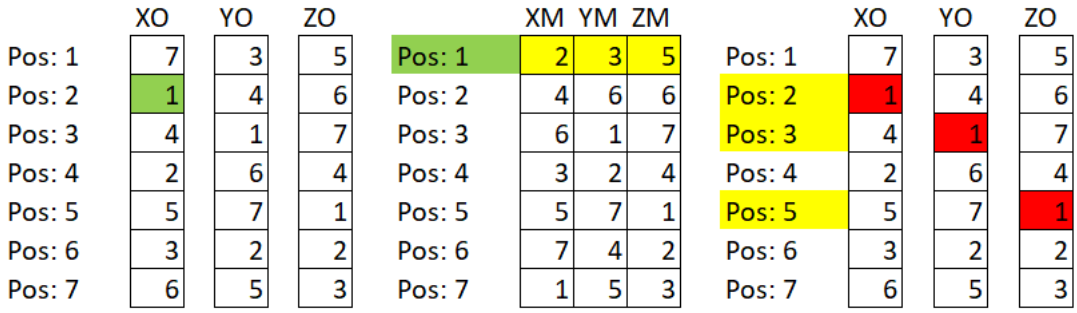
The value in having a set of these *Moved To* indexes is that if we combine them together into an n by d array (where n is the number of entries in the data list and d is the number of indexes), we then have in each row of the array a reference to where each of the unsorted data entries will go to in all of the *Originated From* indexes. Using this relation we can delete a single data entry in all three of the *Originated From* indexes by going to the row in the combined *Moved To* index, going to the positions in all 3 of the *Originated From* indexes referenced in that row of the *Moved To* index and removing the value found in all three of those positions.

This process is shown below in Figure 4.5 where the relationship between the 3 *Originated From* and *Moved To* indexes is displayed and then in figure 4.6 an example is given of how an entry is selected in one *Originated From* index and the data in the *Moved To* indexes are used to find where to remove that entry in all 3 *Originated From* indexes.



	XO	XM	YO	YM	ZO	ZM
Pos: 1	7	2	3	3	5	5
Pos: 2	1	4	4	6	6	6
Pos: 3	4	6	1	1	7	7
Pos: 4	2	3	6	2	4	4
Pos: 5	5	5	7	7	1	1
Pos: 6	3	7	2	4	2	2
Pos: 7	6	1	5	5	3	3

Figure 4.5: The original list sorted with respect to each 3 dimensions with the numeric lists next to them showing where each of the sorted elements came from in the original list.



	XO	YO	ZO
Pos: 1	7	3	5
Pos: 2	1	4	6
Pos: 3	4	1	7
Pos: 4	2	6	4
Pos: 5	5	7	1
Pos: 6	3	2	2
Pos: 7	6	5	3

	XM	YM	ZM
Pos: 1	2	3	5
Pos: 2	4	6	6
Pos: 3	6	1	7
Pos: 4	3	2	4
Pos: 5	5	7	1
Pos: 6	7	4	2
Pos: 7	1	5	3

	XO	YO	ZO
Pos: 1	7	3	5
Pos: 2	1	4	6
Pos: 3	4	1	7
Pos: 4	2	6	4
Pos: 5	5	7	1
Pos: 6	3	2	2
Pos: 7	6	5	3

Figure 4.6: In this figure we see three consecutive stages. First the entry in position 2 of the XO *Originated From* index is selected to be removed and we take the value held within this position (1). Second we select the position in the combined *Moved To* index equal to the value we obtained in the previous step and we now take the 3 values in this position of the *Moved To* index. Third we delete the entries in the three *Originated From* indexes of positions equal to the values we got from the previous step. As the values are the same in all the deleted positions we know we have deleted the same entry across all 3 listings - and we did so without having to search through the *Originated From* indexes to find it

Consuming the *Originated From* and *Moved To* indexes to construct a kd-tree

Once the *Originated From* and *Moved To* indexes explained in the previous two sections have been constructed we have all the artefacts required to construct the kd-tree. We do not even need to reference a copy of the original data list as the only relevant data from the original data list (the entry ordering with respect to each dimension) is contained within the *Originated From* indexes in a more readily accessible manner.

How this construction algorithm works is that the *Originated From* indexes are used to represent the remaining data points to be placed in the kd-tree while the *Moved To* indexes will be used to delete all references to individual data entries once they are placed in the kd-tree. The only additional complexity is that every time a data entry is placed in the kd-tree, we need to inform the recursive algorithms constructing that node's two child nodes of the data points they no longer have access to – as when recursively constructing child nodes in a kd-tree, the left child will only have access to the lesser data entries (with respect to that dimension) while the right child node will only have access to data entries which are greater than the placed entry (with respect to that dimension). As the *Originated From* indexes come pre-sorted and are never updated or reshuffled, this requirement is met by passing to each child recursion the upper and lower bounds of the x, y and z *Originated From* indexes that those child recursions still have access to – these bounds never overlap with the bounds of other child recursions which will later make the algorithm safe for parallelization as long as only one level of the kd-tree is constructed at a time.

With this explanation of the additional considerations of the kd-tree construction algorithm (seen in Algorithm 8) detailed, the kd-tree construction algorithm works similarly to the binary tree construction algorithm (seen in Algorithm 3) – with the two additional requirements that one needs to count how many of the entries in the current set are unconsumed and relevant to the child's current bounds (and then place the middle most criteria matching entry) and then locate and delete all references to the placed entry before beginning the next two recursive calls to the algorithm.

While the workings of this algorithm are explained in the previous section, the algorithm to delete all instances of a data entry across the three *Originated From* indexes is detailed in Algorithm 6. This algorithm will be called whenever a data point is successfully placed in the kd-tree.

Algorithm 6: DeleteFromAllDimensions

Input: position reference r, 2D *OriginatedFrom* array A, 2D *MovedTo* array B
Result: Set data point to null in all 3 listings

```
A[0][B[0][r]] = Null;
A[1][B[1][r]] = Null;
A[2][B[2][r]] = Null;
```

Finally, the algorithm used to count all valid data points in the set provided to each instance of the recursive kd-tree node construction algorithm are given below in Algorithm 7. This algorithm accepts a single data point as input and then checks that the data point has not been consumed in any of the three *Originated From* indexes – essentially performing the first 2 of 3 steps demonstrated in Figure 4.6. This algorithm will be called for each of the data points being checked for each kd-tree node placement and is the key to the efficiency of this approach. While performing 3 lookup operations per data point may look expensive on their own, this approach lets the same 3 sorted lists be reused to construct the entire kd-tree without having to re-sort them. Without this sort of approach, the data points being passed to every child node would have to be re-sorted with respect to the current dimension in every call of the recursive algorithm – which would have a far higher compute cost overall.

Algorithm 7: DataPointStillValid

Input: position reference r, 2D *MovedTo* array B, currentLowerAndUpperBounds L
Result: Check if data point is within valid bounds and if it hasn't already been consumed

```
if r != Null then
    if L[0][0] <= B[0][r] <= L[0][1] then
        if L[1][0] <= B[1][r] <= L[1][1] then
            if L[2][0] <= B[2][r] <= L[2][1] then
                Return True;
Return False;
```

Full kd-tree construction example

As an example of this algorithm in action, the set of *Originated From* indexes and *Moved To* indexes listed in Figure 4.5 will be used to construct a kd-tree from beginning to end. Before we start, look at Figure 4.7. This is an example of the resulting kd-tree that is achieved when re-sorting the remaining data subsets for

Algorithm 8: kd-tree node placing algorithm

Input: 2D *OriginatedFrom* array A, 2D *MovedTo* array B, 2D currentLowerAndUpperBounds L, current dimension d

Result: Placed kd-tree node and up to two recursive calls to the function if child nodes are required

```
counter → 0;
medianPos → 0;
medianCounter → 0;
for each dataPoint in OriginatedFrom[d] between L[d][0] and L[d][1] do
    if DataPointStillValid[dataPoint[d],B,L] then
        counter → counter + 1;
for each dataPoint in OriginatedFrom[d] between L[d][0] and L[d][1] do
    if DataPointStillValid[dataPoint,B,L] then
        medianCounter → medianCounter + 1
    if medianCounter == counter/2 then
        Break from loop;
    medianPos → medianPos + 1;
NewNode → OriginatedFrom[d][L[d][0]+medianPos];
Place NewNode in kd-tree Tree;
DeleteFromAllDimensions[NewNode];
if data points left of OriginatedFrom[d][L[d][0]+medianPos] > 0 then
    Re-execute algorithm on data points with L[d][1] set to
     OriginatedFrom[d][L[d][0]+medianPos] with next dimension (d+1)
if data points right of OriginatedFrom[d][L[d][0]+medianPos] > 0 then
    Re-execute algorithm on data points with L[d][0] set to
     OriginatedFrom[d][L[d][0]+medianPos] with next dimension (d+1)
```

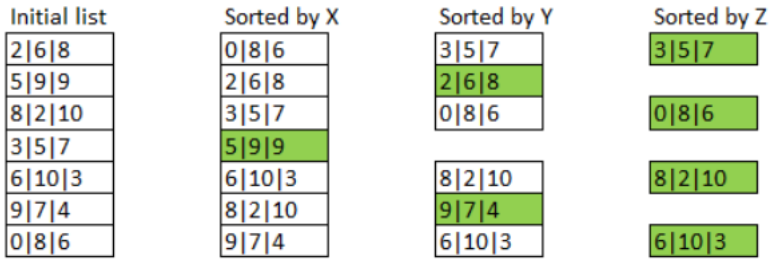


Figure 4.7: A kd-tree constructed manually by placing the middle most element and then re-sorting the remaining data points at each level of the tree

every recursive call of the kd-tree construction algorithm. In the example to follow, the same tree will be achieved without any re-sorting of data subsets.

To place the first node we are going to start with the x dimension. As no data points have been consumed we are still working with the entirety of the XO list and we will simply insert the middle most entry in the list. As the value being inserted into the tree is a (2), we then proceed to check which positions need to be removed in the three *Originated From* indexes by finding their positions in the *Moved To* indexes –

	XO	YO	ZO
Pos: 1	7	3	5
Pos: 2	1	4	6
Pos: 3	4	1	7
Pos: 4	2	6	4
Pos: 5	5	7	1
Pos: 6	3	2	2
Pos: 7	6	5	3

	XM	YM	ZM
Pos: 1	2	3	5
Pos: 2	4	6	6
Pos: 3	6	1	7
Pos: 4	3	2	4
Pos: 5	5	7	1
Pos: 6	7	4	2
Pos: 7	1	5	3

	XO	YO	ZO
Pos: 1	7	3	5
Pos: 2	1	4	6
Pos: 3	4	1	7
Pos: 4	2	6	4
Pos: 5	5	7	1
Pos: 6	3	2	2
Pos: 7	6	5	3

Figure 4.8: Inserting the root node of the kd-tree.

in the (2) position. This order of events can be seen in Figure 4.8. When running the recursive call for the left and right child node however, we will pass along the information that only the 1-3 x positions are considered valid for the left child node while only the 5-7 x positions are considered valid for the right child node.

Next we will insert the left child node of the root node. As this is happening on the second level of the kd-tree we will sort with respect to the y dimension (so the YO index is our reference point). As there are no bounds on the YO index yet we check every entry in the index which hasn't been marked as removed to see if they fall inside the kd-tree branch's bounds. As can be seen below in Figure 4.9, the entries highlighted blue meet the bound criteria set by the XO index while the three yellow entries do not. We thus insert the middle most blue entry into the kd-tree and mark that entry as consumed in all three *Originated From* indexes.

Placing the right child node of the root node happens in the exact same way (except that we count the entries highlighted yellow in Figure 4.9 as valid instead of the blue highlighted ones). The insertion steps can be seen in Figure 4.10. In Figure 4.10 the entry which was inserted as left child node of the root has already been removed – and while it seems like this could have interfered with the operations of the right child node, each data set dealt with in this method is distinct (so the removed node would have been counted as invalid anyway).

Next we will place the left child of the left child of the root node. As can be seen in Figure 4.11, this node inherits a bounding box on both the XO and YO indexes. In Figure 4.11, the entry which is marked yellow fell completely outside both boundary boxes while the entries marked as purple met the boundary conditions for one of the two boundary boxes but failed for the other. The entry marked green was the only entry left which met the placing criteria and so it would be placed in the kd-tree.

First we count how many of the remaining coordinates in the valid YO bounds are valid in all dimensions. In this case only the three data points coloured blue meet the XO criteria. We select the middle most valid data point in the YO list (1).

	XO	YO	ZO
Pos: 1	7	3	5
Pos: 2	1	4	6
Pos: 3	4	1	7
Pos: 4	2	6	4
Pos: 5	5	7	1
Pos: 6	3	2	2
Pos: 7	6	5	3

	XM	YM	ZM
Pos: 1	2	3	5
Pos: 2	4	6	6
Pos: 3	6	1	7
Pos: 4	3	2	4
Pos: 5	5	7	1
Pos: 6	7	4	2
Pos: 7	1	5	3

	XO	YO	ZO
Pos: 1	7	3	5
Pos: 2	1	4	6
Pos: 3	4	1	7
Pos: 4	2	6	4
Pos: 5	5	7	1
Pos: 6	3	2	2
Pos: 7	6	5	3

Next, we insert (1) into the kd-tree, locate all instances of (1) in the XO, YO and ZO lists and then mark them as removed.

	XO	YO	ZO
Pos: 1	7	3	5
Pos: 2	1	4	6
Pos: 3	4	1	7
Pos: 4	2	6	4
Pos: 5	5	7	1
Pos: 6	3	2	2
Pos: 7	6	5	3

	XM	YM	ZM
Pos: 1	2	3	5
Pos: 2	4	6	6
Pos: 3	6	1	7
Pos: 4	3	2	4
Pos: 5	5	7	1
Pos: 6	7	4	2
Pos: 7	1	5	3

	XO	YO	ZO
Pos: 1	7	3	5
Pos: 2	1	4	6
Pos: 3	4	1	7
Pos: 4	2	6	4
Pos: 5	5	7	1
Pos: 6	3	2	2
Pos: 7	6	5	3

Figure 4.9: Inserting the left child node of the root node.

	XO	YO	ZO
Pos: 1	7	3	5
Pos: 2	1	4	6
Pos: 3	4	1	7
Pos: 4	2	6	4
Pos: 5	5	7	1
Pos: 6	3	2	2
Pos: 7	6	5	3

	XM	YM	ZM
Pos: 1	2	3	5
Pos: 2	4	6	6
Pos: 3	6	1	7
Pos: 4	3	2	4
Pos: 5	5	7	1
Pos: 6	7	4	2
Pos: 7	1	5	3

	XO	YO	ZO
Pos: 1	7	3	5
Pos: 2	1	4	6
Pos: 3	4	1	7
Pos: 4	2	6	4
Pos: 5	5	7	1
Pos: 6	3	2	2
Pos: 7	6	5	3

Figure 4.10: Inserting the right child node of the root node.

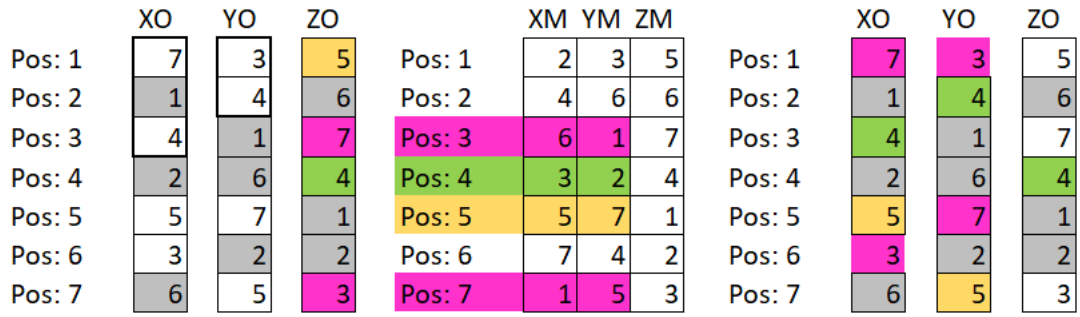


Figure 4.11: Inserting the left child node of the left child node of the root node.

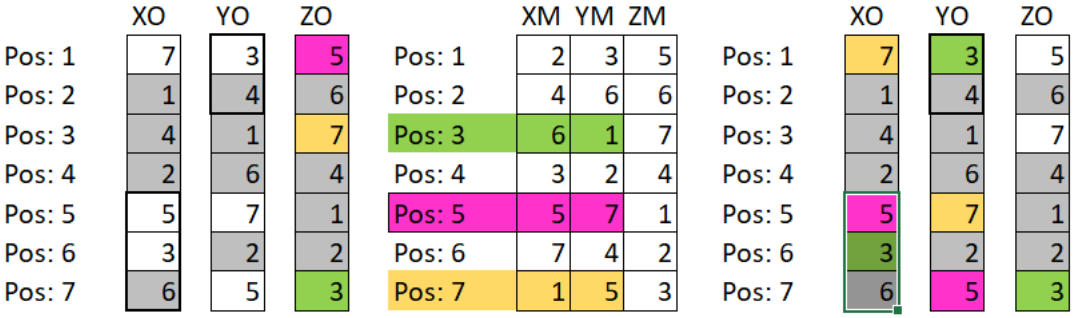


Figure 4.12: Inserting the left child node of the right child node of the root node.

As one final step to demonstrate we will look at the left child of the right child of the root node. As can be seen in Figure 4.12, there can be greyed out (removed) entries in the active boundary boxes – this does not lead to any data conflicts, just less computations to make.

Once this placing process has been completed we land up with the kd-tree displayed in Figure 4.13. This kd-tree contains references to the data points held in the original unsorted data list so we need to replace the references with the actual values to get the resulting kd-tree to look identical to the one generated earlier in Figure 4.7.

Converting the sequential kd-tree construction algorithm to a GPU algorithm

As noted before, there are two separate halves to the kd-tree construction algorithm. The first one needs to generate the *Originated From* and *Moved To* indexes while the second consumes those arrays to build the kd-tree. When including a GPU in

Figure 4.13: In this picture we replace the reference numbers inserted into the kd-tree with the original values found in those reference positions (which can be viewed in Figure 4.3. This results in the correctly formed kd-tree.

the approach, the first point is that the GPU can be used for both halves of the algorithm or just the first or second algorithm – meaning you can get the CPU to do one component while the GPU does the other.

If one wishes to generate the *Originated From* and *Moved To* indexes with the GPU then one uses the following modified bitonic sort function (which has been modified to update a second incremental array (of 1 to n) in the same way the CPU quicksort was modified to do.

When constructing the kd-tree on the GPU with a set of *Originated From* and *Moved To* indexes, one is nearly running the same algorithms as the CPU version with two modifications. As all but the newest GPUs cannot handle recursion (and unrestrained recursion is ill advised anyway), the algorithm needs to be modified to store pending child operations in an array instead of executing them and then resubmitting those tasks for computation when the current batch is done – which leads to the behaviour of generating a single layer of the kd-tree per round. The second modification is the need for a scheduler algorithm to tell the GPU code when to kick off each level of computation. The algorithms are given below:

Algorithm 9: GPU kd-tree construction scheduler

Input: 2D *OriginatedFrom* array A, 2D *MovedTo* array B, pendingTasks array T
Result: executes all currently pending kd-tree node algorithms in parallel, waits for them to finish then queues the next batch

4.6 List of algorithm implementations

This section lists the pseudo-codes of all the algorithms implemented for this research (a list of which has already been given in Chapter 3), some brief discussions on their

Algorithm 10: GPU based kd-tree node placing algorithm

Input: 2D *OriginatedFrom* array A, 2D *MovedTo* array B, 2D currentLowerAndUpperBounds L, current dimension d, pendingTasks array T

Result: Placed kd-tree node and up to two tasks s logged in the pendingTasks array for future exectution

```
counter → 0;
medianPos → 0;
medianCounter → 0;
for each dataPoint in OriginatedFrom[d] between L[d][0] and L[d][1] do
    if DataPointStillValid[dataPoint[d],B,L] then
        counter → counter + 1;
for each dataPoint in OriginatedFrom[d] between L[d][0] and L[d][1] do
    if DataPointStillValid[dataPoint,B,L] then
        medianCounter → medianCounter + 1
    if medianCounter == counter/2 then
        Break from loop;
    medianPos → medianPos + 1;
NewNode → OriginatedFrom[d][L[d][0]+medianPos];
Place NewNode in kd-tree Tree;
DeleteFromAllDimensions[NewNode];
if data points left of OriginatedFrom[d][L[d][0]+medianPos] > 0 then
    Set L[d][1] to OriginatedFrom[d][L[d][0]+medianPos], set dimension to next
     dimension and then append task to T with current A, B, L and d values
if data points right of OriginatedFrom[d][L[d][0]+medianPos] > 0 then
    Set L[d][0] to OriginatedFrom[d][L[d][0]+medianPos] , set dimension to next
     dimension and then append task to T with current A, B, L and d values
```

expected memory usage and expected run times. A hashed index CPU range search algorithm is also included - as openMPI runs this algorithm repetitively and the openMP algorithm was based off it. Before the algorithms a brief symbols section is given as many of the symbols are used repetitively through the algorithms.

4.6.1 Key: Descriptive symbols

As the following discussions on range searching techniques will all contain discussions on their expected runtimes under favourable and unfavourable situations, this section will provide some abbreviations of common terms which will appear in the mathematical formula on a regular basis:

Symbols:

- N: The number of atoms within the current amino acid chain
- T: The number of different types of atoms in the current amino acid chain
- n: The number of sought type 1 atoms within the current amino acid chain
- m: The number of sought type 2 atoms within the current amino acid chain
- t: The number of unrequested atom types searched through before finding a sought atom type
- $AtomA$: The place holder name of the first type of atoms which are being sought for proximity matches
- $AtomB$: The place holder name of the first type of atoms which are being sought for proximity matches
- $Chain$: The amino acid chain which composes the PDB structure being searched
- i: A counter value used for loops
- j: A secondary counter value used for loops
- P: The number of GPU threads being run in parallel in any given GPU process
- p: The current GPU thread (out of a total of P threads) being looked at
- k A constant coefficient
- Av Total number of PDB structures being held
- $[+], [-], [\times], [\div]$ Mathematical operations. These symbols are used in this section when discussing the operational costs of running different algorithms.
- $[C]$ A comparison operation. This symbol is used in this section when discussing the operational costs of running different algorithms.

4.6.2 Required memory space for range searching resources

There are three resources which are used by the range searches detailed below:

- The list of unsorted PDB structures (Av)
- Hashed indexes for all the PDB structures
- Kd-trees for all the PDB structures

Not all range searches use all of these and some brute force range searches only use the unsorted PDB structures themselves. In this section a brief overview is given of how much memory space each resource would require on RAM if it is utilised.

Required memory space for unsorted PDB structures

Each unsorted PDB structure is a list of data points consisting of 3D coordinates (x, y and z) and an atom type. In this research the atom types were stored as smallint instead of strings (with a reference table to convert back) and the x, y and z coordinates were converted from floats to ints (for GPU compatibility) by multiplying by 100. This put the memory required for each data point at ($smallint + 3int$) and the total memory requirement at $(smallint + 3int) * N * Av$ which can be shortened to $Av * (N * k)$.

Required memory space for hashed indexes

The hashed index used in this research was a hashtable which converted atom types into numbers and then hashed by the unique atom numbers present in the PDB structure being looked at (T) and contained pointers to lists containing the locations of all instances of each type of atom within the structure each hash table represented ($N * k$). This put the storage space required to hold the hashed index at $T + N * k$ for a single protein and $Av * (T + N * k)$ for the entire protein set.

Required memory space for kd-trees

The kd-trees used in these algorithms contained all the atoms within the protein they represented in a standard binary tree format. The only issue with this representation is that as the kd-trees constructed in this research were stored in data structures with standardized entry spaces, the storage format required that the very bottom row of the kd-tree be stored in its entirety regardless of whether it was filled with atoms or mostly empty cells – as storing up to 50000 kd-trees in a non-uniform manner tended to complicate the act of locating them (especially when they were moved to the GPU). As the bottom row of a kd-tree contains twice the entries of the previous level (1,2,4,8 etc.), this meant that for a protein of length N, the storage space required to house it could be anywhere between N and $2N$ – which put the total storage space of the complete kd-tree set at $Av * 2N$.

As use of kd-trees in the utilised algorithms still required the original unsorted PDB structures (as the kd-trees contained links back to the original PDB structures to fetch spatial coordinates and atom types), as well as the hashed index to find all instances of $AtomA$, kd-tree approaches required the memory space of all 3 resources putting their actual memory requirement at $Av * (T + N * k + 1) + 2N \times k$ on the CPU side (as GPU side memory requirements were more dynamic).

4.6.3 CPU hashed index range search

Concept

This is a normal CPU range search algorithm which uses hashed indexes to fetch the lists of all instances of $AtomAs$ and $AtomBs$ for the current PDB structure then does a brute force comparison on the distance between all possible instances of the $AtomAs$ and $AtomBs$ to check for proximity matches.

4.6.4 Pseudo-code implementation

Algorithm 11: CPU Hashed Index Range Search

Input: Hashed index of all PDB structures H, sought $AtomA$, sought $AtomB$
Result: List of all $AtomA - AtomB$ pairs of required proximity
Retrieve list of all $AtomA$ from H;
Retrieve list of all $AtomB$ from H;
while *i is less then the number of found $AtomAs$* **do**
  while *j is less then the number of found $AtomBs$* **do**
    if *The ith $AtomA$ and the jth $AtomB$ are within the required proximity* **then**
      Add this $AtomA$–$AtomB$ pair to the list of found proximity matches;

Expected completion time

A hashed index has an expected return time of $O(1)$ so when seeking a list off all n instances of $atomA$, it will take $O(n)$ steps to fetch them all. For the comparison half of the algorithm, one is performing $n * m$ calculations. This puts the total operations to $n + n * m$, which approximates to a $O(n^2)$ run time.

4.6.5 CPU kd-tree range search

General concept

The CPU kd-tree range search first uses a hashed index to find all instances of the *AtomAs* within the PDB structure being processed and then finds all requirement meeting proximity matches with *AtomBs* by only descending down branches within the associated kd-tree which can contain proximity matches of sufficient closeness – and checking each node reached to see if it is the sought *AtomB* type.

Pseudo-code

Algorithm 12: CPU Kd-tree Range Search

Input: Kd-tree for current PDB structure H, sought $AtomA$, sought $AtomB$
Result: List of all $AtomA - AtomB$ pairs in required proximity
Retrieve list of all $AtomA$ from H;
Set current kd-tree node to the root node;
Begin recursive part of algorithm:
Get the coordinates of the current kd-tree node's atom;
if *the coordinates of the current kd-tree node's atom is in the current AtomA′s required proximity* **then**
    if *the current kd-tree node's atom is a AtomB* **then**
        Record proximity match;
    Repeat algorithm for left and right child node;
else
    for *both child nodes* **do**
        if *Distance between the current kd-tree node's atom and the current AtomA is less then the distance between the ChildNode and current AtomA in current kd-tree dimension, and the distance between ChildNode and current AtomA is greater then the required proximity* **then**
            Then do not call recursive algorithm on child node;
        else
            Recursively call algorithm on child node;

Expected completion time

For explanation's sake, it is best to first explain the runtime complexities of performing a basic search through a kd-tree before discussing the complexities of performing a range search through a kd-tree – as range searches are more complicated. When searching a kd-tree, the worst case completion time is $O(n)$ – which occurs when the

entire kd-tree is searched before finding the sought entry [38]. In evenly distributed kd-trees however, the worst completion time will only be $O(\log(n))$ as no single data point within an evenly distributed kd-tree can be stored in a position which takes more than $\log(n)$ moves away from the the tree's root to reach. Range searches are more costly than normal search operations however as you are not looking for an exact match within the kd-tree but rather a viable spatial area represented by the descent options chosen in the kd-tree.

To understand the difference, one has to delve into more specific comparisons than the previous range search algorithms. For a normal kd-tree search, one performs a single value comparison between the target atom and the kd-tree's current splitting value at every branch in the tree. This amounts to a single comparison's worth of operations $O(1)$ per split in the kd-tree. In a range search however, it is not enough to know if a given branch in the kd-tree is moving closer to the epicentre of your search target. Rather you need to check if the tree is moving towards the viable area around your target epicentre, is moving away from your target epicentre or has moved far enough away in a given direction that the kd-tree will not be able to return to the viable area anywhere down the current branch.

At every node in the kd-tree that a range search algorithm hits, the expensive Pythagorean distance operations (consisting of three additions and three multiplications (3[+]+3[×])) needs to be computed to determine if the current node is within range of the current type *AtomA*. On top of this, checking whether the algorithm should descend along either child node requires 4 comparison operations for each child (4[=]). This means that every node in a kd-tree hit by a range search for a single *AtomA* will incur the processing costs of $3[+] + 3[\times] + 8[C]$ and if there is a potential proximity match the algorithm will check to see if the current atom is an *AtomB*. The total operations associated with navigating each split in a kd-tree for a range search is thus $(n+n\times X\times(3[+]+3[\times]+8[C]+n\times Q))$ where X is the average number of nodes searched in a kd-tree and Q is the number of close proximity nodes encountered which need to be checked with Pythagorean algorithms.

The number of variables involved makes determining the average and worst case scenarios of kd-tree range searches hard to formulate. The general theme however is that each movement along a branch in the kd-tree is an expensive operation in range searching – which means that in the best case scenario, very few branches would have to be traversed to find all potential proximity pairs while in the worst case scenario, every branch would need to be traversed, resulting in a total cost of $N*(n+n*X*(3[+]+3[\times]+8[C]+n*Q))$ per *AtomA* being parsed through the

kd-tree.

In terms of big-O notation, the average running time of a kd-tree range search is determined by how many instances of the sought *atomA* and *atomB* there are and how often any viable pair is within the required proximity of each other. If there is a single *atomA* and a single *atomB* then the Big-O notation will equate to between $O(\log(n))$ [38] and $O(n)$ if every node in the tree is within the viable distance of the selected *atomA*. In the worst case however, where there are $\frac{n}{2}$ *atomAs* and $\frac{n}{2}$ *atomBs* and all are within the required proximity, then the worst case scenario would be $O(n^2)$. As the expected Big-O notation is hard to pin down, for the rest of this chapter the expected complexity for this range search will be referred to as $O(n\log(n))$ as this falls between the unlikely extremes of $O(n)$ and $O(n^2)$.

4.6.6 OpenMPI hashed index range search

General concept

The OpenMPI approach performs range searches which are algorithmically identical to the *CPU hashed index range search* but allows multiple instances of this algorithm to be run on separate processing nodes. This is known as an embarrassingly parallel approach.

As multiple processing nodes are being utilised in parallel, the full list of PDB structures which need to be range searched are split into equal length sub-lists (with each processing node receiving an equal length sub-list of PDB files to search) and once each processing node has finished processing its load the results are compiled back together.

4.6.7 Pseudo-code

Expected completion time

Each processing node will express the same completion times as the base *CPU hashed index range search* – namely $O(n^2)$. As the total list of PDB structures to be searched is split across Y processing nodes, the total execution time tends to be about $O(n^2/Y)$. With the resources utilised by this project, the highest number of

Algorithm 13: OpenMPI Hashed Index Range Search

Input: Hashed index of all proteins H, sought $AtomA$, sought $AtomB$, List of PDB structures to process Av, number of openMPI nodes $numNodes$

Result: List of all $AtomA - AtomB$ pairs of required proximity

→ Divide List of PDB structures (Av) into $numNodes$ sublists;

→ Pass a sublist to each of the $numNodes$ openMPI nodes;

→ Each openMPI node processes its list of PDB structures with the *CPU hashed index range search* algorithm (11);

→ Results from each openMPI node are recompiles on the launching node;

processing nodes which were used together were the twelve processing cores on the two Intel chips on the reserved machine.

4.6.8 OpenMPI kd-tree range search

General concept

The OpenMPI approach performs range searches which are algorithmically identical to the *CPU kd-tree range search* but allows multiple instances of this algorithm to be run on separate processing nodes.

As multiple processing nodes are being utilised in parallel, the full list of PDB structures which need to be range searched are split into equal length sub-lists (with each processing node receiving an equal length sub-list of PDB files to search) and once each processing node has finished processing its load the results are compiled back together.

4.6.9 Pseudo-code

Algorithm 14: OpenMPI Kd-tree Range Search

Input: Kd-trees of all PDB structures H, sought $AtomA$, sought $AtomB$, List of PDB structures to process Av, number of openMPI nodes $numNodes$ (which is equal to the number of available physical nodes)

Result: List of all $AtomA - -AtomB$ pairs of required proximity

→ Divide List of PDB structures (Av) into $numNodes$ sublists;

→ Pass a sublist to each of the $numNodes$ openMPI nodes;

→ Each openMPI node processes its list of PDB structures with the *CPU kd-tree range search* algorithm (12);

→ Results from each openMPI node are recompiled on the launching node;

Expected completion time

Each processing node will express the same completion times as the base *CPU Kd-tree range search* – which we approximating to $O(n \log(n))$. As the total list of PDB structures to be searched is split across Y independent processing nodes however, the total execution time tends to be about $O(\frac{n}{Y} \log(\frac{n}{Y}))$.

4.6.10 GPU brute force range search

General concept

This is a simple brute force approach which relies on the GPU's fast parallel processing speed to find instances of the sought atoms instead of using a hashed index to quickly find all instances of the required atoms. This approach is more viable on the GPU as it is able to simultaneously look at up to 512 separate atoms at the same time per processing block. Once all instances of *AtomAs* and *AtomBs* have been found, a thread is allocated to each found instance of *AtomA* and each thread then compares its *AtomA* to all the found *AtomBs*. Of note is that while these comparisons are being performed, other PDB structures are being silently loaded onto the GPU in the background. This overlapping of processing and loading allows GPU algorithms to eliminate some of the overhead of data transfers from the computer's RAM to GPU general memory.

Pseudo-code

Algorithm 15: GPU Brute Force - locate both atom type lists

Input: unsorted PDB structure list U, sought *AtomA*, sought *AtomB*
Result: List of *AtomAs* and a list of *AtomBs*
Initialize P threads;
for *each of the P threads* **do**
    while i *is less then* $(N + P) \div P$ **do**
        If the ith position in *Chain* equals *AtomA* Add ith position to global list of *AtomAs*; Ifthe ith position in *Chain* equals *AtomB* Add ith position to global list of *AtomBs*;
;

Algorithm 16: GPU Brute Force - range search

Input: List of *AtomAs*, List of *AtomBs*
Result: List of all $AtomA--AtomB$ pairs in required proximity
Initialize P threads;
for *each of the P threads* **do**
    Assign a unique instance from the *AtomAs* list to the thread;
    while i *is less then* $(AtomBs + P) \div P$ **do**
        if *the* $i \times P + pth$ *position in the AtomBs list is within sufficient proximity of the threads AtomA to warrant a proximity match* **then**
            Add *AtomA–AtomB* pair to the global proximity matches list;
;

Expected Completion time

Like the original CPU brute force algorithm, this GPU approach will look at every position in the PDB structure being processed. Unlike the basic CPU variant, this GPU variant is able to look at P separate atoms in the protein atom chain at the same time. This means that with a basic brute force approach, where the total comparisons are split evenly between all GPU threads, the fixed runtime for finding all instances of $AtomA$ and $AtomB$ is $2N/P$.

Having found all instances of $AtomA$ and $AtomB$, the total operations required to check all possible proximity matches is: $n * m/P$. This puts the total time required to perform a GPU brute force range search at $(2N + n * m)/P$ – which results in a running time of $O(n^2/P)$. As P can scale very high (to 512 and higher) and the largest proteins looked at in this research were 16384 atoms in length, P was often found to be larger than N, which would bring the notation down to $O(n)$ in many cases.

Required memory space for storage

This algorithm requires space on both the CPU for housing the PDB structures loaded from secondary storage into RAM as well as storage on the GPU to house the current PDB structures which the GPU is tasked with processing. As it is faster to locate the sought atoms within PDB structures on the GPU with a brute force approach then it is to fetch pre-constructed lists of those atoms from hashed indexes with either the CPU or the GPU, the GPU only requires space to house the PDB data sets it is tasked with processing – and does not require space for kd-tree constructs or hashed indexes. On the CPU this requires the aforementioned $Av * N * k$ total memory space to house the base PDB structure data. The amount

of memory space utilized (or required) on the GPU is more fluid but basically comes down to how much global memory space your GPU has – as you usually want to use all of it.

As the GPU has a finite global memory stash, and the time taken to send data to and from the GPU is not instantaneous, you want your GPU to always be performing computations on one set of data while the GPU simultaneously loads the next set of data onto the GPU for processing. This means that ideally the amount of space used by the GPU for storing protein data is all the storage space it has but with the caveat that just under half of the storage space is reserved for loading the next batch of data into general purpose GPU memory while the other half of the storage space will be filled with the set of protein data currently being processed by the GPU – with the GPU switching between the two halves of memory so the consumed protein data half has the time to load its next protein data batch while the other half of memory is being processed.

4.6.11 GPU hybrid approach (CPU side atomA present checking)

General concept

This approach executes exactly the same GPU algorithm as the *GPU brute force range search* approach above except that the hashed index used in the *CPU hashed index range search* is used to decide if each PDB structure should be processed on the GPU or discarded without processing. By using the hashed index on the CPU to first check if any proximity matches are possible (ascertained by checking whether the required atom type is present within the PDB structure to begin with), the overhead of moving result-empty PDB structures to the GPU can be avoided at a fraction of the processing cost – although this slight cost is still paid on PDB structures which need to be processed on the GPU (although there are usually fewer PDB structures which require processing then PDB structures which can be discarded)

Pseudo-code

Expected completion time

This technique centres around avoiding expensive GPU side processing by performing cheap CPU side processing to identify large amounts of data sets which cannot

Algorithm 17: GPU/CPU Hybrid approach

Input: unsorted PDB structure list U, hashed indexes of all proteins T,sought $AtomA$, sought $AtomB$

Result: List of all $AtomA - AtomB$ pairs of required proximity

→ Step 0: Execute the *CPU hashed index range search* (Algorithm 11) to check if there is at least 1 instance of $AtomA$ in the current PDB file and stop if there is not;

→ Step 1: Execute the *GPU Brute force atom location algorithm* (Algorithm 15) with inputs unsorted PDB structure list U, sought $AtomA$ and sought $AtomB$;

→ Step 2:Execute the *GPU Brute Force comparison algorithm* (Algorithm 16) with the list of $AtomA$s and $AtomB$s found in Step 1;

provide any positive matches – and thus are not worth processing on the GPU. As detailed in the previous subsection, the cost of performing a GPU side brute force search is $((2N + n * m) * P)$. Add to this the previously detailed CPU side cost of locating all the instances of a single atom type with a look up structure of $T + n$ and one gets the processing time of $((T + n) + ((2N + n * m)/P))$. This processing time only applies to PDB structures which are processed on the GPU however. If after spending the $T + n$ time required to come to the CPU side conclusion that there are no $AtomAs$ in the current PDB structures, then the expensive GPU half of the computations can be skipped entirely. This means that over the entire set of PDB structures being processed, the cost of processing result-empty PDB structures is a cheap $T + n$ while only the PDB structures with the potential to bear results suffer the heavier processing cost of $((T + n) + ((2N + n * m)/P))$ – which tends to be cheaper than the normal CPU match finding cost of $n * m$ when the counts of n and m are high.

Required memory space for storage

This algorithm requires space on the CPU for housing the protein data loaded from secondary storage into RAM, space on the CPU for the secondary-index structures this algorithm uses to pre-validate proteins as well as storage on the GPU to house the current protein data which the GPU is tasked with processing. As GPUs tend to locate sought atoms by brute force faster than they can fetch exact lists of the sought atoms from secondary-index structures, the GPU only requires space to house the protein data sets it is tasked with processing – and does not require space for secondary constructs. Despite the fact that the GPU does not use secondary-index structures for range searching purposes, these secondary indexes still need to be stored on the CPU to help determine which proteins are worth sending to the GPU

for processing. The CPU thus requires a total storage space of $Av*(T+N*(K+1))$ to house its required structures and data while the GPU's storage requirements behave exactly the same as they do in the *GPU Brute Force* subsection on memory usage in section of 4.6.10.

4.6.12 GPU kd-tree range search

General concept

This kd-tree range search locates the list of *AtomAs* in the same way as the *GPU brute force range search*, by getting large numbers of threads to each look at individual atoms in the current protein chain at the same time. The more interesting side of the GPU conversion of a CPU kd-tree occurs when trying to perform range searches on the kd-tree.

As a GPU thread can only follow a single branch of the kd-tree at a time (without causing massive slowdowns with each algorithmic divergence encountered) the approach decided upon was to instruct each running thread to follow a single branch and if any emergent branches needed to be searched, store the details of those unprocessed searches in a backlog. At regular intervals, the algorithm would pause and every search being stored in the backlog would be given its own thread and then be run in parallel with the threads already being processed. This circumvents the issues of being unable to start new threads mid-process and provides a work-around to the issue of recursion. The issue with this approach is that until the first backlog of unprocessed searches are loaded into threads, there are very few initial searches running on a given kd-tree – which means a cycle of kd-tree searching needs to pass before the actual parallel processing happens in earnest.

As the GPU threads interacting with the kd-tree never modify data inside the tree no race conditions are caused by having multiple threads performing searches in parallel. When compiling found atom pairs into a single list, GPU specific atomic operations are used to make sure there is no data corruption during insertion.

Algorithm 18: GPU kd tree - generate list of AtomAs

Input: unsorted PDB structure

Result: List of all *AtomAs*

→ A copy of the protein is loaded onto the GPU;

→ The GPU splits the process of performing a brute force lookup over $512 \times X$ threads, returning the list of located *AtomAs* to general purpose GPU memory;

Algorithm 19: CPU side controller for GPU kd tree range search

Result: List of all $AtomA - AtomB$ pairs of required proximity

→ Place several initial search requests into the GPU search backlog - these include the coordinates of the *AtomAs* being compared against and the position in the kd-tree to begin searching from (the root node in this case);

→ Begin a GPU RangeSearch for every search Request in the GPU backlog;

→ If the GPU backlog is not empty repeat from step 2, if the backlog is empty retrieve all proximity matches from the GPU side results list;

Algorithm 20: GPU kd-tree range search step for single thread/tree node

Result: List of all $AtomA - AtomB$ pairs of required proximity

→ Let D be the maximum depth of the current kd-tree (14 for proteins of 8192-16384 atom length)

for *D cycles or until the list of threads to run is empty* **do**
- **if** *thread's AtomA is within required proximity of current kd-tree node's atom* **then**
  - Add the right child Node and *AtomA* pair to search backlog;
  - **if** *current Node's atom is the Required AtomB* **then**
    - add pair to results list ;
  - Change current kd-tree node to left child node and go to next cycle loop;
- **else**
  - Set ValidChildNodesCount to 2 **for** *both child Nodes* **do**
    - **if** *child node is outside required proximity and further from required proximity then current node* **then**
      - mark child node as invalid;
      - decrease ValidChildNodesCount by 1;
  - **if** *both child nodes are valid and ValidChildNodesCount=2* **then**
    - add right child node to backlog;
  - **if** *at least 1 child is valid and ValidChildNodesCount is greater then 0* **then**
    - **if** *left child is valid* **then**
      - change current Node to left child node;
      - go to next cycle loop;
    - **else**
      - change current Node to right child node;
      - go to next cycle loop;

Pseudo-code

Required memory space for storage

This algorithm requires space on both the CPU for housing the protein data loaded from secondary storage into RAM as well as storage on the GPU to house the current protein data which the GPU is tasked with processing. While GPUs do not use secondary-indexes to find sought atoms, implementing kd-tree algorithms on the GPU means that the kd-trees do need to be stored on the CPU and then passed (along with the actual protein data) to the GPU for processing. This means that on the CPU, storage space is required to store the protein data and the constructed kd-trees (but unlike the CPU spatial index approaches, no space is needed to store the secondary-index structures) which can be approximated as $Av \times (T + N \times k) + 2N \times k$. As to the amount of space required for storage on the GPU, it behaves very similarly to the space requirements of the GPU brute force algorithm but is more space limited due to the kd-trees which need to be loaded alongside the protein chains for processing.

As with the GPU brute force approach, half of the GPU's global memory will be reserved for loading the data for the next batch of proteins to be processed while the other half of memory will be used to process the previous batch of proteins loaded into the GPU's memory. As the kd-trees could require up to twice as much memory to store as the proteins they represent, this meant that each protein loaded to GPU global memory effectively required 3 times as much storage space as proteins loaded to the GPU for brute force processing – which decreased the size of the protein batches which would be loadable (and thus processable) on any given GPU to a third of what could be managed by a brute force implementation.

Avoiding GPU processing bottlenecks

This kd-tree implementation does not suffer from any bottlenecks besides the fact that it starts processing only a single route down the tree (as it will not start processing multiple until that first thread has identified other valid paths to descend etc).

To list a few of the bottlenecks it avoids, this algorithm does not suffer having multiple threads trying to access the same data as each thread descends a different branch of the tree and all nodes refer to distinct locations in memory. If any threads

were to try to access the same memory location (through an extension to the algorithm), as they are performing read operations only and not write operations, the slow down is not too bad. Additionally as data in the tree is never changed by searching it, there are no race conditions. Finally the potential race condition of multiple threads trying to add proximity matches to the results list was resolved by having an atomic operation resolve which memory position each thread would write to – which stopped multiple threads trying to write to the same spot by accident.

4.6.13 Implementing the pre-processing algorithms

When executing range searches, there are a set of resources which need to first be loaded into the memory of the executing device before the range search can begin. For the simplest range search approaches, this means that the data being processed needs to be present in a readable format to begin executing – but for more complex range search techniques, secondary structures need to be constructed from the unprocessed data before the algorithm can execute.

In the context of range searching PDB structures, there are between one and three pre-processing steps which may need to be completed for any given algorithm. As mentioned in the last paragraph, each algorithm needs to have the PDB structures loaded into fast access memory (RAM) and converted into a usable format. Range searches which use indexing techniques additionally need to construct hashed indexes before the scan executes and kd-tree range searches need to construct both a hashed index as well as a kd-tree before they can execute.

In this subsection a brief overview will be given of what occurs in the first two of these pre-processing phases as well as a description of how they were implemented and the ways in which parallelization was worked into them for comparison purposes – the third possible pre-processing phase (kd-tree construction) was described earlier in this chapter due to its complexity.

Uploading (and unpacking) PDB structures

The first pre-processing phase is both the most time consuming phase of processing PDB structures and required by all range searching implementation. All range search techniques require that the PDB structures be moved into the processing system's RAM before the algorithm can execute and as the speed of accessing secondary

storage is very slow (and PDB structures sets can be very large) the act of loading the PDB structures into memory is often more time consuming than performing the range searches upon it. Additionally, PDB structures are often stored in bloated formats (as protein files tend to store large amounts of data irrelevant to any singular experiment) and thus the required PDB structures usually need to be extracted from the storage files before the range search algorithms can be used.

Extracting PDB structures from secondary storage files such as PDB files can be a demanding process as many PDB files contain errors which cause simple text parsing approaches to break down or return inaccurate results. As developing a PDB extracting algorithm was not in the scope of the project, a lightweight open source library called ESBTL was used to extract the PDB structures from PDB files in secondary storage into an easily accessible format in RAM.

The variations implemented in this research were the base sequential upload and extraction process, a parallelized OpenMPI version (which performed exact copies of the base sequential upload and extraction process across separate processing nodes) and an OpenMP approach which was very similar to the base sequential upload and extraction process but allowed for repetitive operation calls to the open-source library to be done in parallel through multi-threading. As the open-Source extraction library was written for use with the CPU and the inefficiency of sending bulk unprocessed data to a GPU (and the lack of meaningful ways to parallelize a text file parsing script within the confines of a GPU environment) no GPU version of this script was implemented.

Constructing hashed indexes

The second pre-processing phase is that of creating hashed indexes of atoms within the loaded PDB structures – as such indexes are required by all range search algorithms which are not pure brute force approaches. Indexes are structures which allow users to quickly navigate to instances of sought data point types within large structures – which refers to finding all instances of a sought amino acid within an amino acid chain in the context of the experiments done.

For the experiments in this research, a custom hashed index was created which consisted of a simple hash-map to specify the type of atom being sought and if that atom type existed in the current protein chain, would then link to a list of all instances of that atom within the current protein chain. The worst case construction

time for this custom index was worked out to be $O(n)$ and the expected fetch time from the index was $O(1)+numberOfInstancesBeingFetched$.

Like the loading implementations in the previous sub-section, this custom algorithm was coded as a base CPU sequential implementation, an OpenMPI implementation which copied the base CPU implementation but split the total protein set to process across multiple nodes and an OpenMP approach which allowed repetitive operations to be parallelized with multi-threading. In this instance, no GPU hashed index construction algorithm was developed as the time it takes GPUs to locate atoms in unsorted data is faster than GPUs take to fetch the same data from the relevant indexes – plus the construction time of these indexes are fast enough that passing the relevant data to and back from the GPUs would mitigate any processing gains made by computing them there to begin with.

4.7 Making algorithms data race safe

In the above section a number of algorithms were listed for range searching protein data and most of them are designed to run using parallel techniques. In this section a brief overview will first be given on how data races were avoided when constructing kd-trees and hashed indexes and then an overview will be given on why the range searching algorithms were safe from data races.

4.7.1 Avoiding data races in parallel lookup structure construction

Kd-tree construction

When constructing kd-trees, a single value from the remaining data set will be placed in a node within the kd-tree – after this, the remaining data set will be split in half with one half of the data points being reserved for the left child node and the other half of the data set being reserved for the right child node. As the subsets of the total data set split into different child nodes do not interact during the kd-tree construction process, so long as the algorithm dealing with each node being calculated in the kd-tree only manipulates its own data set, there are no race conditions which can occur. This was the approach taken for ensuring there would be no race conditions when constructing kd-trees on a GPU – which was implemented

by restricting the GPU to calculating a single row of the kd-tree on each cycle to ensure no data set overlaps.

For kd-tree construction with openMPI, each kd-tree was constructed in its entirety by an assigned processor and then sent back to be stored in pre-allocated, standard sized memory slots – which prevented any of the data from ever overlapping.

For kd-tree construction with openMP, the only situations where multi-threading returned tangible improvements were for subsections of the algorithm dealing with sorting arrays or performing counts on arrays. Both of these operations were basic enough that standardised approaches could be found to avoid race conditions.

Hashed index construction

These indexes were created exclusively on the CPU side. For openMPI, the results from different processors were just assigned pre-allocated memory slots to be placed in once complete to prevent any clashes. For openMP, care was taken to limit the parallel sections to tasks which did not touch the same data. The only instance where openMP could suffer race conditions was the event that two of the same type of data points were to be placed in the hashed index at the same time. This race condition was avoided by requiring entries to the hashed table to be appended sequentially once their insertion location had been chosen.

4.7.2 Race conditions do not occur in range searches

When performing range searches with existing kd-trees or hashed indexes, there are only data fetch/read operations from shared resources (no writing to them) and in the kd-trees the resources being read are always distinct. Because the data being worked on is not changed by the range search algorithm, there are no race conditions which could cause operations being run in parallel to corrupt or change the data another thread is busy using. The only instance where race conditions could occur would be if different processes or threads tried to write results from different proteins being worked on to the same memory location – but this event is outside the bounds of the range search algorithms themselves and are trivial to plan for.

4.8 Marshalling data hardware, software and protein data within a single program

4.8.1 Developing a testing program

Due to the high number of algorithms being compared, and the large pre-processing time associated with loading large batches of protein data into RAM, it became necessary to develop a program which could marshal protein data (and pre-processed secondary structures) for repetitive use without the need to reload or redo the pre-processing phases associated with each range search algorithm.

To this end the decision was made to develop a program which would hold a copy of all the loaded protein data, spatial-secondary structures and non-spatial secondary structures for repetitive use, allowing any CPU or GPU inclusive algorithms to immediately start their range search phase by accessing all required resources from the implemented (RAM based) platform instead of secondary storage.

The approach chosen was to write a C++ based program which would accept as input a file containing the locations of all the PDB files to be processed, the types of range searches to be run and the format of output to be used for gathered results - although for this research the actual results were only recorded in several trial runs to confirm each algorithm returned the same output, after which only the completion times were outputted for comparison.

For completeness sake this section details how the program operates - with emphasis on how the first half (loading and pre-processing data) and the second half (executing any number of range searches on the loaded data) is enabled in the program.

Pre-processing pipeline

When the program is run, the program initially assigns enough space to cater for the desired datasets specified in the program's configuration file. The program then creates three additional resources: a structure containing all the stored protein data in RAM, a hashtable containing numeric references for all the atom types being stored in the protein structure, and a set of kd-trees and secondary search structures.

First the program loads all the specified protein data into searchable structures with proteins of similar size being separated into different sets. The protein data

is loaded from PDB files through the use of an open source library called ESBTL. Using ESBTL, the structural data of all the atoms within each protein are saved to storage structures. While the coordinate data is taken straight from ESBTL, the names of each atom are converted to numeric representations decided upon by the program. This is achieved by creating a hashtable of all the atom names loaded by ESBTL throughout the loading phase and then referring to the atom name by a number representing its position within the hashtable. Users never see any of the numeric representations of the atom names but by using numbers instead of strings the range search phases can be run notably faster (and the GPU range searches are unable to process strings, meaning the conversion to numerable representations was necessary).

Once all the protein data have been loaded, a kd-tree and secondary search structure is constructed for each protein stored by the program. Both the kd-trees and secondary lookup structures are constructed solely on the CPU. While variations were implemented where the kd-trees were constructed by CPU/GPU hybrid codes, the default setting was chosen as CPU construction so the program would work on machines without an available GPU - although the runtime comparisons are recorded in the next section).

For the sake of modularity, all the protein data, kd-trees, and secondary search structures are housed within a single object, while the name-to-number hashtable is stored in a second object and the range search settings in a third. These three objects thus formed the standard resources to be referenced by all the range search algorithms made available within the program.

Execution pipeline

After the pre-processing phase, the program checks the configuration object to see what range searches the program is scheduled to run and then proceeds to sequentially execute each separate range-search job with the settings specified for each select range search. The program allows for any number of range searches to be run with any type or combination of the range searches enabled by the program.

Chapter 5

Results

This section covers the results obtained when implementing the algorithms described in chapter 4 to meet the research objectives described in chapter 3. As such this chapter is split into 3 sections to cover the main objectives, secondary objectives and side experiments which were of relevance but not central to the research. The format of this chapter is to list tables and graphs of the associated experiments and offer brief commentary on any notable results within the data.

The first section will cover the two main objectives of the research. The first objective being to compare GPU algorithm performance characteristics to other parallel algorithms at protein range searches, while the second objective is to try to determine what sort of scenarios GPU algorithms perform better or worse in.

The second section will cover the results of different parallel platforms in performing the pre-processing stages of extracting data from PDB files, creating hashed indexes of PDB structures and constructing kd-trees of PDB structures.

The third section will cover the process of finding the optimal settings to use for the openMP and openMPI algorithms to run against the GPU algorithms.

Of note is that the runtimes that are displayed throughout this chapter are the averages of 4 identical runs of each algorithm with the exact same parameters – to help average the performance of how each algorithm behaved with the parallel setups.

5.1 Main Objectives

5.1.1 Comparing absolute run time of parallel algorithms on randomly selected PDB structures

This section deals with the main experiment of comparing the best implementations of the different parallel algorithms at processing large batches of randomly selected PDB structures. As certain openMPI and openMP algorithms performed better with smaller node and thread counts in certain low intensity searches, multiple versions of these algorithms appear in the results tables.

This data is displayed across Tables 5.1 to 5.3 and shows the impact on total search time incurred by searching for atom proximity matches of atoms with different occurrence frequencies. This was done to show the relationship between the computational complexity of the process being attempted and the total processing time required by different parallel approaches to complete those tasks.

This set of tables documents the required run time of an atom pair which does not exist, the atom pair OG - N (which on average results in 635 relevant atoms per file) and the atom pair CB - CG (which on average results in 875 relevant atoms per files).

Range Search Completion time:				
Atom Pair: Non-existent atom pair				
Required atom pair proximity: 4 Ångstroms				
Unit of measurement: Seconds (s)				
Unsorted File Lengths				
Expected frequency of sought Atoms: 0 expected instances per file				
Number of PDB structures:	12 500	25 000	37 500	50 000
CPU Brute Force	0.39	0.79	1.59	1.79
CPU Hashed Index	0.02	0.03	0.05	0.07
CPU Kd-tree	0.01	0.02	0.03	0.35
GPU Brute Force	3.19	5.76	9.40	12.24
Hybrid Brute Force	0.17	0.22	0.64	0.81
GPU Kd-tree	2.19	3.79	6.07	9.13
OpenMPI -4 processes (Hashed Index)	0.01	0.01	0.07	0.35
OpenMPI -12 processes (Hashed Index)	0.02	0.02	0.02	2.88
OpenMPI -4 processes (kd-tree)	0.01	0.06	0.01	0.55
OpenMP run with 16 threads: Hashed Index force	12.05	25.75	34.57	52.30
OpenMP run with 16 threads: Kd-tree	12.41	27.04	33.91	102.95

Table 5.1: Processing times of range searches which searched for pairs of non-existent atoms (ensuring zero results) executed through different range search implementations. Results are displayed in seconds.

Range Search Completion time:				
Atom Pair: OG-N				
Required atom pair proximity: 4 Ångstroms				
Unit of measurement: Seconds (s)				
Unsorted File Lengths				
Expected frequency of sought Atoms: 635 expected instances per file				
Number of PDB structures:	12 500	25 000	37 500	50 000
CPU Brute Force	24.98	51.07	96.32	113.62
CPU Hashed Index	24.29	49.63	104.97	136.87
CPU Kd-tree	11.68	17.80	31.49	52.03
GPU Brute Force	16.22	31.76	49.88	66.75
Hybrid Brute Force	16.03	32.98	49.88	67.06
GPU Kd-tree	12.66	25.09	39.99	55.77
OpenMPI with 4 processes (Hashed Index)	7.04	15.65	23.42	49.60
OpenMPI with 12 processes (CPU secondary structure brute force)	2.64	4.75	7.47	19.4
OpenMPI with 4 processes (Kd-tree)	2.47	5.64	10.46	45.82
OpenMP run with 16 threads: Hashed Index	14.21	30.83	39.78	63.34
OpenMP run with 16 threads: Kd-tree	13.52	34.05	39.74	493.27

Table 5.2: Processing times of medium input intensity range searches executed through different range search implementations measured in seconds

Range Search Completion time:				
Atom Pair: CB-CG				
Required atom pair proximity: 4 Ångstroms				
Unit of measurement: Seconds (s)				
Unsorted File Lengths				
Expected frequency of sought Atoms: 875 expected instances per file				
Number of PDB structures:	12 500	25 000	37 500	50 000
CPU Brute Force	207.10	420.73	635.67	1011.51
CPU Hashed Index	202.83	413.25	905.50	1235.95
CPU Kd-tree	227.61	287.87	460.30	616.86
GPU Brute Force	17.17	32.18	76.41	139.31
Hybrid Brute Force	15.72	32.32	49.30	65.76
GPU Kd-tree	14.78	30.21	55.04	313.39
OpenMPI with 4 processes (Hashed Index)	57.16	117.09	180.01	263.53
OpenMPI with 12 processes (Hashed Index)	20.34	40.97	64.24	110.03
OpenMPI with 4 processes (Kd-tree)	39.15	82.54	130.23	196.34
OpenMP run with 16 threads: Hashed Index	30.02	62.99	90.30	146.16
OpenMP run with 16 threads: Kd-tree	26.07	54.50	95.52	442.30

Table 5.3: Processing times of high result range searches executed through different range search implementations measured in seconds

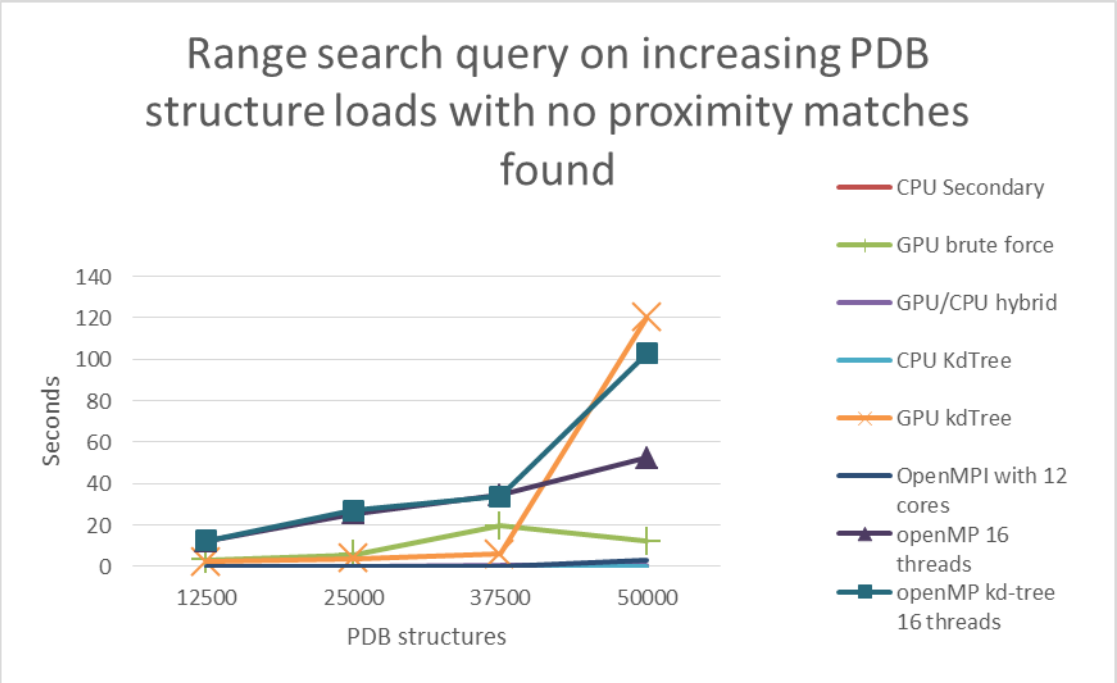


Figure 5.1: Visualization of Table 5.1's results. In this graph one sees a search for atom pairs which are not present in the PDB structures being processed – meaning the minimum amount of processing possible is being done in each search. While it's hard to see in the figure (as most of the algorithms take close to zero time to complete) all the sequential CPU algorithms, the two lower process OpenMPI algorithms and the CPU/GPU hybrid approach take under a second to complete at the 50000 PDB structure mark. The GPU and OpenMP brute force approaches take up to 12 and 52 seconds however while the GPU kd-tree and OpenMP kd-tree approaches take up 120 seconds to complete. As the algorithms which complete very slowly in this no results range search are the best performing range searches in the most compute intensive range search however (Table 5.3), the implication is that these types of parallelism carry a fixed overhead cost per structure searched which only becomes efficient to pay once a certain compute requirement threshold per file has been reached.

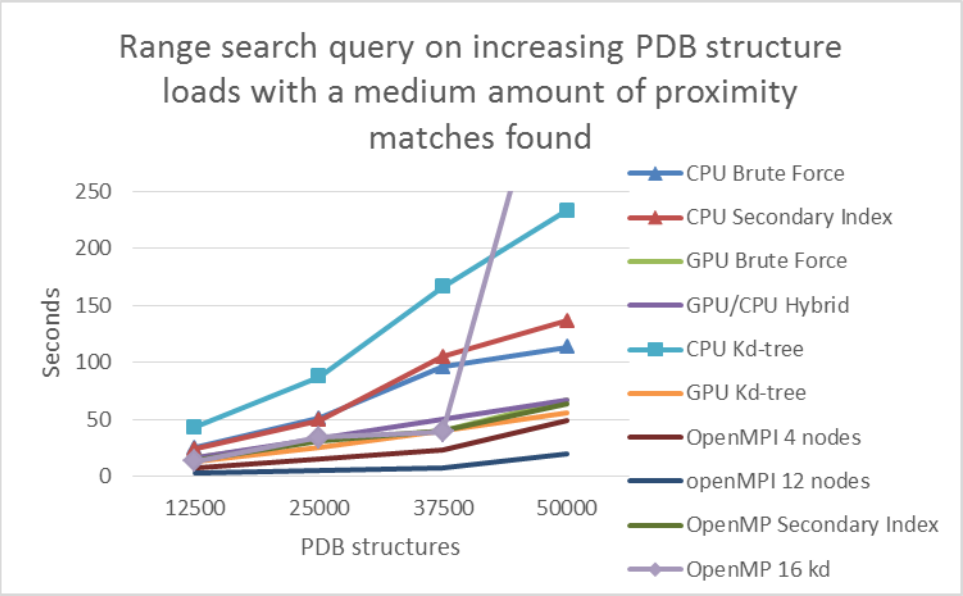


Figure 5.2: Visualization of Table 5.2's values. In this table an atom pair of medium occurrence is being sought out and one can see that while all of the algorithms take more time to process than they did with the zero result count jobs in figure 5.1, both CPU approaches and the OpenMP kd-tree approach are beginning to take much longer to process than the other parallel approaches. Unlike the zero result search in figure 5.1, the GPU kd-tree approach is no longer an inefficient algorithm in comparison to the others.

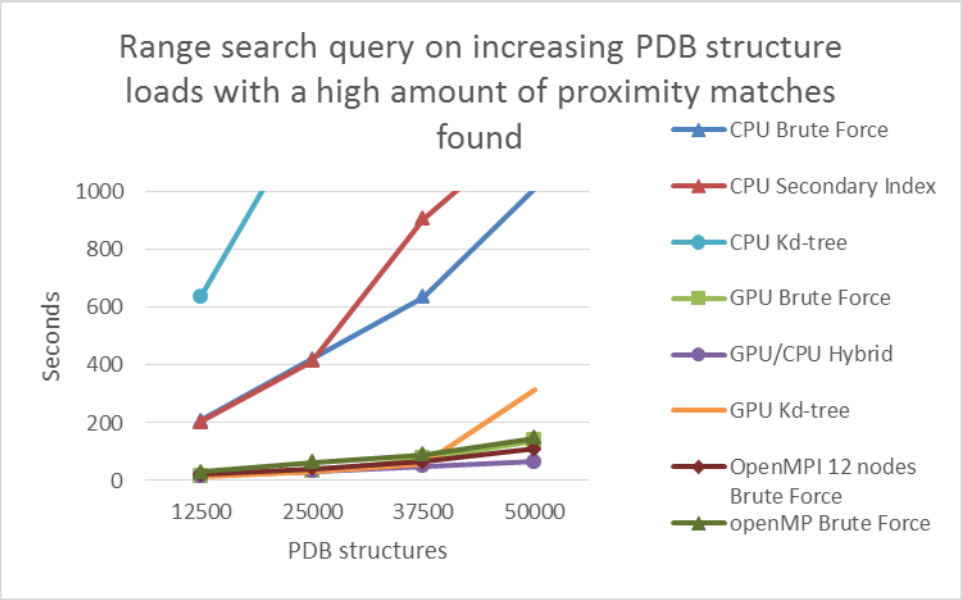


Figure 5.3: Visualisation of Table 5.3's values. In this graph an atom pair which occurs very frequently has been range searched and the CPU approaches have become far slower than the parallelized approaches. At this level of compute intensity OpenMPI run with four cores no longer keeps up with GPU and OpenMP runs – but the 12 core OpenMPI run still keeps up with all parallel approaches except the CPU/GPU hybrid approach.

Discussion of results

The zero result count searches displayed in table 5.1 (executed by searching for proximity matches between two types of atoms which do not exist) revealed the lowest possible execution times that the different parallel algorithms could take to complete their processing loads. When comparing the results of Table 5.1 to Table 5.2 and Table 5.3 however, one sees that the GPU algorithms and openMP hashed index approaches which started out as the slowest approaches actually become the fastest approaches once the most compute heavy proximity match is reached (the 12 process openMPI run does keep up in the compute heavy proximity match but in the previous two experiments the 4 process openMPI run was enough to keep up with the other parallel approaches).

When looking at the medium input intensity run times in Table 5.2 on page 105 we see that all the parallel platforms decrease the run time of the CPU range search algorithms to roughly half of the time the base CPU algorithms take to run – with the OpenMPI approach decreasing the time taken to about a 5th when run with 12 nodes and the kd-tree approaches performing slightly better than their brute force

counterparts.

Finally when comparing the run times of the highest input intensity range searches in Table 5.3 on page 106 the relationships get more interesting. In this high results setting the GPU brute force and OpenMP approaches complete over five times faster than the base CPU algorithms – outdoing the OpenMPI approach which performed better in both the low and medium results settings. Of note is that in this case the GPU kd-tree algorithm becomes notably less efficient than the GPU brute force variants on the largest datasets.

For the GPU implementations in specific, it is worth noting that the difference between the run times of queries with no atom pairs to search through and queries with high quantities of potential atom pairs to search through is very small. This is due to the fact that unlike the CPU algorithms which are able to abort their processes early if it becomes clear there will be no results, the GPU brute force has no escape point until the majority of the work has already been done – while the GPU kd-tree approach has an escape point at set intervals. The benefits of the CPU/GPU hybrid approach are also observable at this point as while this approach still takes longer than the pure CPU approach for the null result queries, the added CPU validation phase greatly decreases the time wasted processing queries on the GPU.

The OpenMP results are a bit more complicated. In the no result queries the OpenMP runs had very high run times even though the OpenMP algorithms implemented the CPU algorithm's escape clauses for aborting queries which could return no results.

5.1.2 Comparing absolute run time of parallel algorithms on set lengths of randomly selected PDB structures

Table 5.4 on page 111 and table 5.5 on page 112 show how long different range search approaches took to range search 5000 files while searching for proximity between low frequency atoms and then high frequency atoms respectively. In these two sets of experiments, all settings were fixed across all runs except for the length of the proteins in the file lists being fed into each algorithm.

This setup was done to reduce the noise caused by processing randomly selected PDB structures (especially as the smaller length PDB structures were much more common) so the ability of different algorithms to process different sized queries could

be viewed more clearly.

Range search completion times with varying average file sizes (low results):					
Atom Pair: C1'-C2'					
Required atom pair proximity: 4 Ångstroms					
Unit of measurement: Seconds (s)					
Number of random files processed: 5000					
Known occurrence frequency of sought atoms: above 4 instances per file *(Low)*					
Size of PDB structures:	1024	2048	4096	8192	16 384
CPU Brute Force	0.12	0.14	0.62	0.64	3.86
CPU Hashed Index	0.08	0.077	0.28	0.42	2.04
CPU Kd-tree	0.27	0.16	0.44	1.43	4.59
GPU Brute Force	2.89	2.62	3.04	3.32	3.89
Hybrid Brute Force	0.54	0.48	0.55	0.70	0.98
GPU Kd-tree	1.48	1.12	1.61	1.92	2.09
OpenMPI 4 processes (Hashed Index)	0.03	0.02	0.05	0.14	0.55
OpenMPI 12 processes (Hashed Index)	0.02	0.02	0.03	0.39	0.32
OpenMPI 4 processes (kd-tree)	0.08	0.04	0.12	0.49	1.41
OpenMP run with 16 threads: Hashed Index	2.89	2.86	3.22	2.94	3.71
OpenMP run with 16 threads: Kd-tree	4.11	4.23	4.22	4.29	4.98

Table 5.4: Processing times of low occurrence atom range searches of protein entries with different average lengths measured in seconds

Range search completion times with varying average file sizes (high results):					
Atom Pair: CB-CG					
Required atom pair proximity: 4 Ångstroms					
Unit of measurement: Seconds (s)					
Number of random files processed: 5000					
Known occurrence frequency of sought atoms: above 650 instances per file *(High)*					
Size of PDB structures:	1024	2048	4096	8192	16 384
CPU Brute Force	2.21	8.26	54.55	127.93	478.92
CPU Hashed Index	2.24	8.35	52.97	125.71	453.49
CPU Kd-tree	3.94	11.62	24.49	61.36	225.63
GPU Brute Force	1.67	2.24	4.49	7.08	10.65
Hybrid Brute Force	1.21	2.11	3.61	7.49	10.52
GPU Kd-tree	4.38	5.86	6.82	7.23	10.41
OpenMPI 4 processes (Hashed Index)	0.40	1.91	6.68	25.60	102.05
OpenMPI 12 processes (Hashed Index)	0.16	0.72	3.37	9.03	35.28
OpenMPI 4 processes (Kd-tree)	1.10	3.17	6.65	17.02	62.01
OpenMP run with 16 threads: Hashed Index	2.99	3.32	4.30	7.65	39.04
OpenMP run with 16 threads: Kd-tree	4.41	4.97	6.06	9.73	23.94

Table 5.5: Processing times of high occurrence atom range searches of proteins entries with different average lengths measured in seconds

Discussion of results

The first notable behaviour in these two experiments is that there are clear areas where the parallel kd-tree approaches perform better than hashed index approaches – which is hidden by noise in the previous random file experiments in Tables 5.1 to 5.3.

If one looks at the GPU algorithms specifically, which had the longest completion times in Table 5.1 and Table 5.5 (in the 1024 structure length range for Table 5.5) and the best performing algorithms in the medium to long length PDB structure

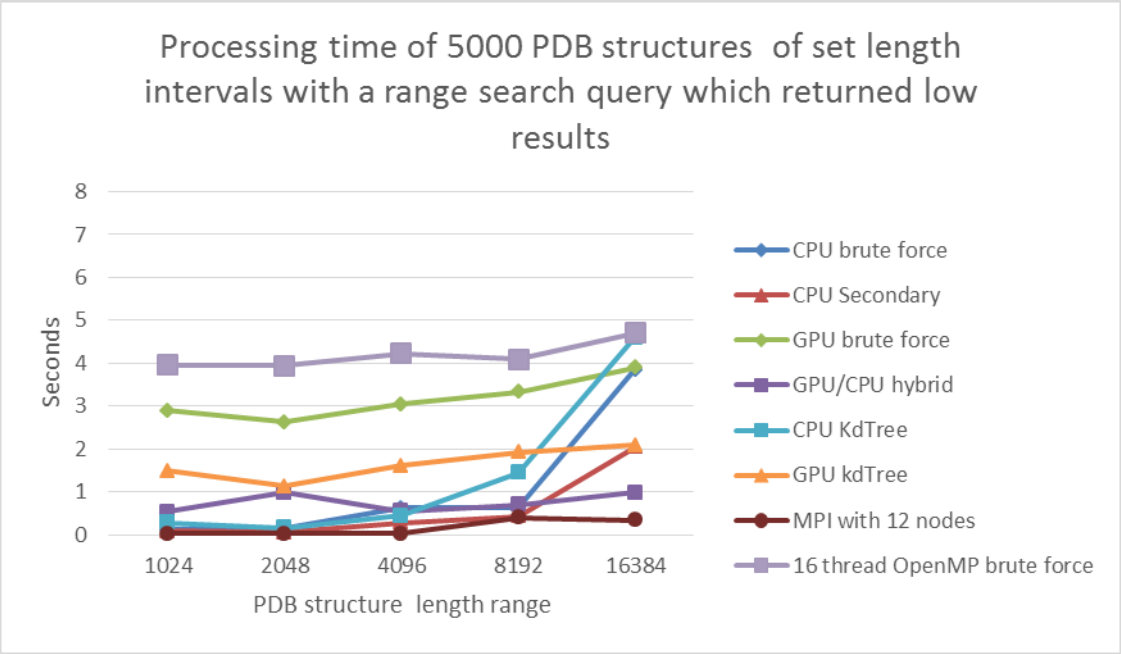


Figure 5.4: In the low result searches of Table 5.4 on page 111, the OpenMPI and GPU/CPU hybrid approaches perform most efficiently time wise while the CPU kd-tree approach and openMP approaches perform terribly

searches in Table 5.5, one sees that the GPU kd-tree algorithm completes faster than the GPU brute force at low sought atom intensity range searches (Table 5.4) but completes slower than the GPU brute force approach in high sought atom intensity range searches for shorter PDB structure chains (although the GPU kd-tree approach does improve in running time compared to the GPU brute force approach as the individual structures increase in length). While the GPU kd-tree does catch up in running speed with the GPU brute force approach for the most compute intensive range searches (the largest files with the high expected sought atom count), the kd-tree algorithms for all the CPU side implementations outperformed their parallel hashed index counterparts at processing the most compute intensive range searches – meaning that the GPU setting seems to get less out of this spatial structure than more traditional sequential architectures do.

A final note from these two tables is how well the openMP and GPU implementations scale with increased processing requirements per individual protein structures. If one looks at Graph 5.5 one can clearly see how these two parallel platforms are hardly affected by compute load increases from PDB structures which keep doubling in size, while the CPU and openMPI run times more than double with the PDB structure size.

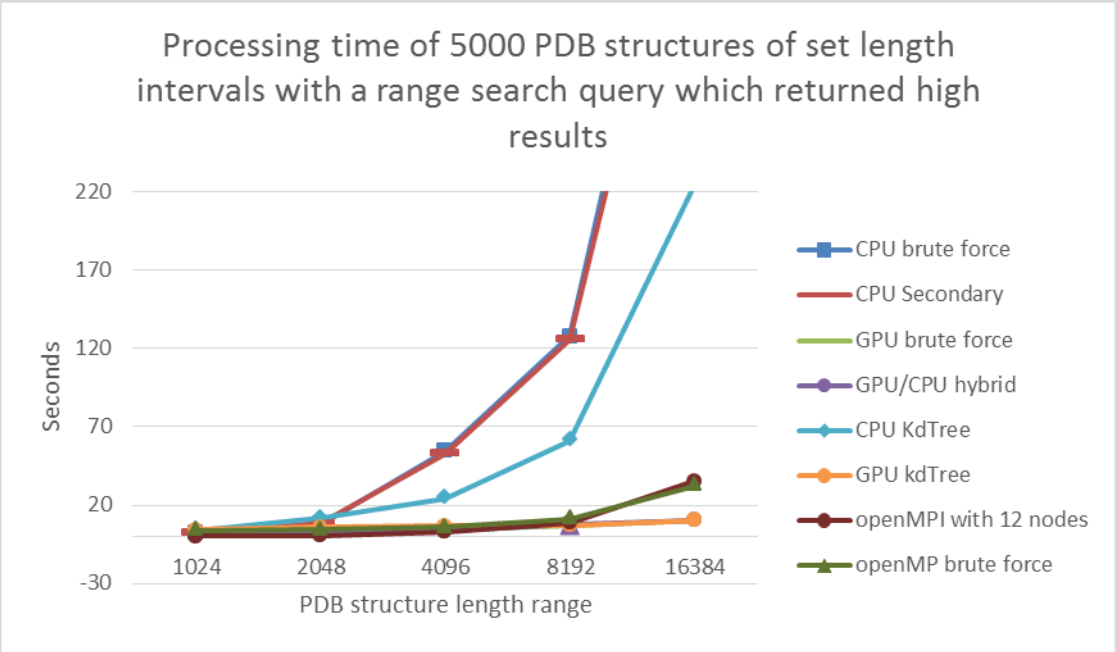


Figure 5.5: In the high results searches of Table 5.5 on page 112, the normal CPU techniques take excessively long to process while the GPU, openMPI and OpenMP techniques perform comparable well – until the largest file sizes are processed, where the OpenMP multi-threading approach and openMPI approach lose some ground to GPU based parallelism.

5.2 Secondary Objectives

The following three subsections record the run times of different approaches to complete the three segments of this research's pre-processing stages: the loading of data from PDB files, the construction of hashed indexes for the algorithmic approaches which required them, and the construction of kd-tree structures for the algorithmic approaches which required them.

5.2.1 PDB structure extraction from PDB files

Table 5.6 notes the amount of time that different PDB data extracting approaches took to load the protein data from secondary storage into RAM. As the libraries which extract this sort of data were written solely for CPU environments, the table below only contains the results of approaches executed in CPU based environments. While attempting to write a custom GPU based PDB data parser was considered, there is no logical way to speed up the process of extracting data from complex file structures of PDB files with a secondary GPU based processor.

The method of extracting protein data from PDB files was with an open source library called ESBTL (Easy Structural Biology Template Library) – a light weight library built to allow the retrieval and manipulation of PDB file information. The normal CPU approach made use of the base ESBTL functionality while the OpenMP approach was simply an adaption of the normal CPU approach where the act of processing incoming files was allocated to be run over 16 threads (which was done by specifying that the loop which made the ESBTL calls would use distinct resources on each call and then instructing openMP to execute up to 16 instances of this loop whenever possible). The OpenMPI approaches simply allocated subsets of the total data load to be processed on separate OpenMPI processes – with each OpenMPI process being assigned a physical core on a single processing node to work on.

PDB Load time comparison:				
Unit of measurement: Seconds (s)				
Number of PDB files:	12 500	25 000	37 500	50 000
CPU implementing ESBTL	1386	2854	7508	8045
OpenMP with 16 threads: implementing ESBTL	912	1793	2640	3624
OpenMPI with 4 processes: implementing ESBTL	371	509	932	1673
OpenMPI with 12 processes: implementing ESBTL	133	270	441	651

Table 5.6: The compute times of different approaches for loading PDB file data to RAM measured in seconds

The results achieved here are straight forward. When comparing the total time required by each algorithm to load the exact same PDB structures, both OpenMP and OpenMPI adaptions reduced the total runtime of the base CPU loading algorithm. In this setting, the reduction in processing time provided by OpenMPI scales linearly with the number of processes allocated to the process.

5.2.2 Comparing hashed index construction across parallel algorithms

Table 5.7 deals with the creation time of non-parallel lookup structures for loaded protein data. The indexing tables referenced in this table were constructed to allow CPU algorithms to obtain lists of any of the atoms inside any of the platform's stored proteins without having to manually search through the entire protein to find them. Constructing these efficient indexing tables were much faster than creating the kd-tree structures which are dealt with in the next subsection.

hashed index construction time:								
Indexes built from loaded PDB structures:								
Time in (s), and speed-up shown								
Number of PDB structures:	12500		25000		37500		50000	
CPU	17	1.0	31		84		105	
OpenMPI with 4 processes	6	2.8	10	3.1	16	5.2	20	5.2
OpenMPI with 12 processes	2	8.5	4	7.7	7	12	10	10.5
OpenMP with 16 threads	3	5.6	8	3.8	6	14	10	10.5

Table 5.7: The compute times for constructing non-spatial lookup structures for stored protein chains measured in seconds. The speed-up is shown with respect to the CPU version

In this set of results OpenMPI provides a linear increase in efficiency equal to the number of processes past the four processes mark but provides below linear speedup at 12 process usage. This is likely because the total processing time is very low so the overhead of running OpenMPI is still visible - especially seeing as the OpenMPI run with 12 processes has almost returned to linear improvement on the batch which takes over 100 seconds for the CPU to run. Unlike its inferior performance in the last set of experiments (loading data) OpenMP now provides a greater improvement to runtime than the 4 processes openMPI run and about equal improvement to the 12 processes OpenMPI run. This reversal is most likely related to the fact that the previous experiment required each algorithm to load data from secondary memory while the construction of these secondary lookup structures already had all the relevant data loaded into RAM – highlighting the advantage of OpenMPI's ability to use not only the processing power of multiple physical compute cores but also other increased capabilities of separate physical cores such as increased parallel

throughput.

5.2.3 Comparing kd-tree construction time across parallel algorithms

This project's kd-tree construction took on two distinct phases:

- Constructing pre-sorted position arrays for each dimension to be cycled through
- Consuming the constructed arrays to construct kd-trees of the represented protein chains

In addition to the various CPU-based parallel approaches, there were multiple ways to blend the use of GPU side processing and CPU side processing.

Kd tree construction times:				
–Mixed file ranges–				
Unit of measurement: Seconds (s)				
Number of PDB structures:	12 500	25 000	37 500	50 000
CPU Pre-process and Construction	803	1634	2924	3771
CPU Pre-process / GPU Construction	36	67	101	166
GPU Pre-process / CPU Construction	1191	2108	3047	4587
GPU Pre-process and Construction	329	658	981	1314
OpenMPI 4 processes: CPU Pre-process and Construction	219	442	702	933
OpenMPI 12 processes: CPU Pre-process and Construction	78	155	232	308
OpenMP run with 16 threads: CPU Pre-process and Construction	38	74	103	149

Table 5.8: The compute times for constructing spatial lookup structures for stored protein chains measured in seconds

The results for the construction of kd-trees was a good example for the potential use of CPU and GPU processing in tandem. While OpenMPI again provided linear speedup based on the number of processing nodes assigned to the task, in this instance both the GPU inclusive approach and the OpenMP approaches provided far greater speedup than the distributed computing approach could provide. While the improvements offered by utilizing OpenMP parallelism are straight forward, it is interesting to note that using exclusively GPU side processing to construct kd-trees was less efficient than using a mixture of CPU pre-processing and GPU based kd-tree construction components.

Another note being that the fairly complex kd-tree construction algorithm described in 4.5.2 was the one which achieved the impressive completion times with the CPU pre-process / GPU construction combination.

5.3 Measuring speed-up

The performance of openMP and openMPI algorithms are tied not only to the operations performed within the algorithms themselves but also the resources and potency of the hardware upon which they are run.

This section evaluates speed-up for OpenMP and OpenMPI, noting that the physical architecture has 12 physical or 24 hyper-threaded cores.

5.3.1 Identifying the optimal OpenMP parallel resourcing

OpenMP allows for users to manually specify that certain segments of code should be run in parallel through multi-threading. When initialising an OpenMP algorithm with parallelized segments of code, the user can choose to specify what the maximum number of concurrent parallel threads should be at any point in time. In the graphs below, the runtime of the hashed index OpenMP range search algorithm and then the kd-tree OpenMP range search algorithm with increasing numbers of threads are shown while processing PDB structures of different length ranges.

OpenMP hashed index range search times:								
5000 PDB structures per batch								
Atom Pair: CB-CG								
Required atom pair proximity: 4 Ångstroms								
Unit of measurement: Seconds (s), and speed-up shown								
Average length of PDB structures:	1024		2048		4096		8192	
CPU version	1.14	1.0	5.38	1.0	18.60	1.0	71.56	1.0
2 threads	13.2	0.08	13.7	0.4	14.8	1.2	18.6	3.8
4 threads	7.1	0.16	7.4	0.7	7.7	2.4	9.9	7.2
8 threads	3.9	0.3	3.9	1.3	4.1	4.5	5.5	13.0
12 threads	3.0	0.7	3.5	1.5	3.5	5.3	4.2	17.0
16 threads	3.4	0.33	3.3	1.6	3.7	5.0	4.5	15.9
20 threads	3.2	0.3	3.2	1.7	3.5	5.3	4.4	16.2

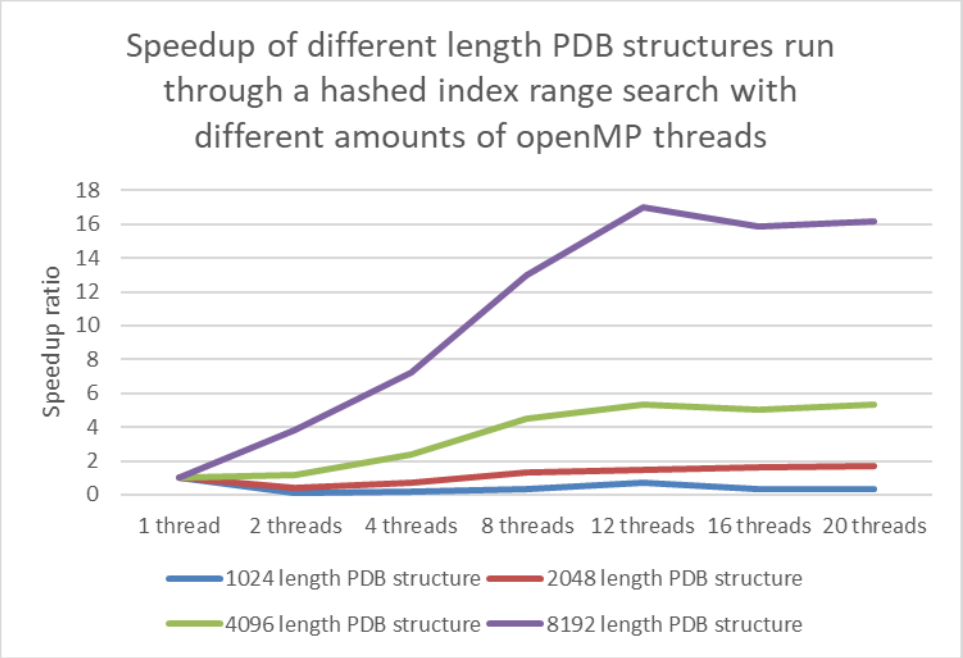
Table 5.9: The compute times of performing OpenMP brute force range searches on 5000 PDB structures measured in seconds with an accompanying speedup ratio showing the difference in run times they had with the base CPU version featuring no threading. As can be seen, high numbers of threads need to be used for the openMP approach to not perform worse then the CPU version on small PDB files while even a few openMP threads resulted in impressive speedup when applied to the longer (and thus more compute intensive) PDB files.

OpenMP kd tree range search times:								
5000 PDB structures per batch								
Atom Pair: CB-CG								
Required atom pair proximity: 4 Ångstroms								
Unit of measurement: Seconds (s), and speed-up shown								
Average length of PDB structures:	1024		2048		4096		8192	
CPU version	3.9	1.0	11.6	1.0	24.5	1.0	61.3	1.0
2 threads	14.3	0.27	15.7	0.7	18.0	1.4	24.2	2.5
4 threads	7.6	0.5	8.5	1.3	9.7	2.5	13.0	4.7
8 threads	4.2	0.9	4.6	2.5	5.3	4.6	7.08	8.6
12 threads	3.9	1.0	3.6	3.2	3.9	6.3	4.9	12.5
16 threads	3.4	1.1	3.8	3.0	3.9	6.3	4.7	13.0
20 threads	3.3	1.2	3.6	3.2	3.7	6.6	5.1	12.0

Table 5.10: The compute times of performing OpenMP kd tree range searches on 5000 PDB structures measured in seconds with an accompanying speedup ratio showing the difference in run times they had with the base CPU version featuring no threading.

As can be seen in both table 5.9 and table 5.10 as well as graph 5.3.1 below, while providing two or four threads has the effect of doubling or quadrupling the speed at

which the PDB structures could be processed, there are clearly diminishing returns per thread added over 8 threads. Also of note is that increasing the thread count when processing larger files (4096 – 8192 lengths) there is a larger speedup then what the extra openMP threads provide in the shorter PDB structures. What was also found was that while using 20+ threads or more could provide slight speedup to OpenMP algorithms, using above 16 threads occasionally caused algorithms to become unstable when processing PDB structures in the 16000 atom length range – and as using 16 threads provided most of the speedup boost anyway, the decision was taken to use 16 thread OpenMP runs for all the other experiments performed in this research.



5.3.2 Identifying the optimal OpenMPI parallel resourcing

OpenMPI allows users to run OpenMPI processes – enclosed environments which can only interact with other OpenMPI processes through message passing. If the processing load is heavy, an individual CPU processing core will usually be completely occupied by running a single openMPI process – although if the core still has a lot of idle time then it may be able to increase efficiency by running two openMPI processes at a time. The usual behaviour for openMPI however is to have distributed computers of multi-core setups hosting an openMPI process per available core.

In this research a host was used which contained 2 CPU chips, each of which had 6

processing cores (for 12 processing cores total) and each processing core was hyper-threaded – which meant that depending on the processing load, the 12 core setup could possibly simulate 24 cores.

In the main experiments, the approach taken in performing the range searches with openMPI was to split the unpackaging of the PDB structures into the memory caches associated with each OpenMPI process and then have each OpenMPI process execute the range searches on the PDB data it was personally storing.

In the graphs below, the runtime of the hashed index OpenMPI algorithm and then the kd-tree OpenMPI algorithm with increasing numbers of nodes utilized are shown while processing protein files of different length ranges.

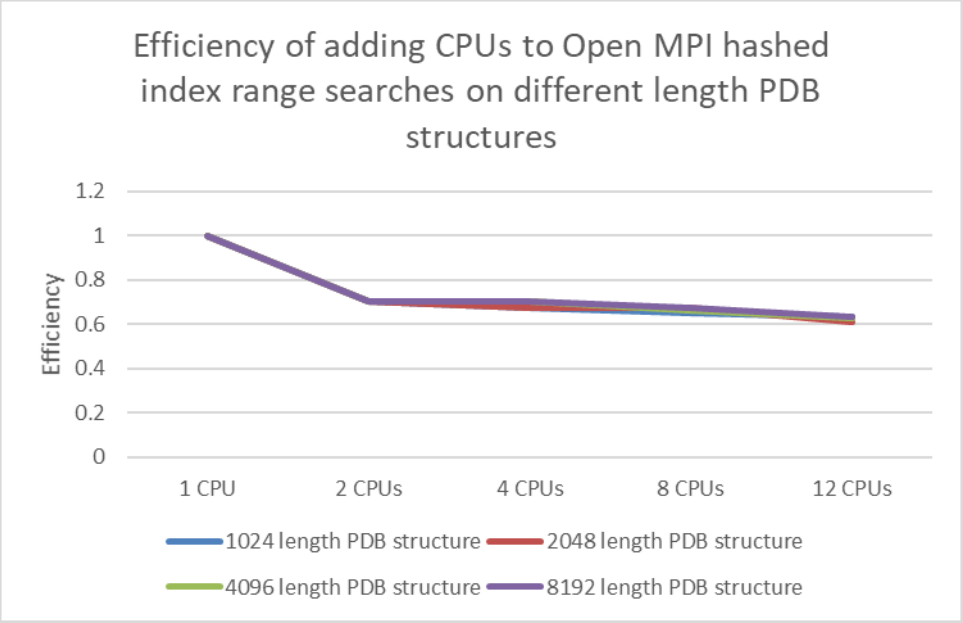
OpenMPI hashed index range search times:								
5000 PDB structures per batch								
Atom Pair: CB-CG								
Required atom pair proximity: 4 Ångstroms								
Unit of measurement: Seconds (s), and speed-up shown								
Average length of PDB structures:	1024		2048		4096		8192	
CPU version	1.14	1.0	5.38	1.0	18.60	1.0	71.56	1.0
2 processes	0.79	1.4	3.88	1.4	13.45	1.4	51.03	1.4
4 processes	0.41	2.7	1.96	2.7	6.75	2.8	25.71	2.8
8 processes	0.22	5.2	1.00	5.4	3.51	5.3	13.26	5.4
12 processes	0.15	7.6	0.73	7.3	2.48	7.5	9.37	7.6

Table 5.11: The compute times of performing OpenMPI brute force range searches on 5000 PDB structure batches measured in seconds with an accompanying speedup ratio showing the difference in run times they had with the base CPU version featuring no threading.

OpenMPI kd tree range search times:								
5000 files per batch								
Atom Pair: CB-CG								
Required atom pair proximity: 4 Ångstroms								
Unit of measurement: Seconds (s), and speed-up shown								
Average length of PDB structures:	1024		2048		4096		8192	
CPU version	3.9	1.0	11.6	1.0	24.5	1.0	61.3	1.0
2 processes	2.12	1.8	6.30	1.8	13.42	1.8	32.94	1.9
4 processes	1.08	3.6	3.13	3.7	6.85	3.6	17.10	3.6
8 processes	0.59	6.6	1.68	6.9	3.54	6.9	8.93	6.9
12 processes	0.41	9.5	1.2	9.6	3.44	7.1	6.19	9.9

Table 5.12: The compute times of performing OpenMP kd tree range searches on 5000 PDB structure batches measured in seconds with an accompanying speedup ratio showing the difference in run times they had with the base CPU version featuring no threading.

Unlike the diminishing returns seen in the OpenMP multi-threading section above, doubling the node count used in the OpenMPI runs provided an almost non-diminishing increase to speedup equal to the number of cores utilized. As there were no downsides to using as many cores as possible, all 12 cores of the Intel Xeon CPU were utilised during experimentation along with example runs of 4 core openMPI runs to show how openMPI performs with less available resources.



5.4 Relevance to prior work

This research report extended previous research at Wits: (1) *PH2* [34], a framework for mining structural properties from the PDB database using Hadoop; and (2) *SpeeDB* resource [59] for querying batches of PDB structures using kd-trees. This research explored alternative approaches to parallelism and showed that although index structures could be beneficial, appropriate parallelism is more important.

On the insights gained from observing how GPU algorithms behaved when compared to other parallel algorithms, a recurring trait of GPU processing was that GPUs offered very significant parallelism but to be competitive, data set sizes had to be relatively large. This means that tasks which are either embarrassingly parallel or both embarrassingly parallel and compute intensive at the same time are good candidates for GPU processing. One consideration with GPUs is that conditionals and loops make it difficult o exploit the parallel architecture if different threads take different control paths through the program. For the brute force implementation this was not an issue, but for the spatial index approaches they required significant into structuring the code to avoid this problem. In summary – and this may be generalisable – although the result kd-tree algorithm was not worse performing than the brute force approach, there was only small performance improvement to use the spatial structure.

As an example, the GPU's ability to do brute force comparisons was powerful enough to compete, and generally outperform, other parallel algorithms which were using specialised lookup structures and utilizing attributes of the spatial data to gain efficiency. With this in mind, GPU brute force algorithms can be crafted and executed in a time efficient manner as a validation tool for more complex algorithmic endeavours, a time efficient way to validate theoretical interactions on provided data samples or a good tool to perform generalised searches on new data sets.

This is not true in all cases. Molecular Dynamics frameworks such as *NAMD* [55] already have extensive parallel (and GPGPU) implementations and use relatively sophisticated data structures. The difference in our case is that the algorithm searches the data structure once and then answers the question. For NAMD the molecular simulation requires significant processing for each "box". The spatial locality information that the structure provides can be used for much more computation in NAMD than in our case and so it easier to keep the GPU computationally active without fetching new data.

The lesson is that the data structure to be used should be determined by the value that the indexing provides. In some cases, a simpler brute-force approach will be as or almost as efficient as a complex data structure.

Chapter 6

Conclusion

This research looked into comparing the performance characteristics of GPUs against openMP and openMPI algorithms at performing on large protein databases.

To cover the objectives listed in chapter 3, this conclusion will be split to cover 3 topics in order of decreasing importance, namely: How GPUs compared against openMP and openMPI at protein range searching; what scenarios GPUs exhibited better performances in when compared to the other parallel algorithms, and how each parallel platform handled the PDB structure pre-processing phases where applicable.

6.1 Evaluating the performance characteristics of GPU algorithms against openMP and openMPI

The first notable result of section 5.1.1 was that while there was a clear improvement in processing time between a normal CPU hashed-index range search and a CPU kd-tree range search (with the CPU kd-tree range search batches generally finishing in half the time of the CPU hashed-index range search batches), the improvement in compute time achieved with the kd-tree approach in the sequential setting was not observed in the GPU and openMP algorithms. The GPU and openMP hashed-index and kd-tree range search batches either completed in very similar amounts of time or the hashed-index range search would finish faster. As this was a trend across all the experiments, the implication was that algorithms which emphasised parallelism provided better performance gains on parallel architectures than algorithms featuring complex search structures which had to be modified to better fit into parallel environments.

The exception to this was seen in the openMPI kd-tree range search batch runtimes as openMPI was able to maintain the improvement which the kd-tree range search had over the hashed-index range search in the sequential CPU runs. This occurred because the openMPI just ran exact duplicates of the CPU kd-tree algorithm upon different processing nodes at the same time while the GPU and openMP kd-tree range searches had to make use of lower level parallelism with the complex kd-tree algorithms – meaning that the kd-tree range search scaled better with a embarrassingly parallel implementation then it did with course-grained or fine-grained parallel implementations.

As a final point while talking about kd-trees, while memory requirements were usually not an issue in these parallel range search algorithms, the additional space required by kd-trees had a notable impact on how the GPU kd-tree range search had to be batched. Memory on the GPU is finite and while there is ample space for normal tasks, there was an intent to batch load PDB data onto the GPU as quickly as possible while using it and due to the size of the kd-trees, using kd-tree algorithms on the GPU tippled the amount of data which needed to be loaded to GPU global memory for each PDB structure processed.

On the topic of how GPU algorithms compared to the other parallel algorithms in absolute run time, what was found was that range searches needed to be sufficiently compute heavy before GPUs would become the fastest to complete the same range search queries – though when those compute thresholds were met (either through very long PDB structures or high result range searches) the GPU brute force algorithms would outperform the other parallel algorithms by a large margin. In the rest of the cases where the GPU brute force algorithms were not the fastest to complete, a 12 process OpenMPI algorithm would usually perform better than the rest.

6.2 Effect of required total compute intensity per PDB structure on parallel algorithms

Section 5.1.2 dealt with the effect on processing time of the different parallel algorithms when the PDB files being processed continued to double in size.

When looking at the compute intensive range search (the one which returned high quantities of results) what was found was that compared to the other parallel algorithms, the GPU implementations completed much slower on short PDB structures (1024 $\rightarrow$ 2048 range), performed similarly to other parallel implementations

on medium length PDB files (4096 range), and completed long PDB structure (8192 $\rightarrow$ 16384) range searches much faster than the other parallel algorithms.

OpenMP exhibited the same sort of behaviour as the GPU algorithms (in that it had worse performance than the OpenMPI algorithms on short PDB structures but better on long PDB structures) but could not keep up with the processing speed of the GPU algorithms in the longest PDB structure lengths looked at (8192 $\rightarrow$ 16384). OpenMPI by comparison completed the fastest on the shorter PDB structures.

6.3 Performance of different parallel algorithms at pre-processing PDB data

As seen in Table 5.6 and Table 5.7, the time taken to load the protein data from secondary memory and to perform complex secondary structure construction (where such was desired) took orders of magnitude more time than the amount of time taken to range search that same data. The time taken to load 50000 files and construct kd-trees for them on a CPU was about 8000 and 4000 seconds respectively, while the worst time for any range search to be performed on those 50000 files was only about 650 seconds max.

Different parallel platforms were able to reduce the total run time of these pre-processing phases in different ways – with OpenMPI having the greatest impact on the loading of protein data from secondary storage, GPU based processing having a very good kd-tree construction time (about 22 times faster than the base CPU implementation) and OpenMP having a less impressive impact on the loading of PDB structures into RAM while matching the best hashed index and kd-tree construction times.

While every parallel platform did seem to have its niche in these phases, the OpenMPI approach showed a remarkable ability to scale its completion time based on the number of OpenMPI processes it could run – giving the impression that if enough cores could be made available to OpenMPI, it would eventually out scale the other approaches at pre-processing without much diminishing returns.

6.4 Future work

There are three general extensions which can be made to the algorithms this research focused on: improving how the GPU based kd-tree algorithms worked internally to provide better run times, implementing these algorithms on quad-trees instead of kd-trees, or improving the complexity of the queries which the implemented algorithms can execute to deal with more complex requests – which would make this research more relevant to extending some of the prior works cited in the background section.

6.4.1 Improving CUDA core usage in the GPU kd-tree range search

Compared to the GPU brute force range search, the GPU kd-tree range search performed either equally well or worse in nearly all the scenarios explored.

One difference between the GPU brute force and GPU kd-tree range search was that the GPU brute force approach was able to search for matches between all instances of the sought molecules in parallel while the kd-tree only searched for matches with a single sought molecule through the kd-tree at a time in a highly parallel manner. The issue this causes was that the brute force approach was able to use as many parallel processors on the GPU as were available instantly while the kd-tree algorithm would start with a single process and build up more parallel usage over time.

As the GPU kd-tree algorithm works with a queue of nodes/molecules it needs to process in parallel, there is no algorithmic reason as to why one could not queue up multiple molecules to search through the kd-tree in parallel in advance (and start range searching for 5-10 molecules instantaneously instead of 1). Making this change would alleviate the issue of low cuda core usage at the start of the algorithm until the GPU kd-tree algorithm had started navigating enough kd-tree branches to utilize more of the GPU's parallel processing potential.

The obstacle which needs to be overcome to do this however is calculating (and allocating) enough memory space on the GPU to store all the queued position/molecule combos which need to be processed on the kd-tree's next cycle of processing - while ensuring there is enough space to store all resulting proximity matches which arise from those parallel kd-tree descents - as getting more results than your partition for causes memory leaks in the GPU environment and in large PDB structures, the proximity match count occasionally substantially spikes.

6.4.2 Using quad-trees instead of kd-trees

A limiting factor of how well GPUs process kd-trees is the speed at which the viable tree descent routes can be picked up to run in parallel. As the kd-tree starts at the root node, you start the algorithm running a single task as opposed to the GPU brute force approach which can immediately start running 512+ tasks from the start. While this slow start is unavoidable if one is running pure tree structure range searches, using a tree which grows in width faster means that the number of new paths which can be picked up during each round of processing increases. In a kd-tree, each node has at most 2 child nodes – which means that during each round of processing you can at most double the amount of parallel tasks for the GPU to run. As the nodes in a quad-tree have up to 4 child nodes each, this means that parallelism can scale up twice as fast as with the kd-tree.

6.4.3 Matching multiple proximities simultaneously for protein orientation

The algorithms in this research currently find proximities between molecules in a single PDB structure. A worthwhile extension would be to investigate the extension of finding proximity matches between molecules in separate PDB structures. Besides being a complex enough query to have use on its own (and a query type which could extend prior work such as the still available *SpeeDB* online resource [59]), the added complexity of this algorithm would provide a more compute heavy setting which should allow for better experiments on how well the GPU's parallel architecture can mitigate the extra processing requirements of more compute intensive algorithms than the singular PDB structures range searched can provide.

Chapter 7

Bibliography

[1] Farid F Abraham. Computational Statistical Mechanics Methodology, Applications and Supercomputing. *Advances in Physics*, 35(1):1–111, 1986.

[2] Mark J Abraham, Teemu Murtola, Roland Schulz, Szilárd Páll, Jeremy C Smith, Berk Hess, and Erik Lindahl. GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX*, 1:19–25, 2015.

[3] Frank H Allen. The Cambridge Structural Database: a Quarter of a Million Crystal Structures and Rising. *Acta Crystallographica Section B: Structural Science*, 58(3):380–388, 2002.

[4] Michael P Allen and Dominic J Tildesley. *Computer Simulation of Liquids*. Oxford university press, 2017.

[5] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, pages 483–485. ACM, 1967.

[6] David Arndt, Jason R Grant, Ana Marcu, Tanvir Sajed, Allison Pon, Yongjie Liang, and David S Wishart. PHASTER: a better, faster version of the PHAST phage search tool. *Nucleic Acids Research*, 44(W1):W16–W21, 2016.

[7] Tania A Baker, James D Watson, Stephen P Bell, A Gann, MA Losick, and R Levine. *Molecular biology of the gene*. Benjamin-Cummings Publishing Company, 2003.

[8] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. *The R*-Tree: an Efficient and Robust Access Method for Points and Rectangles*, volume 19. ACM, 1990.

[9] Jon L Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[10] Bruce M Boghosian. Computational Physics on the Connection Machine: Massive parallelism-a new paradigm. *Computers in Physics*, 4(1):14–33, 1990.

[11] Bernard R Brooks, Robert E Bruccoleri, Barry D Olafson, David J States, S a Swaminathan, and Martin Karplus. CHARMM: A Program for Macromolecular Energy, Minimization, and Dynamics Calculations. *Journal of Computational Chemistry*, 4(2):187–217, 1983.

[12] Ian Buck. The Evolution of GPUs for General Purpose Computing. In *Proceedings of the GPU Technology Conference 2010*, page 11, 2010.

[13] Nama R Budhathoki, Zorica Nedovic-Budic, et al. Reconceptualizing the Role of the User of Spatial Data Infrastructure. *GeoJournal*, 72(3-4):149–160, 2008.

[14] Nichole Cerutti, Barry V Mendelow, Grant B Napier, Maria A Papathanasopoulos, Mark Killick, Makobetsa Khati, Wendy Stevens, and Alexio Capovilla. Stabilization of hiv-1 gp120-cd4 receptor complex through targeted interchain disulfide exchange. *Journal of Biological Chemistry*, pages jbc–M110, 2010.

[15] Mukesh Chourasia, G Madhavi Sastry, and G Narahari Sastry. Aromatic–aromatic interactions database, A2ID: an analysis of aromatic π-networks in proteins. *International Journal of Biological Macromolecules*, 48(4):540–552, 2011.

[16] Rachel Courtland. The Status of Moore's Law: Its Complicated. *IEEE Spectrum*, 28, 2013.

[17] Francis HC Crick. On protein synthesis. In *Symp Soc Exp Biol*, volume 12, page 8, 1958.

[18] Thomas Scott Crow. *Evolution of the Graphical Processing Unit*. PhD thesis, Citeseer, 2004.

[19] Javier De Las Rivas and Celia Fontanillo. Protein–Protein Interactions Essentials: Key Concepts to Building and Analyzing Interactome Networks. *PLoS Comput Biol*, 6:e1000807, 2010.

[20] Sujay B Desai, Surabhi R Madhvapathy, Angada B Sachid, J Pablo Llinas, Qingxiao Wang, G Ho Ahn, Gregory Pitner, Moon J Kim, Jeffrey Bokor, Chenming Hu, et al. MoS2 Transistors with 1-Nanometer Gate Lengths. *Science*, 354(6308):99–102, 2016.

[21] Peter E Dewdney, Peter J Hall, Richard T Schilizzi, and T Joseph LW Lazio. The Square Kilometre Array. *Proceedings of the IEEE*, 97(8):1482–1496, 2009.

[22] Javier Diaz, Camelia Munoz-Caro, and Alfonso Nino. A Survey of Parallel Programming Models and Tools in the Multi and Many-core Era. *IEEE Transactions on Parallel and Distributed systems*, 23(8):1369–1386, 2012.

[23] Peter Eastman and Vijay S Pande. Efficient Nonbonded Interactions for Molecular Dynamics on a Graphics Processing Unit. *Journal of Computational Chemistry*, 31(6):1268–1272, 2010.

[24] Michael Flynn. Flynns Taxonomy. *Encyclopedia of Parallel Computing*, pages 689–697, 2011.

[25] Michael J Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 100(9):948–960, 1972.

[26] Michael P Frank. The Physical Limits of Computing. *Computing in Science & Engineering*, 4(3):16–26, 2002.

[27] Mark S Friedrichs, Peter Eastman, Vishal Vaidyanathan, Mike Houston, Scott Legrand, Adam L Beberg, Daniel L Ensign, Christopher M Bruns, and Vijay S Pande. Accelerating Molecular Dynamic Simulation on Graphics Processing Units. *Journal of Computational Chemistry*, 30(6):864–872, 2009.

[28] Martin Fuechsle, Jill A Miwa, Suddhasatta Mahapatra, Hoon Ryu, Sunhee Lee, Oliver Warschkow, Lloyd CL Hollenberg, Gerhard Klimeck, and Michelle Y Simmons. A Single-Atom Transistor. *Nature Nanotechnology*, 7(4):242–246, 2012.

[29] Adam Godzik, Andrzej Kolinski, and Jeffrey Skolnick. Topology Fingerprint Approach to the Inverse Protein Folding Problem. *Journal of Molecular Biology*, 227(1):227–238, 1992.

[30] William Gropp, Steven Huss-Lederman, and Marc Snir. *MPI: the complete reference. The MPI-2 extensions*, volume 2. Mit Press, 1998.

[31] John L Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[32] Anthony V Guzzo. The Influence of Amino-Acid Sequence on Protein Structure. *Biophysical journal*, 5(6):809–822, 1965.

[33] Jeremy Hall, Stelvia Matos, Stefan Gold, and Liv S Severino. The Paradox of Sustainable Innovation: The Eroomeffect (Moores Law Backwards). *Journal of Cleaner Production*, 172:3487–3497, 2018.

[34] Scott Hazelhurst. Ph2: an hadoop-based framework for mining structural properties from the pdb database. In *Proceedings of the 2010 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, pages 104–112. ACM, 2010.

[35] John L Hennessy and David A Patterson. *Computer Architecture: a Quantitative Approach*. Elsevier, 2011.

[36] Lawrence Hunter. Molecular Biology for Computer Scientists. *Artificial Intelligence and Molecular Biology*, pages 1–46, 1993.

[37] IBM. IBM Research. Square Kilometer Array: Ultimate Big Data Challenge, 2013.

[38] Hemant M Kakde. Range searching using kd tree. *from the citeseerx database on the World Wide Web: http://citeseerx. ist. psu. edu/viewdoc/summary*, 2005.

[39] Laxmikant Kalé, Robert Skeel, Milind Bhandarkar, Robert Brunner, Attila Gursoy, Neal Krawetz, James Phillips, Aritomo Shinozaki, Krishnan Varadarajan, and Klaus Schulten. Namd2: greater scalability for parallel molecular dynamics. *Journal of Computational Physics*, 151(1):283–312, 1999.

[40] Michael Kanellos. Moores Law to roll on for another decade. `https://www.cnet.com/news/moores-law-to-roll-on-for-another-decade/`, 2003. accessed: 2018-09-10.

[41] Bala Krishnamoorthy and Alexander Tropsha. Development of a four-body statistical pseudo-potential to discriminate native from non-native protein conformations. *Bioinformatics*, 19(12):1540–1548, 2003.

[42] Der-Tsai Lee and CK Wong. Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. *Acta Informatica*, 9(1):23–29, 1977.

[43] Victor W Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, et al. Debunking the 100X GPU vs. CPU myth: an Evaluation of Throughput Computing on CPU and GPU. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 451–460. ACM, 2010.

[44] Benjamin G Levine, John E Stone, and Axel Kohlmeyer. Fast analysis of Molecular Dynamics Trajectories with Graphics Processing UnitsRadial Distribution Function Histogramming. *Journal of Computational Physics*, 230(9):3556–3569, 2011.

[45] Sébastien Loriot, Frédéric Cazals, and Julie Bernauer. ESBTL: efficient PDB parser and data structure for the structural and geometric analysis of biological macromolecules. *Bioinformatics*, 26(8):1127–1128, 2010.

[46] David Luebke and Greg Humphreys. How GPUs Work. *Computer*, 40(2), 2007.

[47] Michael Macedonia. The GPU Enters Computing's Mainstream. *Computer*, 36(10):106–108, 2003.

[48] Svetlin A Manavski and Giorgio Valle. CUDA Compatible GPU Cards as Efficient Hardware Accelerators for Smith-Waterman Sequence Alignment. *BMC Bioinformatics*, 9(Suppl 2):S10, 2008.

[49] J Andrew McCammon and Stephen C Harvey. *Dynamics of Proteins and Nucleic Acids*. Cambridge University Press, 1988.

[50] Gordon E MOORE. Progress in Digital Integrated Electronics. *SPIE milestone series*, 178:179–181, 2004.

[51] R Newman and J Tseng. Cloud Computing and the Square Kilometre Array. Memo 134, 2018.

[52] NVIDIA. Tesla-KSeries-Overview-LR. `http://godzilla.kennedykrieger.org/penguin/Tesla-KSeries-Overview-LR.pdf`, 2013. Page: 2.

[53] Technical report OpenMP Architecture Review Board. OpenMP Application Program Interface Version 4.0, 2013.

[54] Xiangda Peng, Yuebin Zhang, Huiying Chu, and Guohui Li. Free Energy Simulations with the AMOEBA Polarizable Force Field and Metadynamics on GPU Platform. *Journal of Computational Chemistry*, 37(6):614–622, 2016.

[55] James C Phillips, Rosemary Braun, Wei Wang, James Gumbart, Emad Tajkhorshid, Elizabeth Villa, Christophe Chipot, Robert D Skeel, Laxmikant Kale, and Klaus Schulten. Scalable Molecular Dynamics with NAMD. *Journal of Computational Chemistry*, 26(16):1781–1802, 2005.

[56] Steve Plimpton. Fast Parallel Algorithms for Short Range Molecular Dynamics. *Journal of Computational Physics*, 117(1):1–19, 1995.

[57] Worldwide protein Data Bank. Protein Database Statistics Deposition Statistics. `https://www.wwpdb.org/stats/deposition`, 1999. accessed: 2018-09-10.

[58] Shweta Purawat, Pek U Ieong, Robert D Malmstrom, Garrett J Chan, Alan K Yeung, Ross C Walker, Ilkay Altintas, and Rommie E Amaro. A Kepler Workflow Tool for Reproducible AMBER GPU Molecular Dynamics. *Biophysical Journal*, 112(12):2469–2474, 2017.

[59] David E Robillard, Phelelani T Mpangase, Scott Hazelhurst, and Frank Dehne. SpeeDB: fast structural protein searches. *Bioinformatics*, 31(18):3027–3034, 2015.

[60] John T Robinson. The K-D-B-Tree:A Search Structure for Large Multidimensional Dynamic Indexes. In *Proceedings of the 1981 ACM International Conference on Management of Data*, pages 10–18. ACM, 1981.

[61] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006.

[62] Manfred J Sippl. Knowledge-based potentials for proteins. *Current Opinion in Structural Biology*, 5(2):229–235, 1995.

[63] John E Stone, David J Hardy, Ivan S Ufimtsev, and Klaus Schulten. GPU-Accelerated Molecular Modeling Coming Of Age. *Journal of Molecular Graphics and Modelling*, 29(2):116–125, 2010.

[64] John E Stone, Antti-Pekka Hynninen, James C Phillips, and Klaus Schulten. Early Experiences Porting the NAMD and VMD Molecular Simulation and Analysis Software to GPU-Accelerated OpenPOWER Platforms. In *International Conference on High Performance Computing*, pages 188–206. Springer, 2016.

[65] Pandurangan Sundaramurthy, Khader Shameer, Raashi Sreenivasan, Sunita Gakkhar, and Ramanathan Sowdhamini. HORI: a web server to compute Higher Order Residue Interactions in protein structures. *BMC Bioinformatics*, 11(1):S24, 2010.

[66] David E Tanner, James C Phillips, and Klaus Schulten. GPU/CPU Algorithm for Generalized Born/Solvent-Accessible Surface Area Implicit Solvent Calculations. *Journal of Chemical Theory and Computation*, 8(7):2521–2530, 2012.

[67] KG Tina, Rana Bhadra, and Narayanaswamy Srinivasan. PIC: Protein Interactions Calculator. *Nucleic Acids Research*, 35(suppl_2):W473–W476, 2007.

[68] CUDA Toolkit. CUDA Toolkit Documentation. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html/simt-architecture` 2013. accessed 20/03/2017.

[69] V Turner, JF Gantz, D Reinsel, and S Minton. The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things. IDC White Paper, April 2014.

[70] Nicholas Wilt. *The Cuda Handbook: A Comprehensive Guide to GPU Programming*. Pearson Education, 2013.

[71] wwPDB. *Protein Data Bank Contents Guide: Atomic Coordinate Entry Format Description - Version 3.30*, volume 3. wwPDB, 2008.

[72] Jinbo Xu, Ming Li, Dongsup Kim, and Ying Xu. Raptor: Optimal Protein Threading By Linear Programming. *Journal of Bioinformatics and Computational Biology*, 1(01):95–117, 2003.

[73] Changsheng Zhang, Chao Lu, Zhifeng Jing, Chuanjie Wu, Jean-Philip Piquemal, Jay W Ponder, and Pengyu Ren. AMOEBA Polarizable Atomic Multipole Force Field for Nucleic Acids. *Journal of Chemical Theory and Computation*, 14(4):2084–2108, 2018.

[74] Jianting Zhang, Simin You, and Le Gruenwald. Large-Scale Spatial Data Processing on GPUs and GPU-Accelerated Clusters. *SIGSPATIAL Special*, 6(3):27–34, 2015.