Master Thesis: Fast Galactic Structure Finding using Graphics Processing Units

Daniel Wood
MSc Computer Science

Supervisor: Dr. Patrick Marais [1]
Co-supervisor: Dr. Andreas Faltenbacher [2]

Computer Science Department
University of Cape Town

May 28, 2014

[1]patrick@cs.uct.ac.za
[2]faltenbacher@gmail.com

Abstract

Cosmological simulations are used by astronomers to investigate large scale structure formation and galaxy evolution. Structure finding, that is, the discovery of gravitationally-bound objects such as dark matter halos, is a crucial step in many such simulations. During recent years, advancing computational capacity has lead to halo-finders needing to manage increasingly larger simulations. As a result, many multi-core solutions have arisen in an attempt to process these simulations more efficiently. However, a many-core approach to the problem using graphics processing units (GPUs) appears largely unexplored. Since these simulations are inherently n-body problems, they contain a high degree of parallelism, which makes them very well suited to a GPU architecture. Therefore, it makes sense to determine the potential for further research in halo-finding algorithms on a GPU.

We present a number of modified algorithms, for accelerating the identification of halos and sub-structures, using entry-level graphics hardware. The algorithms are based on an adaptive hierarchical refinement of the friends-of-friends (FoF) method using six phase-space dimensions: This allows for robust tracking of sub-structures. These methods are highly amenable to parallel implementation and run on GPUs.

We implemented four separate systems; two on GPUs and two on CPUs. The first system for both CPU and GPU was implemented as a proof of concept exercise to familiarise us with the problem: These utilised minimum spanning trees (MSTs) and brute force methods. Our second implementation, for the CPU and GPU, capitalised on knowledge gained from the proof of concept applications, leading us to use kd-trees to efficiently solve the problem. The CPU implementations were intended to serve as benchmarks for our GPU applications.

In order to verify the efficacy of the implemented systems, we applied our halo finders to cosmological simulations of varying size and compared the results obtained to those given by a widely used FoF commercial halo-finder. To conduct a fair comparison, CPU benchmarks were implemented using well-known libraries optimised for these calculations.

The best performing implementation, with minimal optimisation, used kd-trees on the GPU. This achieved a 12x speed-up over our CPU implementation, which used similar methods. The same GPU implementation was compared with a current, widely-used commercial halo finder FoF system, and achieved a 2x speed-up for up to 5 million particles. Results suggest a scalable solution, where speed-up increases with the size of dataset used. We conclude that there is great potential for future research into an optimised kd-tree implementation on graphics hardware for the problem of structure finding in cosmological simulations.

Acknowledgements

Thank you to Dr. Patrick Marais and Dr. Andreas Faltenbacher. Patrick, for his guidance, suggestions and hard work; and Andreas for the continued support and patience while supplementing my limited understanding of the field of Astronomy. Thanks to my parents for their continued support and understanding, as well as to Nina for the comments and suggestions. Finally, thank you to my colleagues for the good times in the lab; keeping company during all-nighters and the technical support.

Contents

List of Figures

Chapter 1

Introduction

This chapter will explore the context and motivation for an investigation into accelerating current halo-finding algorithms using entry-level graphics processing units (GPUs). We describe the problem statement and relate this to the research objectives that we aim to address. Finally, we detail the structure of this thesis.

1.1 Motivation

Structure-finding, in a cosmological context, refers to an algorithmic method of identifying gravitationally-bound objects in the universe, such as dark matter halos. Such dark matter halos generate the gravitational potential wells, within which galaxies and stars are expected to form [18]. In the study of galaxy formation and evolution, the identification of such structures is important. As such, researchers in the field have created a large number of methods and algorithms to accomplish this task.

However, with new technological developments, in particular, the extensive use of high performance computing, this software needs to deal with increasingly larger amounts of data, both accurately and efficiently. Consequently, simulations including post-processing are limited in scale by the computing resources available. Processing such large amounts of data using simple serial code can be cumbersome, and often takes a great deal of time to complete. Researchers have therefore developed a variety of parallel algorithms to deal with this data. Parallel programs are able to analyse multiple particles at once, thus deceasing analysis run-time by a significant degree. A many-core approach to the problem using GPUs, however, seems largely unexplored. Since the majority of halo-finding methods operate on the positional data of many particles, halo-finding can be viewed as an n-body problem. That is, a simulation of a dynamical system of particles, often influenced by physical forces such as gravity [4]. As such, there is considerable potential for accelerating the identification of dark matter halos by leveraging the massively parallel computational power of modern commodity graphics hardware.

Modern GPUs have shown tremendous potential in accelerating algorithms amenable to parallel implementation. In addition, graphics cards are becoming increasingly accessible to programmers with regards to both cost and programmability. As such, operating on the assumption that the majority of potential users of these systems have access to entry-level graphics cards, the development of a halo-finding code that makes use of commodity graphics hardware could present a more efficient alternative to current single- and multi-threaded solutions.

1.2 Problem statement

New technological developments in telescopes and other sensory equipment are constantly enabling the generation of ever larger data sets. That is, larger numbers of n-body particles per simulation. Dealing with this volume of data efficiently, with specific regards to halo-finding, is becoming an increasingly difficult task. Astronomers without access to large amounts of computational processing power are limited to smaller simulations that do not require such immense computational power for post-processing.

We propose an investigation into the possibility of leveraging the massively parallel processing capabilities of entry-level graphics hardware to efficiently process dark matter halo-finding in large N-body simulations. This will be explored as an alternative to current single- and multi-threaded solutions.

1.3 Research goals

Keeping the aforementioned problem statement in mind, we aim to address the following research objectives:

- Investigate current methods of halo-finding and determine an algorithm favourable to both CPU and GPU implementations.
- Conduct a proof-of-concept exercise by developing halo-finding applications for both the CPU and the GPU using commodity graphics hardware. These applications will foster familiarity with the algorithm chosen, following the halo-finding method investigation, as well as other nuances inherent in the implementation of the algorithm and the problem as a whole.
- Develop competitive halo-finding applications for both the CPU and GPU using commodity graphics hardware. These applications will leverage lessons learned during the proof-of-concept implementations and investigate the feasibility of potential future work in developing a commercial GPU halo-finder.
- Conduct comparative tests on all applications that were implemented. These tests will compare the GPU and CPU running-times for the developed applications, as well as a current commercial halo-finder. In the event of interesting unforeseen results, additional tests will expand on these appropriately.

1.4 Thesis structure

This thesis consists of 6 chapters as follows:

1 This chapter introduces the context and motivation for this research. The chapter is divided into 4 parts; motivation, problem statement, research goals and the structure of this thesis.

2 The second chapter introduces relevant concepts in Astronomy and GPU computing, and serves as a background literature review for this research.

3 Following this, an overview of all algorithms used and the design of all applications developed during this research will be explored. The chapter is broken down into multiple sections, each describing a separate algorithm or implementation.

4 This chapter discusses the implementation of the applications developed for this research. Specifically, the chapter describes technical details, as well as data-structures and libraries used during development.

5 The results of our application testing are presented in this chapter, as well as a brief discussion of the data used and other testing details.

6 Finally, this chapter reviews and summarises the work done in this research. The chapter also explores possible improvements on the applications implemented, and its future research potential.

Chapter 2

Background

This chapter introduces relevant background and foundational concepts in Astronomy and graphics processing unit (GPU) computing, paying particular attention to halo-finding. Large scale structures are discussed and a number of these are described in detail. Halo-finding and its importance is explored, and a comparison of current halo-finders, based on appropriate criteria, is presented, clearly highlighting the importance of this research. GPUs and their relevance to this research, and to Astronomy in particular are explained. This introduces a discussion of the programming model used to treat GPUs as generally programmable devices. Finally, the chapter is summarized, with key points and important information highlighted.

2.1 Astronomy

We describe astronomy concepts relevant to this research. In particular, we discuss large scale structures and halos, and present a brief survey into current methods and existing software used to identify these.

2.1.1 Large scale structures

The term *large scale structure* is used to describe any one of a large variety of aggregate structures in the universe. Examples include *galaxies* and *galaxy clusters*, *sub-structures* within a galaxy and *super-clusters*. A galaxy is a massive, gravitationally bound system

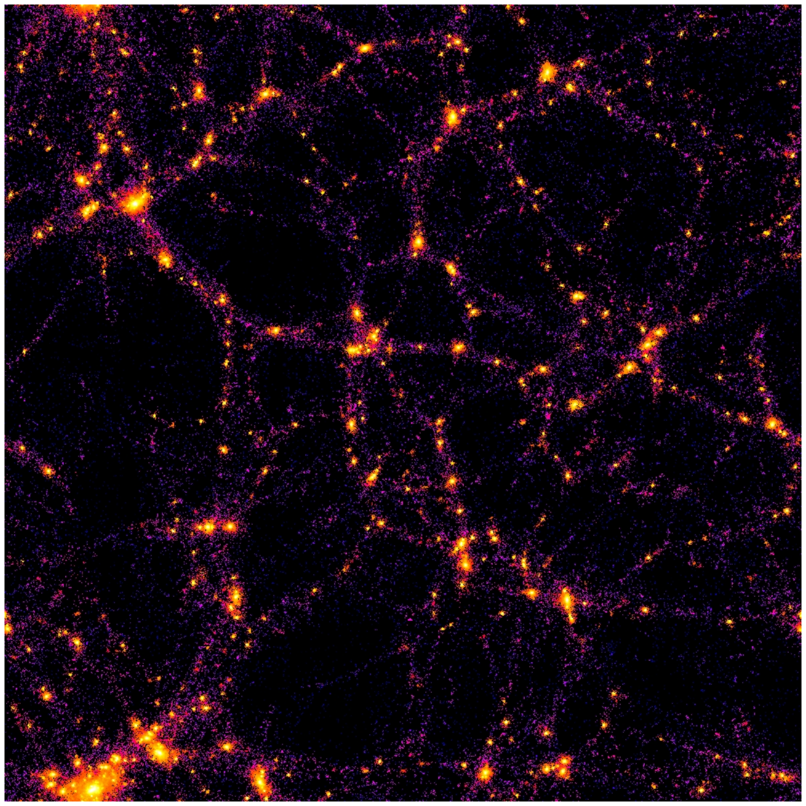

Figure 2.1: *The distribution of dark matter in the universe. Large groups of dark matter create gravity wells within which galaxies are known to form. Discovering dark matter structures is closely related to investigating galaxy formation and evolution [7].*

which consists of a number of elements. These elements include *stars* and *stellar remnants*, as well as an *interstellar* medium of gas and dust, and dark matter [36]. Stellar remnants generally refer to compact stars such as neutron stars and white dwarfs, or a star that is nearing the end of its stellar evolution life-cycle. The interstellar medium of a galaxy is the matter that exists in the vast space between star systems [14]: Such matter can include gas, dust and cosmic rays [36]. *Dark matter* refers to transparent material that is postulated to exist in space and could take several forms including weakly interacting particles, or high energy randomly moving particles. It is hypothesized to account for a large amount of the total mass in the universe. Dark matter cannot be observed directly via telescope or other means, nor does it emit or absorb light or any other electro-magnetic radiation at a significant level. Instead, its existence is inferred from its gravitational effects on visible matter alone [40].

Galaxy clusters, or *galaxy groups*, are largest known gravitationally bound objects and they can generally contain anything from ten to thousands of galaxies [42]. These clusters are often associated with larger, non-gravitationally bound groups known as *super-clusters*. Super-clusters are large groups of smaller galaxy clusters and are among the largest known structures of the universe.

In recent decades, astronomers have discovered that galaxy formation is closely related to the structure of dark matter near the galaxy, placing great importance on discovering the distribution of dark matter in the universe, depicted in Figure 2.1. Due to this, much effort has been invested the development of computer simulations with the purpose of investigating the evolution of *dark matter halos*.

2.1.2 Halos and halo-finding

A *dark matter halo* is an almost spherical component of a galaxy, which is thought to extend far beyond the main visible element. It is a hypothetical component of a galaxy that envelops the galactic disk. That is, it envelops the stellar and gaseous component of a galaxy, as shown in Figure 2.2. The halo also extends well beyond the edge of the visible galaxy. It is hypothesized to account for the vast majority of mass within a galaxy, and is also the main structural component. That is, the mass of this component is great enough that it is primarily responsible for holding the galaxy together via gravity [17].

Since a dark matter halo is a fundamental component of a galaxy, it is important that a means be developed for its detection. *Dark matter halo-finding*, which identifies the dark matter halos, is a key step in extracting useful information from cosmological simulations.

Current cosmological n-body simulations comprise of millions of dark matter particles within which halos, or gravitationally bound structures, need to be identified. An n-body simulation is a simulation of a dynamical system of particles, often influenced by physical forces such as gravity [4]. The fact that these are n-body problems is important, as it implies a large degree of parallelism. Processing such massive amounts of data using serial codes, while more simple, often takes a great deal of time to complete. Researchers have therefore developed a variety of parallel algorithms to deal with this data. Parallel programs are able to analyse multiple particles at once, thus deceasing simulation run-time by a significant degree. Several application programming interfaces (APIs) and technologies have been employed to provide this effect, including *Open MultiProcessing* (OpenMP), *Message Passing Interface* (MPI) and CUDA, a parallel computing platform implemented by graphics processing units (GPUs)

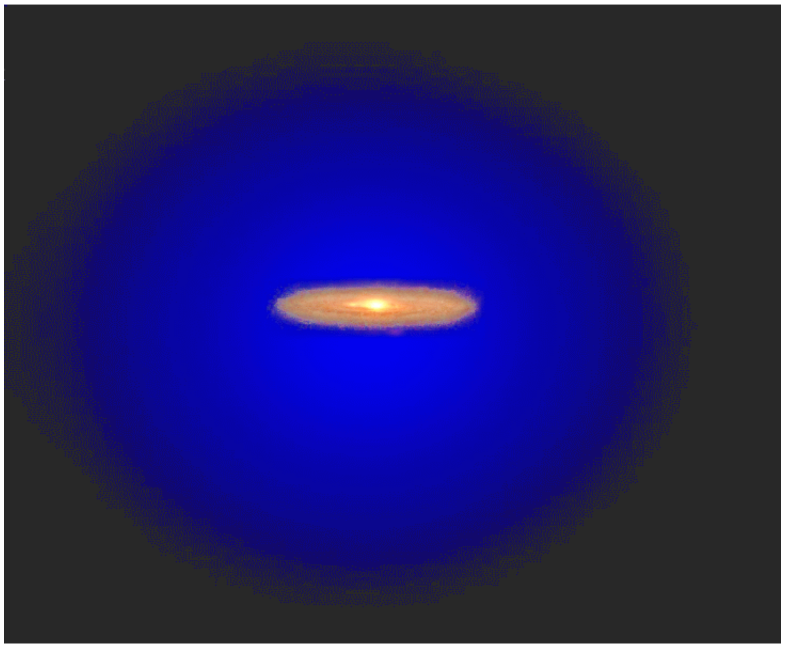

Figure 2.2: *An artist's impression of a dark matter halo surrounding a disk galaxy. The halo surrounds the stellar and gaseous components of the galaxy and extends far beyond the main visible element [22]*

[29].

OpenMP is an API that supports shared memory multiprocessing for parallel programming implementations using multiple central processing unit (CPU) cores [9]. MPI is a standardized message passing interface designed for use with distributed computers [35]. GPUs, however, are fundamentally different in that they do not take advantage of multiple CPU cores, or distributed computing. Instead, they rely on a completely different architecture to accomplish their tasks. Halo-finding implementations on GPUs form the basis of this research, and will be discussed in more detail in section 2.2.

2.1.3 Halo-finder code and method review

There are currently many halo-finding schemes in use, each with their own particular advantages and disadvantages. Here we attempt to compare and discuss current schemes according to objective criteria, and thus demonstrate the deficiencies of existing methods as a motivation for our research. A detailed halo comparison study was presented by A. Knebe et al [21]. The study concludes that the agreement between different halo-finding codes, with regards to halo accuracy, is quite remarkable and reassuring, even though they are based on different techniques. This suggests that the disparity between halo-finding techniques is not extremely important with regards to accuracy, but rather with respect to capability and speed. This highlights the importance of our research. The implementations compared can be found in Table 2.1.

Name	**Processing**	**Dimensionality**
SUBFIND	Sequential	3D
ASOHF	Sequential	3D
BDM	Sequential	3D
SKID	Sequential	3D
AHF	Parallel (MPI)	3D
pSO	Parallel (MPI)	3D
VOBOZ	Parallel (OpenMP)	3D
ORIGAMI	Sequential	3D
LANL	Sequential	3D
FOF	Parallel (MPI)	3D
pFOF	Parallel (MPI)	3D
Ntropy-fofsv	Parallel (MPI)	3D
HAF	Parallel (OpenMP/MPI)	6D
6DFOF	Sequential	6D
Rockstar	Parallel (OpenMP/MPI)	6D

Table 2.1: *A list of the codes compared by A. Knebe et al [21] and sorted by dimensionality. The left most column describes the name of the halo-finder. The middle column describes the type of processing used, and where appropriate, the API used to implement it. Finally, the last column describes whether the code uses only the positions of the particles, or both the positions and the velocities (phase-space).*

These halo-finders employ a wide range of techniques including *friends-of-friends* (FoF), *spherical over-density* (SO) and *phase-space* based algorithms. Phase-space, in the context of this research, refers to the use of both position and velocity coordinates to describe simulation particles. This is considered a six-dimensional problem, as the position and velocity vectors consist of three scalars each.

The FoF method is perhaps the simplest and easiest to implement. Essentially, particles in the simulation are linked together if the distance between them is less than a certain threshold. This threshold is called a *linking length.* After analysing every particle in the simulation, and linking them where appropriate, one is left with groups of linked particles called FoF groups, or FoF halos. It is important to note that FoF groups cannot intersect. That is, a particle can be assigned to only one FoF group for a single linking length. Additionally, sub-structures discovered will always lie completely within a host FoF halo, since they are defined by a smaller linking length. A visual representation can be seen in Figure 2.3 [27].

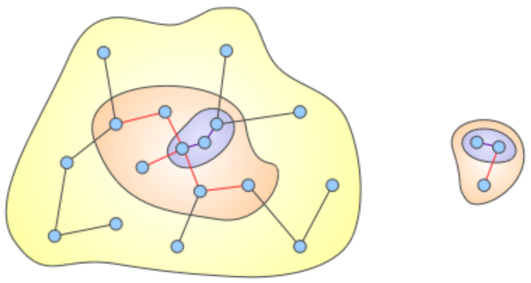

Figure 2.3: *The friends-of-friends method showing two distinct halos. The particles bound by the yellow shaded region indicate gravitationally bound particles [38].*

One drawback of the FoF method is the formation of FoF bridges. This refers to the linking of two different halos, and thus, the formation of a larger, incorrect halo, as seen in Figure 2.4. This occurs when the most distant boundary particles of two different halos are within the linking length threshold distance of each other, and are thus connected forming one halo. Although technically correct according to the algorithm, the newly connected halos are not necessarily one and the same. In addition, if the linking length is set to be too large, sub-structures in the large halos are not detected. On the other hand, if the linking length is too small, sub-structures are detected, but information is lost on a larger scale. These deficiencies can be eliminated via the use of the *hierarchical friends-of-friends* method. This method involves simply increasing and decreasing the linking length by small increments [20].

The SO method is based on the discovery of spherical regions in a simulation which have a certain mean density. This is accomplished by first calculating a local density for each particle by finding the distance to its n^{th} nearest neighbour. Once particles have been sorted according to this, the highest density particle is taken as a candidate centre for the first sphere. A sphere is then grown around this particle, encapsulating all surrounding particles, until the mean density falls below a certain value. The centre of mass in this sphere is nominated as a new centre, and the process is repeated. This process continues until the distance between the new and old sphere centres are below a certain threshold. All particles in this sphere are considered to belong to the same halo, and removed from the sorted list of particles. The next highest density particle on the list is nominated as the centre for a

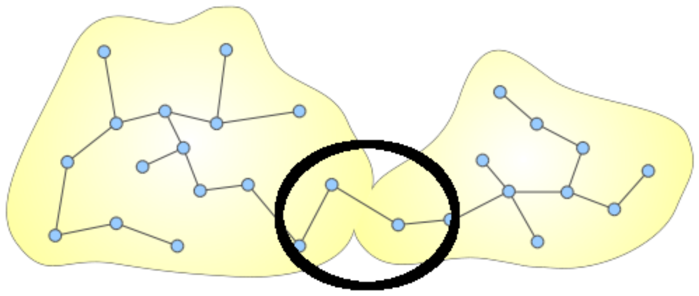

Figure 2.4: *The friends-of-friends method: The formation of a FoF bridge. Two separate halos are incorrectly linked together when the most distant boundary particles of two different halos are within the linking length threshold distance of each other [38].*

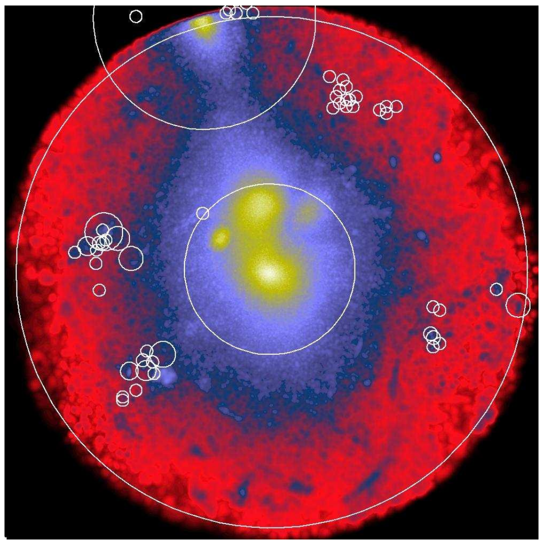

Figure 2.5: *The spherical over-density method: Sub-halo spheres are denoted as white circles [33].*

new sphere and so on. Finally, once all spheres have been found, the spheres that overlap with a larger sphere are merged with the larger sphere [23]. The results of this algorithm are illustrated in Figure 2.5.

Phase-space halo-finders are currently the most complex algorithms, making use of both particle positions and particle velocities. Typically, a phase-space halo-finder simply extends a current known method to six dimensions, either by calculating a phase-space density (in the case of SO), or by using a phase-space linking length (in the case of FoF) [6]. The advantage of using six dimensions instead of three is that halo-finders are able to more accurately find halo centres, even if the centre overlaps with another distinct object.

Run times of each of the above algorithms are relatively similar, but code that takes advantage of parallel processing using MPI and OpenMP have a drastically decreased runtime. There is, however, an absence of GPU implementations for these algorithms. Given the increasing accessibility of such hardware with regards to both cost and usability, this is worth further investigation. Furthermore, as GPUs are very amenable to n-body simulations by design (see section 2.2), it seems prudent to take advantage of the n-body nature of dark matter halo-finding simulations. The FoF approach exhibits a high degree of data-level parallelism, is simple to implement, and is the most common method of halo finding. As such, this method will be used to investigate the appropriateness of future research in the development of halo-finding implementations on GPUs.

2.2 Graphics Processing Units

Graphics processing units (GPUs) have become increasingly commonplace for many kinds of simulations in a wide range of scientific fields. This is largely because they are relatively inexpensive when compared to the super-computers generally used to run such simulations. This section contrasts the design and architecture of GPUs and CPUs, and discusses their importance in a commodity super-computing context. The CUDA (Compute Unified Device Architecture) programming model is also explained, along with the structure and execution method of a typical CUDA program. CUDA kernels and memory types are reviewed, including their inherent advantages and disadvantages, and common CUDA application optimisations are described.

2.2.1 General processing using GPUs

Since the early 2000s, the semiconductor industry has focused on two main avenues for microprocessor design [19]. The *multicore* processor path seeks to increase the number of cores inside a CPU, with, on average, the number of cores present on a CPU doubling with each successive semiconductor processor generation. The *many-core* processor path, however, centres more on the execution throughput of parallel programs. This many-core approach has the ability to turn the computational power of a modern graphics card into general-purpose computing power that can be harnessed by programmers. GPUs are an example of many-core technology, boasting a large numbers of cores, each of which is a heavily

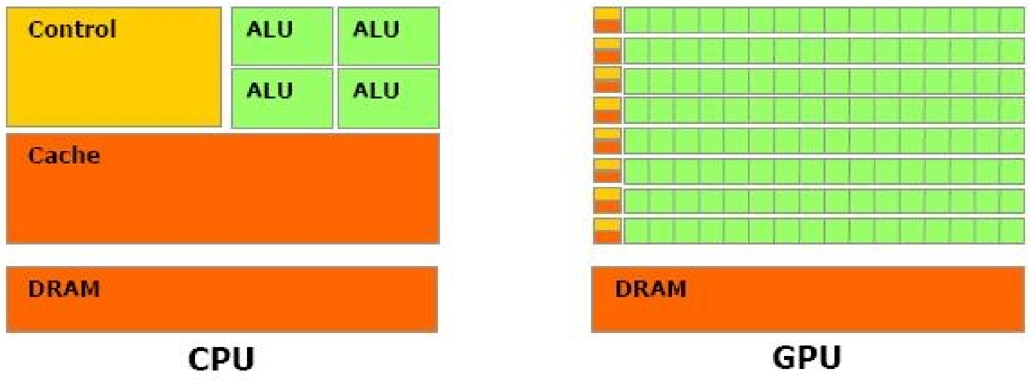

Figure 2.6: *The design architecture differences between a CPU and GPU. GPUs contain many arithmetic logic units (ALUs, shaded green), and are able to process information in parallel [19].*

multi-threaded, in-order, single-instruction processor that shares its control and instruction cache with a number of other cores [19]. Their performance in floating-point arithmetic has been uncontested since 2003. Furthermore, while CPU performance increase has slowed, the performance of GPUs has continued to improve [19]. The reason for this is the vast difference in the fundamental design philosophies between GPUs and CPUs, as illustrated Figure 2.6.

CPU architecture is predominantly designed and optimized for sequential instruction execution. Sophisticated control logic and branch prediction allows instructions from a single thread to execute in parallel while maintaining the appearance of sequential execution [19]. Large cache memories are also used to reduce data and instruction access latencies. In contrast, the architecture of GPUs are motivated by video games, which require the calculation of hundreds of millions of floating-point operations per frame for graphics-intensive games. In order to satisfy these demands, GPUs are optimized for the execution of thousands of concurrent threads. Thus, the hardware takes advantage of a large number of execution threads which find work to do while waiting for long-latency memory accesses. This minimizes the control logic required [19]. In essence, GPUs are designed as numeric computing engines, performing as many floating-point calculations per second as possible. However, focusing design towards such a specific goal typically causes disadvantages in other areas. Thus, although GPUs perform well in many areas, some tasks are better suited to CPUs. Most successful applications use both CPUs and GPUs to accomplish acceleration, executing sections of the code that are not amenable to parallisation on a CPU, and numerically intensive parallelisable parts on the GPU.

General purpose programming for GPUs is used for many parallel tasks, specifically tasks that are suited to the *single instruction, multiple data* (SIMD) class of parallel computers. That is, algorithms with multiple processing elements that perform the same operation on multiple data points simultaneously. This allows for high-throughput computations that exhibit data-parallelism, thus exploiting the large number of cores available. These tasks include ray tracing, computational fluid dynamics and many types of modeling, such as dark matter halo-finding and other astrophysical simulations. In order to allow developers access to the instruction set and memory of the parallel computational elements in GPUs, Nvidia Corporation developed CUDA, a parallel computing architecture for use with their GPUs [29].

2.2.2 The CUDA programming model

CUDA allows developers to treat the GPU as a generally programmable device, rather than a fixed rendering pipeline. Essentially, CUDA requires a program to transfer data from the host (CPU) to the GPU, execute a set of instructions on the data, and copy the result back to the host. Phases of the program that exhibit little or no parallelism are implemented in host code and phases that exhibit a large amount of parallelism are implemented in the device code via a GPU *kernel*. A kernel, in the context of GPU programming, is essentially a function containing code that is to be executed on the GPU. When a CUDA program is run, the execution starts on the host. Typically, memory on the GPU will be allocated, and the specific data to be processed will be copied over. In order to process this data, a kernel is invoked and launched. The execution is then controlled by the device, where a number of parallel threads are generated to take advantage of data parallelism. The collection of threads used to run the kernel is called a *grid*. When all threads of a kernel complete their execution, the corresponding grid terminates, and the execution continues on the host until another kernel is invoked [19].

It is important to note that, because of the SIMD style of programming, all threads execute the same kernel function on different data. This is accomplished by assigning unique coordinates to each thread in order to distinguish them from each other and to identify the appropriate data to process. The grid of threads generated to run the kernel is sub-divided into *blocks*. Each block uses two-dimensional references, uniquely labeling each thread with a block index and a thread index. This thread hierarchy is illustrated in Figure 2.7.

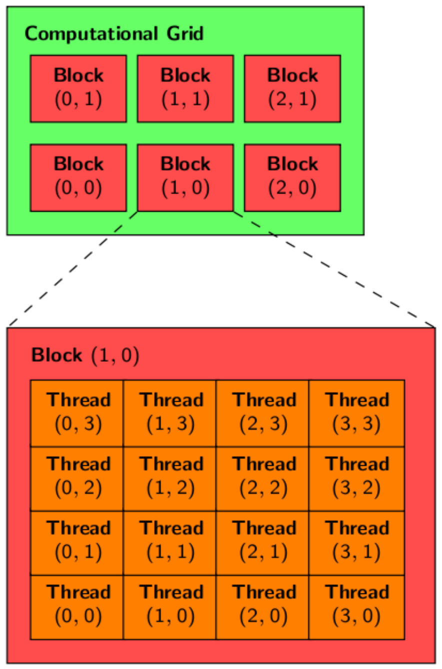

Figure 2.7: *The organisation of CUDA threads. When a kernel is invoked, a grid of threads is generated. A grid is made up of many blocks, each of which contains a number of threads. Blocks are passed to different multi-processors on the GPU [31].*

When a grid of threads is generated for a kernel, blocks are passed to *streaming multi-processors* (SM), each of which execute in parallel with each other. SMs are the parts of the GPU that run the kernel. Every SM contains a number of CUDA cores, each of which executes a sequential thread in parallel. Once a block of threads is assigned to a SM, it is further divided into 32 thread units called warps. The concept of warps is not part of the CUDA specification, but knowledge of them can be beneficial when optimising the performance of a CUDA application. When an instruction executed by threads in a warp is delayed while waiting for the result of long latency operation, the warp is not executed. Instead, another warp in the same block that is not waiting on an operation is executed. This mechanism of allowing another warp to work when one is blocked effectively lowers the latency of expensive operations and is referred to as *latency hiding*. This can greatly increase the efficiency of the application [19].

2.2.3 CUDA memory

In addition to a programming model, CUDA also provides access to a number of different memory types, each tailored for specific tasks. The different memory types offered are described below, as well as brief descriptions of their uses and advantages.

Global memory

Global memory, residing in device memory (DRAM), is the most commonly used memory as it has the greatest capacity, being equal to the amount of DRAM provided by the graphics card model. Although large, it has a low memory access efficiency. That is, it offers high bandwidth, but suffers from very high latencies. This can be alleviated somewhat, however, through the *coalescing* of memory transactions. That is, memory transactions in which consecutive threads access consecutive memory addresses [43]. This is explained further in section 2.2.4. Global memory is most often used to store the data to be worked with on the device. From here, data may be assigned to other memory types as needed.

Registers

In CUDA compute capability 2.x, every SM on the GPU houses 32768 32-bit registers, which are allocated to threads when the kernel is launched [43]. They can contain integer as well as floating-point data and are limited in scope to their assigned thread. They are the largest, fastest memory on the SM. Note this is not comparable with memory types such as global memory, as global memory does not reside on the SM. Registers are generally used to hold frequently accessed variables that are private to each thread.

Local memory

Local memory is normally used when an SM runs out of resources, causing registers to *spill* data. Since it is part of the same device memory as global memory, it exhibits similar characteristics, such as high latency. However, the local memory addressing ensures that memory transactions are automatically *coalesced* [19]. This is explained further in section 2.2.4.

Constant memory

Constant memory resides in DRAM, and together with global memory, can be written to and read by host code by calling API functions. Constant memory allows read only access by the device, but provides low-latency, high-bandwidth access when all threads simultaneously access the same memory location [19]. It is generally quite small, for example, on the Nvidia GTX 260 TI graphics card, the hardware used in this research, it totals 65KB.

Shared memory

Shared memory resides on the SM and is shared by all threads in a block. Shared memory is generally quite small, averaging only 49KB on most devices. Its advantage, however, is its very low latency (about 100x faster than global memory). It is generally used when the same data must be accessed multiple times, and can, in specific situations, vastly increase the performance of a program [43].

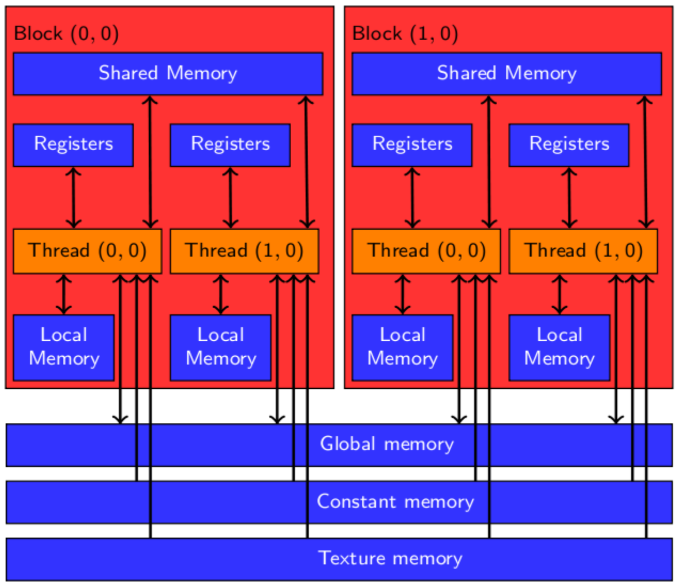

Figure 2.8: *A diagram depicting the many memory types available for use, as well as their access scope. Global and constant memory are accessible by any thread, as well as the CPU. Local memory and registers are thread specific and accessible only from the GPU, and shared memory can be accessed by all threads in a single block [31].*

It should be noted that global, local and constant memory are all part of the same local video memory of the graphics card, and differ only in access models and caching algorithms.

Selecting the memory appropriate to the situation can greatly influence the performance of any GPU implementation. A visual representation of CUDA memories can be seen in Figure 2.8, and their scope, lifetime and speeds are described in Table 2.2.

Memory type	Scope	Lifetime	Latency (cycles)
Register	Thread	Kernel	0
Shared	Block	Kernel	0
Texture	Grid	Program	High cost on first access, then 0
Global	Grid	Program	Greater than 100
Constant	Grid	Program	Greater than 100

Table 2.2: *CUDA memory descriptions with regards to scope, lifetime and latency. With regards to latency, lower is better. Registers and shared memory are stored on chip and so provide very fast access with no latency. Global and constant memories are stored in the device DRAM and are relatively expensive to access [19].*

2.2.4 CUDA optimisations

In general, the first implementation of a CUDA program can provide good speed-ups over CPU versions. However, there are a number of ways in which we can exploit the GPU hardware to fully leverage the parallelism of the problem and thus gain additional speed-ups.

Instruction-level parallelism

Instruction-level parallelism is a measure of how many operations in a program can be computed simultaneously. In the context of SIMD architectures such as GPUs, this refers to how many operations in the kernel can be performed simultaneously. Consider the following example:

$$\begin{aligned} C &= A + B \\ E &= C + D \\ F &= A + D \end{aligned} \tag{2.1}$$

In the equations above, the processing of one equation cannot begin before the equation before it has started, as $C+D$ must wait until C is computed, and so on. However, swapping the last two equations around, $A+D$ can be computed at the same time as $A+B$. This reduces the number of clock cycles required to execute the code [24].

Data-level parallelism

Data-level parallelism, in a multi-processor system, such as GPUs, refers to executing a single set of instructions on each processor on multiple data elements in parallel. There are many methods to accomplishing this, but determining which method that will be most efficient requires creativity and understanding the specific application at hand [32]. For example, computing the gravitational effect on a single body in a simulation can be parallelised at data-level by assigning a single body to each thread, and calculating the accumulated force exerted on that body by every other body in the system [30]. A more detailed description of this is discussed in chapter 3.

Memory coalescence

As detailed in sub-section 2.2.3, memory coalescing refers to memory transactions in which consecutive threads access consecutive memory addresses. Since all threads in a warp execute the same instruction, when a load or store instruction is called, the hardware detects whether the threads access consecutive memory locations. If they do, these memory accesses are coalesced into a single consolidated access, improving latency. Coalescing memory is of high importance in CUDA program optimisation [28].

Memory type selection

Selecting the appropriate memory type to use in any application is of great importance when optimising CUDA applications. As each memory type has specific advantages and disadvantages, its use is completely application-dependent. For example, in the case where multiple threads are accessing a single memory location simultaneously, intelligent use of constant memory can alleviate latency and bandwidth issues. Alternately, when the same data is being accessed numerous times, the use of shared memory can increase the access efficiency drastically [28].

Profiling and load balancing

Many tools have recently become available to assist developers in profiling CUDA applications in an effort to reveal optimisation flaws. *Nvidia Nsight* is an example of a CUDA integrated development environment (IDE) with a built-in profiler. Profilers enable simple detection of *bottlenecks*, *occupancy* statistics and many other factors affecting run-time performance of GPU applications. A bottleneck refers to an operation that takes time to complete, blocking the program from continuing until this operation has finished. Understanding and eliminating bottlenecks in CUDA programs is essential for a well optimised application. Occupancy, in a GPU context, is defined as the number of thread groups (warps) that are active at one time. That is, at any point in time, warps may be processing data, or accessing global memory. While accessing global memory, warps are idle for hundreds of instructions while other warps continue processing. Maximising thread occupancy can play a large role in CUDA optimisation [28].

Finally, since a warp consists of 32 threads, it can execute 32 instances of a single instruction with different data. If threads take different control flow, however, such as conditional statements, they are no longer executing the same instruction, and some of those 32 execution resources become idle. This is called *control divergence*. Should a CUDA application exhibit high degrees of control divergence, it is important to implement load balancing in order to keep all execution resources busy within a warp, and eliminate wasted resources [12].

2.3 Summary

Structure or halo-finding is an important step in extracting useful physical data from cosmological simulations in order to assist research in structure formation and galaxy evolution. Many methods have been developed to accomplish this.

Knebe et al. [21] concluded that, at large, different halo-finding codes agree with regards to results and accuracy. Therefore, it seems prudent to instead compare these algorithms on speed and capability. As such, and considering the n-body nature of halo-finding, the implementation of one of these algorithms on a GPU, in an attempt to decrease run-time, seems a logical next step. The hierarchical friends-of-friends method, combined with 6D phase-space, makes a good candidate for this investigation, due to its simplicity and ease of implementation, as well as its high degree of data-level parallelism.

The CUDA programming model was introduced, and the typical execution of a CUDA program discussed. CUDA provides access to a number of different memory types, each with their own uses and advantages. Finally, we discussed a number of methods to optimise CUDA programs, including data-parallel computing, memory selection and coalescence and profiling. Intelligent use of memory types can decrease the application run-time significantly, specifically the use of shared memory for fast access to local thread computation results.

Combining the FoF approach to halo-finding with GPUs, and exploiting the many methods of GPU code optimisation, may result in a decreased run-time for large cosmological simulations.

Chapter 3

Design

This chapter provides an overview of all systems implemented in this research and a description of the friends-of-friends algorithm. It also motivates the design choices for the friends-of-friends algorithm and all data structures and methods employed by each implementation. To begin, we describe the friends-of-friends algorithm and its extension to six dimensions. This is followed with the constraints effecting our system designs, and a description of four prototype application designs. These systems consist of two CPU implementations and two GPU implementations, with the CPU code intended as benchmarks for the GPU programs. Our first CPU and GPU implementations were designed as proof-of-concept feasibility exercises.

3.1 Friends-of-friends algorithm

Each prototype system implements the friends-of-friends algorithm using different data structures and methods, each with specific advantages and disadvantages.

The friends-of-friends algorithm, described in Chapter 2, was chosen specifically for it's simplicity and ease of implementation, as well as its high potential for parallelisation. Particles in the algorithm are considered *friends* if they are separated by less than a predetermined friends-of-friends linking length ℓ. This distance is generally specified in units of the mean inter-particle separation. That is, L/N for a cube with a side length L containing N^3 particles [37]. However, it is generally considered good practice to specify a density constraint for dark matter halo groups. We can define a minimum over-density contour d_{min} for a halo group as follows:

$$d_{min} = \frac{2\Omega_{sim}}{\frac{4}{3}\pi l^3} \tag{3.1}$$

where Ω_{sim} is the over-density of the simulation volume against the background universe. It should be noted that d_{min} is the minimum contour level and that the density of a halo group is usually much larger [37]. The linking length is compared against the Euclidean distance between two particles p_i and p_j, which is determined by the following equation:

$$distance = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2} \tag{3.2}$$

where x, y, and z are position parameters for a particle p in 3-dimensional space. Particles satisfying this condition are linked together and considered *friends*, and all linked particles are considered one halo, provided that they are greater than the minimum number of particles required to form a group, N_{min}. This value can be modified as necessary depending on the group sizes being searched for, and is implemented to prevent the identification of false positives. That is, random particles that are identified as friends, but are not numerous enough to constitute a halo.

Although 3D position-space halo-finders provide good accuracy with regard to structure and sub-structure detection [25], our implementations also search for particle groups in 6-dimensional phase-space, which includes both position and velocity vectors. This produces contrasts that are much more pronounced than in a traditional 3D simulation. This, in turn, allows for a higher degree of accuracy and more robust tracking of halos at a slight cost in efficiency. This is because particles in the same halo generally have similar velocity vectors. As such, tracking these particles in phase-space allows us to eliminate particles whose velocity vectors deviate from the expected velocity, as they are likely members of another halo, thereby increasing halo accuracy. In addition, phase-space halo-finders are capable of detecting sub-halos and other phase-space structures [26] which may be of interest to an astronomer. In a phase-space search, the typical Euclidean distance calculation falls away and a new particle linking constraint is introduced for particles p_i and p_j:

$$(\frac{\vec{x}_i - \vec{x}_j}{l})^2 + (\frac{\vec{v}_i - \vec{v}_j}{\psi})^2 < 1 \tag{3.3}$$

where l is the linking length in position space, and ψ is the linking length in velocity space. It should be noted that for $\psi \to \infty$, the equation reduces to the standard 3D friends-of-friends scheme as shown below [21].

$$(\frac{\vec{x}_i - \vec{x}_j}{l})^2 < 1 \tag{3.4}$$

This can be rewritten as:

$$\sqrt{(\vec{x}_i - \vec{x}_j)^2} < l \tag{3.5}$$

which denotes the standard form for a 3D friends-of-friends linking constraint. A visual interpretation of the friends-of-friends algorithm is described in Figure 3.1.

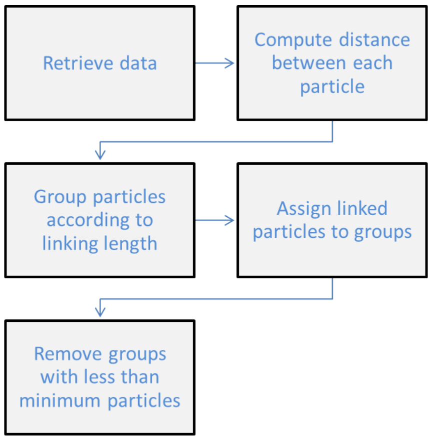
Retrieve data
Compute distance between each particle
Group particles according to linking length
Assign linked particles to groups
Remove groups with less than minimum particles

Figure 3.1: *A broad overview of the steps followed in the friends-of-friends algorithm. We begin by retrieving the data, and computing the distance between each particle. Particles are grouped together and linked when the distance separating them is less than the linking length. Finally, groups with less than the minimum amount of particles are removed.*

3.2 System design constraints

The use of GPU hardware places a number of constraints on application design. These are inherent to GPU hardware architecture, as well as the nature of the problem. Since the CPU implementations will be developed as benchmarks for their GPU counterparts, all application designs must take these constraints into account.

3.2.1 Efficiency

The computational complexity for a naive friends-of-friends implementation is $O(n^2)$ for n particles, as each particle needs to be compared to every other particle in the simulation. This problem is very well suited to GPU implementation due to the large amount of inherent parallelism. Unfortunately, the algorithm does not map to the CPU as conveniently. Methods of implementation on the CPU should reduce this complexity so that it scales linearly with problem size. This is discussed further in 3.2.2.

Furthermore, in order to ascertain a fair comparison of CPU and GPU performance results for this research, it is important that the CPU applications be as efficient as possible. That is, since the CPU applications are being implemented as benchmarks for their GPU counterparts, their run-time must be comparable to currently used commercial halo-finding simulations. This ensures that our results are both realistic and significant. As such, both CPU implementations use well-known and efficient libraries.

Finally, the expected speed-up for the GPU applications has a direct correlation to the parallelism inherent in the problem, and the methods employed in the CPU designs should be as close as possible to those used in their GPU counterparts. As such, it is important that all data-structures and algorithms implemented for the CPU applications be amenable to parallelisation, as this serves as a proof-of-concept, and an exploration of viability for the GPU implementation. This impacts CPU application design, as the most effective method on the CPU may not be amenable to GPU implementation. Hence, an efficient CPU system that can be used as a benchmark, whilst still being amenable to a GPU implementation must be designed.

3.2.2 Scalability

Halo-finder data-sets can vary widely with regards to the number of particles required by the simulation. Methods used in the applications should be able to handle very large numbers of particles and performance should scale linearly with data size. Since the näive FoF algorithm is of $O(n^2)$ complexity, alternative data-structures and algorithms must be used in the CPU benchmark applications in order to reduce computational complexity significantly.

3.2.3 Memory

GPU hardware can be difficult to work with, specifically because of the limited amount of memory available. This is directly related to the scalability constraints described in 3.2.2. As the problem size becomes larger, it becomes increasingly difficult to manage memory resources on the GPU. Hence, for large enough problems, data-structures on the GPU may need to be limited due to the amount of memory available, at the expense of computation time. This is discussed further in sections 3.4 and 6.1.

3.3 CPU design

Two single-core CPU implementations were developed using different techniques and data-structures, as seen in sections 3.3.1 and 3.3.2. This allowed us additional flexibility when implementing an efficient CPU benchmark. The first CPU implementation was intended as a proof-of-concept exercise to familiarise us with the algorithm and its implementation. Then, exploiting lessons learned in this application, we can design and implement a more efficient system as seen in 3.3.2.

3.3.1 Minimum spanning tree implementation

This implementation uses Delaunay triangulation in order to seed a graph which can then be efficiently reduced to a Euclidean minimum spanning tree (EMST). Edges in the EMST with weights exceeding the friends-of-friends linking length l are then removed, leaving a number of separate EMST components, each representing a dark matter halo. The nodes are traversed recursively and placed in their respective halo groups.

Given a EMST of set S, the total length of the set of edges of S is minimal, and the length of an edge is the Euclidean distance between its vertices. Furthermore, there exists only one unique path from one vertex to another [5].

EMSTs provide a functional means of identifying dark matter halos. However, the preprocessing involved in constructing a seed tree which can be reduced to an EMST is challenging, as the tree must contain enough edges that it is certain to contain the EMST, but not too many edges so as to unnecessarily increase computation time.

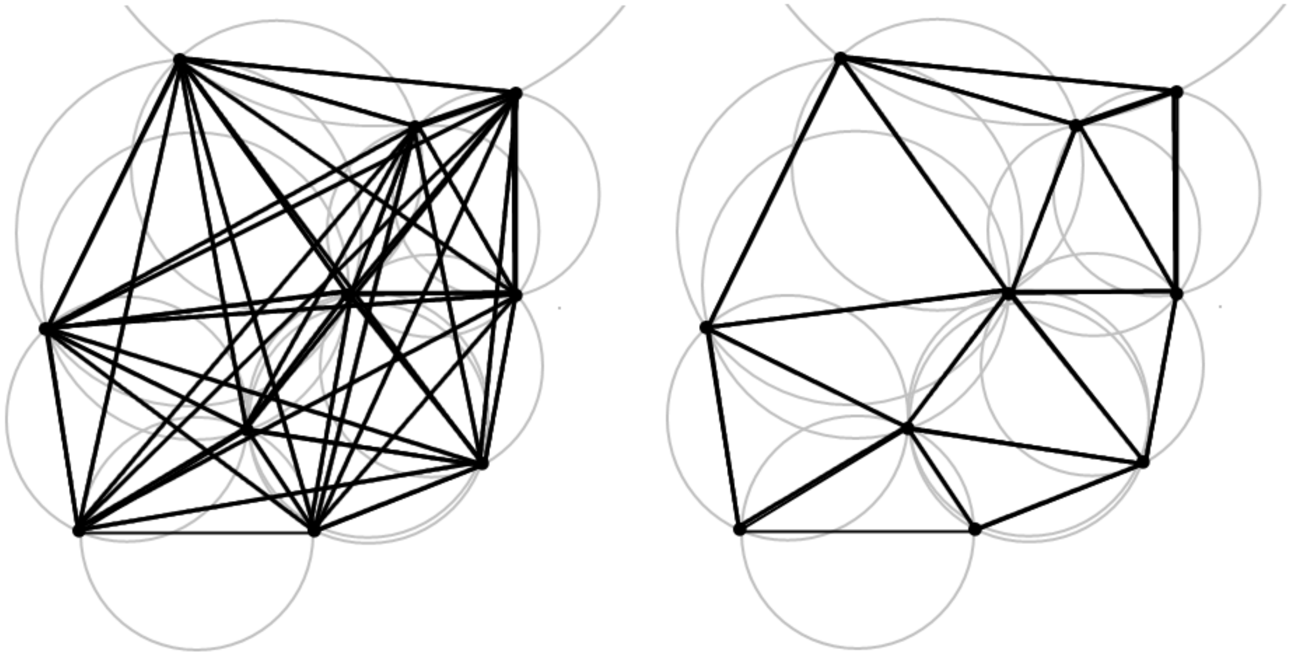

Figure 3.2: *A 2D comparison of number of edges in a complete graph and a Delaunay triangulation. A Delaunay triangulation produces a much-reduced set of edges, enabling a much more efficient transition to a minimum spanning tree.*

Given n points, the easiest way to find an EMST is to construct a complete graph on n vertices, where each vertex is a point in the data-set, and each edge is the Euclidean distance between its vertices. A complete graph contains one edge between every pair of vertices in the graph. That is, a complete graph for n vertices contains $n(n-1)/2$ edges. A typical minimum spanning tree algorithm is then applied. However, simply constructing the EMST in this way is overly complex due to the number of edges, and is not useful as a benchmark for a GPU implementation, as it would provide an unfair comparison due to the increased computation time.

A more efficient approach to constructing an EMST is to do away with the complete graph construction, and perform a Delaunay triangulation on the point set instead. This is due to the fact that a Delaunay triangulation of a set of n vertices contains only n edges.

Given a set P of points in the plane, the Delaunay triangulation is one such that there is no point in P inside the circumcircle of any triangle in the triangulation. A circumcircle of a triangle is a circle that passes through all the vertices of that triangle [10].

Constructing the graph using Delaunay triangulation produces a graph with a greatly reduced set of edges, as depicted in Figure 3.2, whilst still containing the Euclidean minimum spanning tree. Computation time is thus heavily reduced.

Kruskal's algorithm [8] was used to find the EMST. This algorithm was chosen for its simplicity, as well as its better performance on graphs with a small number of edges. Other methods, such as Primm's algorithm, perform better on dense graphs that have many more edges than vertices. Following the computation of the EMST, a representation of the halo groups can be quickly obtained by deleting all edges in the graph that are greater than the FoF linking length. This reduces the graph to a number of disconnected EMSTs. These remaining components form dark matter halos, where each vertex is a particle belonging to that halo. The reduction of the EMST process is shown in Figure 3.3, and the process as a whole is depicted in Figure 3.4.

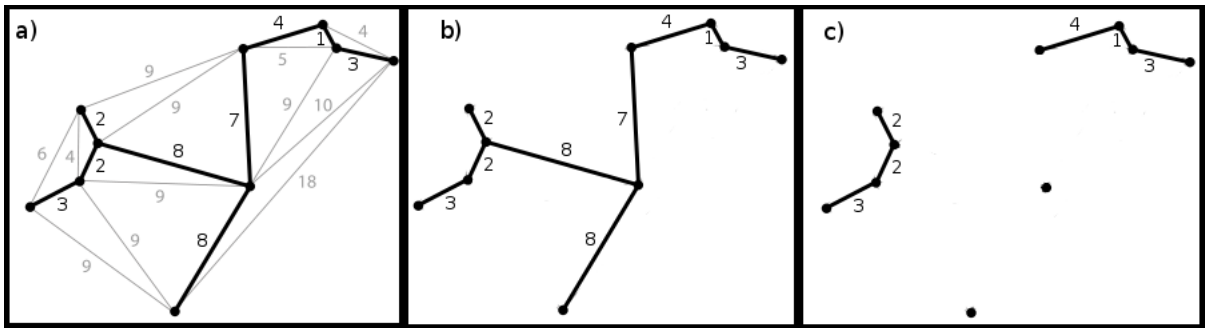

Figure 3.3: *a) A Euclidean minimum spanning tree after redundant edges have been removed, with the removed edges in light gray. b) A Euclidean minimum spanning tree with redundant edges removed. c) A Euclidean minimum spanning tree with edges exceeding linking length l removed. The disconnected EMSTs constitute halos where each vertex is a particle belonging to that halo.*

It is important to note that a EMST contains no cycles. That is, given two nodes, A and B, there exists only one path, following the edges, from A to B. This property is extremely important in the context of the FoF algorithm, as it guarantees multiple disconnected MST components, each constituting a separate halo, following the removal of edges that exceed the FoF linking-length. That is, since the EMST contains no cycles, there exists only a single path from one vertex to any other vertex. Therefore, removing any edge in the graph guarantees two distinct MSTs. Removing all edges with a distance value greater than the FoF linking length results in a number of disconnected EMSTs, each representing a bound set of particles.

This method can be extended to 6 dimensions by replacing the Euclidean distance calculation

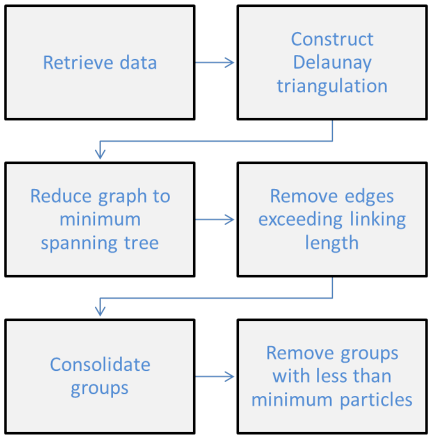
Retrieve data
Construct Delaunay triangulation
Reduce graph to minimum spanning tree
Remove edges exceeding linking length
Consolidate groups
Remove groups with less than minimum particles

Figure 3.4: *An overview of the CPU MST implementation. We begin by retrieving the data and constructing a Delaunay triangulation. The triangulation is then reduced to a minimum spanning tree, and edges that exceed the linking length of the simulation are removed. Following this, vertices that are still linked together are consolidated and added to halo groups, where groups consisting of less particles than the simulation minimum are removed.*

in Equation 3.2 with the 6D Equation 3.3. Once the EMST has been found, edges with a weight less than 1 are deleted, and the separate components are obtained in the same fashion as above.

3.3.2 Kd-tree implementation

This implementation was designed with experience gained from our proof-of-concept CPU implementation, and uses kd-trees in order to spatially partition the problem domain. These data-structures are extremely useful for searches. *Range queries* are computed on each particle in order to find all particles within a FoF linking length. The tree nodes are visited recursively and placed into their respective halo groups. This CPU version improved on the preprocessing and computation abilities of the minimum spanning tree version.

A kd-tree is a binary tree where every node is a k-dimensional point. They are excellent data-structures for search based solutions, due to their ability to rapidly reduce the search space by excluding entire branches of the tree using properties inherent in its space-partitioned construction. This is extremely advantageous for halo-finding, as the data size is often large.

The most common way to construct a kd-tree is to cycle through the axes used to select the splitting hyperplanes. That is, a plane that divides the space into two parts. At the root of a 3D tree, the x-axis is first selected as the splitting plane. To select the point for insertion into the tree, the list of points is divided into two at the median value, and this value is inserted. At the next level of the tree, the same process is applied to the two split lists, except the y-axis is now used as the splitting plane, and at the next level, the z-axis. This method produces a balanced kd-tree. That is, each leaf node is approximately the same distance from the root as all others. This is important, as it maximizes the opportunity for removing large regions of the search space. It is not necessary to split using the median, but this ensures a balanced tree. Although median selection influences computation time, the resultant balanced tree can be better exploited during the tree traversal for range queries. An example kd-tree is depicted in Figure 3.5.

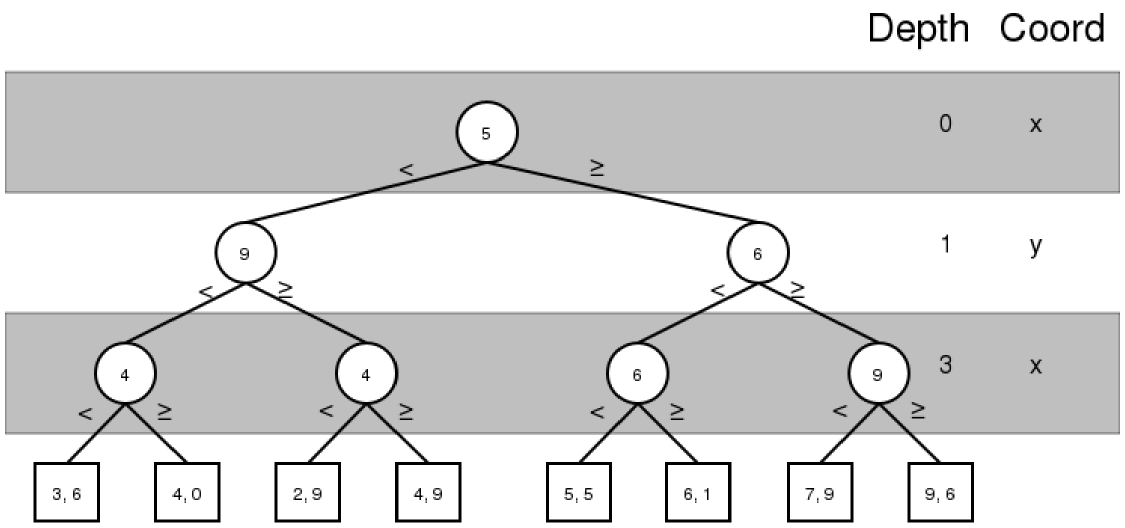

Figure 3.5: *An example of a kd-tree for point set [(6, 1), (5, 5), (9, 6), (3, 6), (4, 9), (4, 0), (7, 9), (2, 9)]. For each successive depth in the tree, a different dimension is used. A kd-tree allows us to quickly remove search spaces (branches of the tree) when searching for specific particles [16].*

Range queries on a kd-tree are performed recursively. A query point, specifying the desired particle position, is specified, and the search begins at the root node. At each node, the Euclidean distance is calculated between the current node and the query point. If this distance is less than the FoF linking length, it is added to the list of nodes within range of the query point. The equations that specify tree branches to be traversed can be seen in Equation 3.6 and 3.7 for the left and right side of the tree respectively. V is the current node in the tree, Q the query point, and ℓ is the FoF linking length.

$$V[\text{splitAxis}] > Q[\text{splitAxis}] - \ell \tag{3.6}$$

$$V[\text{splitAxis}] < Q[\text{splitAxis}] + \ell \tag{3.7}$$

If one of these equations is not satisfied, there is no point within the range of Q in that branch of the kd-tree. Should this be the case, the entire branch may be disregarded, greatly reducing the search space, and minimizing the computation time required. Range queries can be extended to 6 dimensions by replacing the Euclidean distance in Equation 3.2 with that of Equation 3.3, although this increases computation time, due to the additional calculations required. An overview of the system is depicted in Figure 3.6.

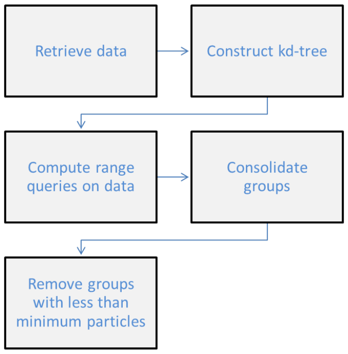
Retrieve data
Construct kd-tree
Compute range queries on data
Consolidate groups
Remove groups with less than minimum particles

Figure 3.6: *An overview of the CPU kd-tree implementation. The data is retrieved, and a kd-tree is constructed. Range queries are computed on the particles and those that are found within range of each other are linked. Lastly, linked particles are sorted into groups, removing those groups with less particles than the simulation minimum.*

3.4 GPU design

Two GPU implementations were developed. The brute-force implementation was developed as a proof-of-concept comparison to a minimum spanning tree CPU implementation. The kd-tree implementation was developed in order to accelerate the range queries performed in the CPU kd-tree application.

3.4.1 Brute force implementation

This implementation uses the GPU to process the set of particles in parallel. It was developed as a comparison to the minimum spanning tree implementation described in Section 3.3.1. Each particle is assigned to a thread and the distance to every other particle in the simulation is calculated. This is optimized through the use of shared memory and a tiling technique. Each particle is assigned to a group, and the CPU identifies dark matter halos accordingly.

When designing a GPU application, it is important to identify the most computation intensive sections of an algorithm. It should be noted that not all types of algorithms are well suited to the GPU. That is, some parts of an algorithm often run better on a CPU. In the case of the FoF method, determining the distance between every particle pair describes an $O(n^2)$ problem. This presents a large amount of potential parallelism, and is the most computation intensive section of the algorithm. Therefore, despite the fact that this could scale poorly on a CPU, it is a good candidate for GPU implementation.

The data is read on the CPU, and copied to the GPU, which computes the distance between each particle. The results are copied back to the CPU, where particle groups are determined. A brief overview of the design is depicted in Figure 3.7.

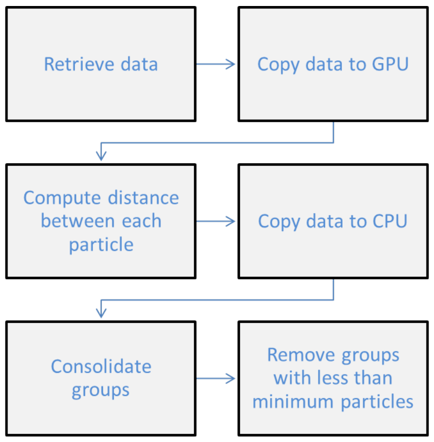
Retrieve data
Copy data to GPU
Compute distance between each particle
Copy data to CPU
Consolidate groups
Remove groups with less than minimum particles

Figure 3.7: *A brief overview of the brute-force implementation. The data is retrieved and copied to the GPU, where particle distances are calculated. The resultant array is copied to the CPU where particle groups can be identified. Groups with less particles than the simulation minimum are disregarded.*

Tiling

While an $O(n^2)$ operation performs relatively well on a GPU due to the degree of parallelism, there are many optimizations that may increase performance further. One of these is to use the *all-pairs* approach to n-body problems together with shared memory [30]. The all-pairs approach places each particle f_{ij} in an NxN grid of pair-wise distances. The distances for particle i can then be obtained by calculating the distance between it and every other particle in row i. However, in order to achieve peak performance via data reuse and avoid issues of limited memory bandwidth, this approach requires some of the computations to be serialized [30]. This is accomplished by introducing the tiling concept, described in Figure 3.8.

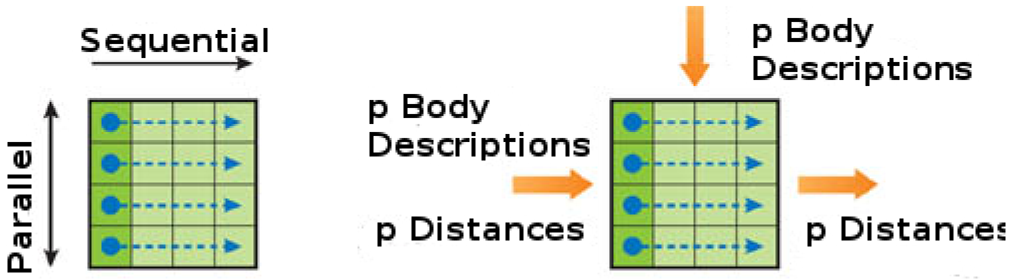

Figure 3.8: *A schematic of a computational tile. Rows are computed sequentially while columns are computed in parallel. This maximises data-level parallelism and allows us to add one tile at a time to shared memory [30].*

A tile is a square region of the NxN grid with p rows and p columns. In order to reuse data, the computation of a tile is arranged such that the interactions in each row are performed sequentially, while each row itself is calculated in parallel [30]. Tiles are clustered into thread blocks. That is, p threads are executed on some number of tiles in sequence. The number of rows in a thread block increases the degree of parallelism, whilst the number of columns increases data reuse. The number of columns also determines the size of bodies data copied to shared memory. Our implementation, as in [30], used square tiles of size p. A description of tile clustering into thread blocks is depicted in Figure 3.9. Particle descriptors for each row are placed into shared memory so that each thread memory access latency is as low as possible.

Memory coalescing

Descriptive data of particles is stored in CUDA's `float4` data type. Using `float4`, as opposed to `float3`, allows coalesced memory access to the arrays stored in GPU memory

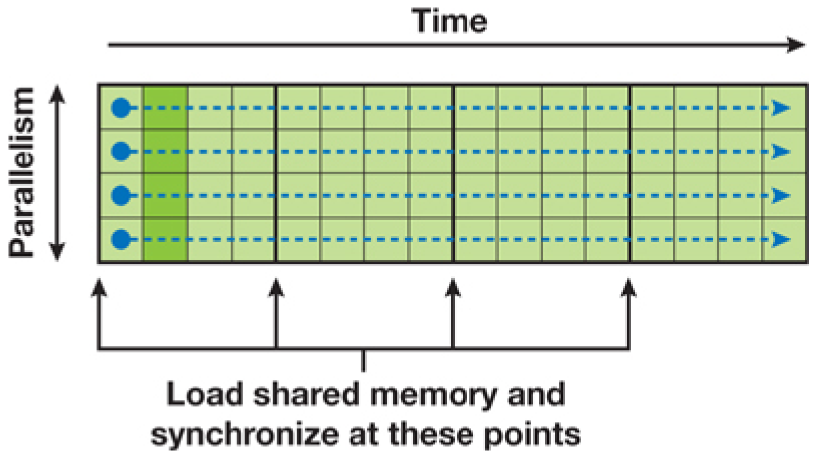

Figure 3.9: *A cluster of tiles. The number of rows in a tile determines the degree of parallelism, while the number of columns in a tile determines the amount of data reuse, and thus the size of data copied to shared memory [30].*

[30]. A coalesced memory transaction is one in which all threads in a warp access global memory at the same time. Simply put, consecutive threads access consecutive memory addresses. This is perhaps the most important factor in optimizing performance on a GPU and this significantly improves efficiency.

Reducing memory

Once the GPU computation is complete, the results must be copied back to the CPU. The easiest solution is to create an array of size n^2, where each element represents a two-particle pairing, and holds the Euclidean distance between every two particles. Returning this to the CPU allows for quick allocation of particles groups by testing the Euclidean distances against the appropriate FoF linking length criterion. However, this method is extremely memory intensive, requiring $O(n^2)$ memory space. Since on-board GPU memory is very limited, we are only able to calculate small data sizes, especially since GPU memory itself must also hold all the data. A better solution is to return an array where each element represents an integer value for each particle, and each integer value represents the particle group, allowing the CPU to quickly determine the group it belongs to, as shown in Figure 3.10. This method uses only $O(n)$ space, requires $O(1)$ look-up time, and scales with data size. The integer value for each array element is determined as a pointer to the group the particle currently belongs to. For our calculation, we adopted the following approach: To begin, each particle is considered its own halo. When a particle discovers a neighbour satisfying the FoF linking condition, the particles link together, and change the values in their respective array elements to the appropriate group number. Once the GPU computation is complete, the CPU can easily allocate each particle to its respective group. However, due to the independent nature of GPU threads and their out-of-order execution, this method presents us with race conditions and a number of other issues. Solutions to these and the method as a whole are discussed in more detail in Chapter 4.

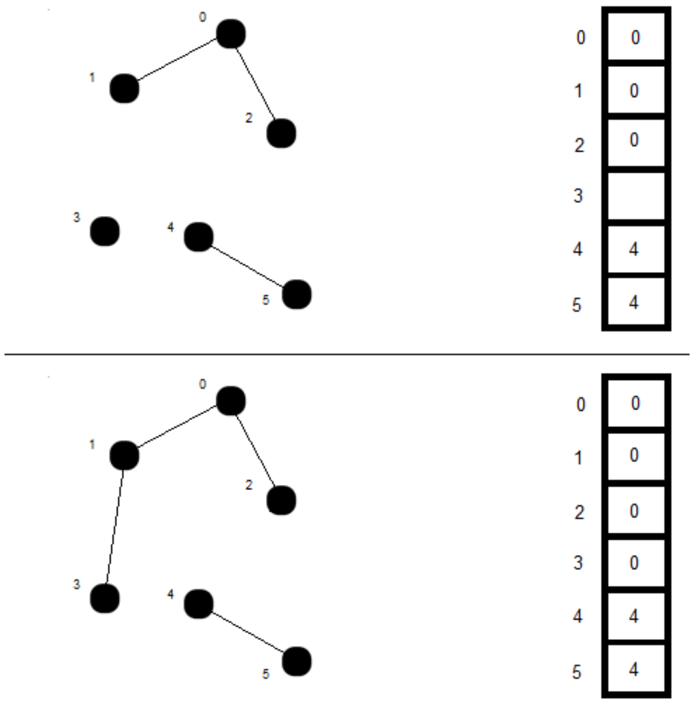

Figure 3.10: *A simplified depiction of how the particles group number is assigned in the array to be returned from the GPU. At the top, particle 1 finds that particle 3 is within range. At the bottom, particle 3 is added to the same halo that particle 1 is part of (halo 0) and the group number is updated in the return array.*

This implementation is extended to 6 dimensions at the expense of computation time. Velocity vectors are stored in a similar manner to position vectors in a 3D implementation, and copied to the GPU. Using Equation 3.3 in place of Equation 3.2, 6D results can be computed on the GPU. However, the additional accuracy 6 dimensions provides comes at the price of speed due to the additional floating point operations required for the computation.

3.4.2 Kd-tree implementation

This implementation builds the kd-tree on the CPU, and uses the GPU to process each range query in parallel. Every particle is then assigned to a halo, and the CPU sorts them accordingly.

Kd-trees are well understood data-structures, and a fair amount of research has been done on their construction, both on the CPU and GPU. The construction of a kd-tree on the GPU is viable, and has been accomplished with some success [44]. This research, however, focuses on accelerating the FoF algorithm, and, in the context of this work, the speed of the range queries performed on the kd-tree. Kd-tree construction on a GPU for this application is left as future work.

This application runs in a similar manner to the application described in Section 3.3.2. The difference, however, is that range queries are computed on the GPU. The data is read, and the kd-tree is constructed on the CPU as described in Section 3.3.2. The tree, as well as a list of query points, are then copied to GPU memory. The list of query points simply contains every point in the simulation, as each point needs to be queried for neighbours with the FoF linking length, and processed accordingly. Each thread on the GPU computes the range query for a specific point, traversing the kd-tree as in Section 3.3.2. These queries are run simultaneously, and an integer array of size n keeps track of the particle's current group, in a manner similar to that described in Section 3.4.1. On completion of the kernel, the integer array is copied back to the CPU, and particles are allocated to their specific groups. An overview for this application is depicted in Figure 3.11.

If the kd-tree is too large to fit in GPU memory, the tree is limited to a certain depth. That is, for large data sizes, the kd-tree construction completes at a specified depth, ensuring that the tree can fit into GPU memory. Particles that would normally be children of the nodes at this depth are simply stored in a list at the leaf node preceding their intended position. Although necessary, this impacts heavily on performance, as once a certain depth has been reached in the tree traversal, no more particles can be excluded from the search, and the remaining nodes must be queried sequentially. This solution can be avoided by using a GPU cluster with enough memory to incorporate a large kd-tree. However, this is left as future work. An example of a depth-limited kd-tree is shown in Figure 3.12.

Extending this application to 6 dimensions is accomplished by replacing the value for the distance provided by Equation 3.2 with that of Equation 3.3. Using this equation, we replace

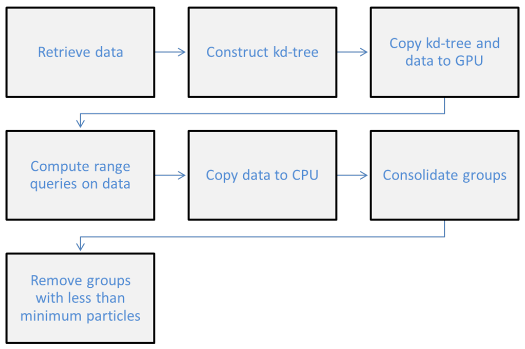
Retrieve data
Construct kd-tree
Copy kd-tree and data to GPU
Compute range queries on data
Copy data to CPU
Consolidate groups
Remove groups with less than minimum particles

Figure 3.11: *A brief overview of the kd-tree GPU implementation. The data is retrieved, and a kd-tree is constructed on the CPU. The kd-tree and data are copied to global memory on the GPU, and range queries are computed on each particle in parallel. A return array is populated with pointers to each particles halo ID, and passed back to the CPU. Here, particles are added to groups, and groups with less than the minimum number of particles for the simulation are removed.*

l in Equations 3.6 and 3.7 with 1, and proceed as normal. Again, the improved accuracy provided by the additional dimensions increases run-time, as more floating point calculations are required.

3.5 Summary

In this section, we described the FoF algorithm to be implemented and how it can be extended to extract halos in a 6D phase-space context. Following this, we examined the efficiency, scalability and memory design constraints inherent in this research. Finally, four prototype systems were designed:

- Using Euclidean minimum spanning trees on a CPU
- Using kd-trees on a CPU
- A brute-force approach using tiling on the GPU
- Using kd-trees on a GPU

The MST CPU and brute force GPU applications were created as proof-of-concept exercises to familiarise us with the problem. They were designed and implemented prior to the other systems. Leveraging knowledge and familiarisation gained during this exercise, we designed

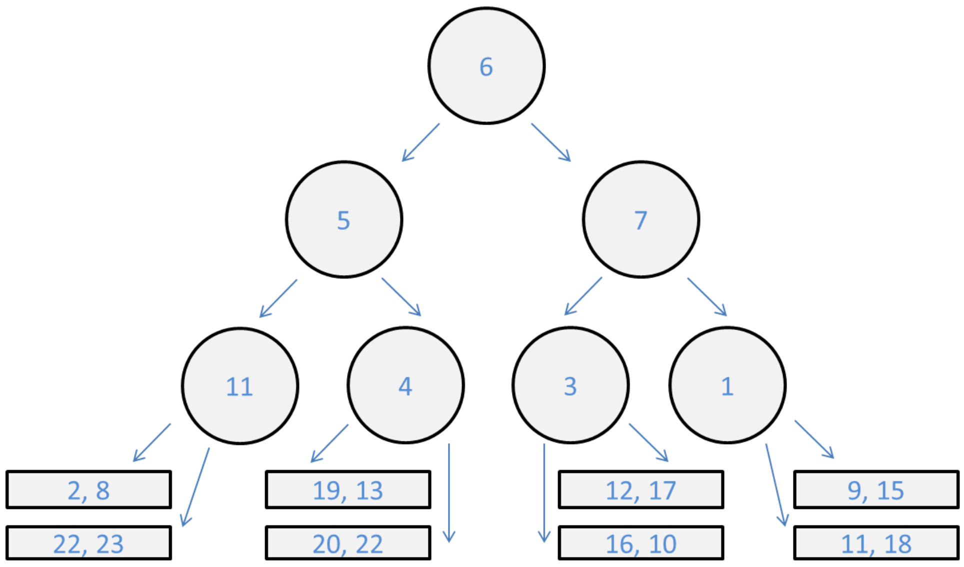

Figure 3.12: *An example of a kd-tree consisting of 23 nodes, where the kd-tree is limited at depth 2. Beginning the search from the root node, we may discard branches of the tree if they do not satisfy Equations 3.6 or 3.7. When we reach a leaf node, removing branches via these equations is no longer an option, and any nodes stored in the list must be searched iteratively.*

a more efficient CPU and GPU system using kd-trees. In the next section, we discuss the implementation details of these applications in greater detail.

Chapter 4

Implementation

A discussion of all methods and libraries used and decisions made to implement the work described in Chapter 3 follows. This chapter also describes the challenges that arose during development, how they influenced decision making and how they were overcome.

We begin by discussing the sequential CPU implementation of the minimum spanning tree method, including data structures and methods used as well as preparation of the data for processing. Post-processing, with regard to consolidating the results and extracting structures, is also described. The implementation of the sequential CPU application using kd-trees follows, with focus on discovering structures using range queries.

Following this is a detailed discussion of the development of a brute force implementation, and a kd-tree implementation, on the GPU. We describe the data-structures and variable types used and contrast these to the sequential CPU versions of the code. We also present the algorithms used in the GPU kernels and explain how processing is split between the CPU and the GPU for maximum acceleration.

The code was written in C and C++ for a Linux 64-bit operating system using a Geforce GTX 460 TI with 1024MB GDDR memory. Version 2.1 of the CUDA tool kit was used for development.

4.1 Minimum spanning tree CPU implementation

We begin the MST implementation by pre-processing the data and preparing it for simulation. A Delaunay triangulation (DT) is extracted from the data to produce a seed graph. The DT is performed using CGAL (Computational Geometry Algorithms Library) [3].

Following this, Kruskal's algorithm is used to reduce the seed graph to an EMST using the Boost Graph Library (BGL) [2]. The edges in the EMST are analysed and edges that do not satisfy the minimum linking length criterion described in Section 3.1 are deleted. Finally, structures constituting dark matter halos are extracted from the resulting graph.

4.1.1 Data-structures

The data for all implementations consists of the number of particles, separated in to dark particles, star particles and gas particles, and the dimensionality of the simulation. Each particle in the simulation is described along the following dimensions:

- mass
- position
- velocity

In the context of these simulations, however, we require only the position and velocity parameters of the particles. Conceptualising each particle as the vertex of a graph, we can define all vertices with the following characteristics:

- `int index` parameter for identification
- `float x, y, z` position parameters
- `float i, j, k` velocity parameters
- A parameter `int colour` denoting the group that the particle belongs to

A vertex with the above description is sufficient to perform a DT on the data using CGAL. The resulting triangulation contains the properties for each vertex and a list of edges. A transition from a DT to a EMST using BGL requires only the edges, the vertices they are connected to and a corresponding list of weights for each edge. As such, we can define an array of pairs for the edges, where the elements in a pair describe the beginning and end vertices of an edge, and an array of floats containing the corresponding weights. Therefore, a graph structure from which an EMST can be found can be defined as follows:

- An array `edge_array` containing the edges of the graph
- An array `weights` containing the weights for the edges
- A value `num_nodes` denoting the number of vertices in the graph

While providing functionality for reducing the graph to an EMST, the data-structure also simplifies the deletion of edges that do not satisfy the minimum linking length constraint for the simulation. Finally, by utilising the `adjacent_vertices` iterator in the BGL, which allows for easy access to adjacent vertices via connected edges, valid result structures can be extracted in an efficient manner.

4.1.2 Delaunay triangulation

CGAL is a well known library that provides efficient and reliable geometric algorithms. Most importantly, it offers data structures and algorithms for Delaunay triangulations in both 2D and 3D, with customisable base structures that allow for the addition of properties relevant to the context in which the algorithm is being used. In the context of this implementation, the typical `float` array containing particle position parameters passed to the DT is extended to allow for the addition of the `index` and `colour` parameters, which are used to represent particles and hold their group number, respectively.

When inserting a new point into the DT, CGAL first uses the `insertion` member function of a basic triangulation. That is, the point is inserted, and the containing triangle, created by the surrounding three vertices, is recreated into three triangles, by connecting each vertex to the newly inserted point. CGAL then performs a sequence of flips to restore the Delaunay property. That is, there is no point in P inside the circumcircle of any triangle in the triangulation. For data points that are distributed uniformly and at random, each insertion operation takes time $O(1)$ on average [3]. However, the data points for our simulations, although random, are not uniformly distributed. This results in an unavoidable performance hit.

The CGAL DT assists in initialising a seed graph that can be used to find the EMST. However, it only computes the edges of the graphs for their relevant vertices. Other properties necessary for the graph creation, such as edge weightings, are computed separately.

4.1.3 Reduction to EMST

The Boost Graph Library (BGL) is a generic C++ interface which allows access to a graph's structure, while hiding its implementation. Each algorithm is implemented in a data-structure neutral manner. That is, only a single template function operates on many different classes of containers. As such, the library is relatively simple to use and provides flexibility. The BGL was used to implement Kruskal's algorithm to compute an EMST on a graph G.

As discussed in Section 4.1.1, to fully initialise a graph G on which we may employ Kruskal's algorithm, we require an array of the edges, the weights for the edges, and a value denoting the number of vertices in the graph. The first is easily obtained from the DT, and the third was obtained during the pre-processing of the data; the number of particles in the simulation. The weights for the edges can be computed by using Equation 3.2, where the position coordinates are obtained from each edge's relevant vertices, and passed to the graph structure.

BGL implements Kruskal's algorithm for finding MSTs in the manner described by the Boost Graph Library documentation [2].

The edges in the EMST are sent to the output iterator w. Should an edge that connects two vertices in different trees be found, the algorithm merges them into a single tree and adds the edge to T. The algorithm uses *union by rank* and *path compression* in order to efficiently compute the disjoint set operations, `MAKE-SET`, `FIND-SET` and `UNION-SET`.

4.1.4 Result structure consolidation

Following the completion of the EMST, the friends-of-friends algorithm can be applied. The edges in the graph are examined, and any edge not satisfying the minimum linking length constraint is deleted. This reduces the graph to a number of disjoint components, where each component contains all the vertices belonging to a single halo structure. These vertices are visited recursively in a manner described in Algorithm 1.

Each vertex in the graph contains the position data for the particle it represents, as well as an integer colour field, which denotes the particle's group. Each vertex in the graph is visited, and its colour is checked. If the colour is 0, that vertex has not been processed, and therefore, nor has any vertex belonging to the same group. This is because the recursive nature of the

Algorithm 1 Consolidate groups MST

```
 1: procedure FindGroupsMST(Graph $g$)
 2:     vector $group$
 3:     vector $listOfGroups$
 4:     int $groupCounter = 0$
 5:     for all vertices in graph $g$ on vertex iterator $it$ do
 6:         $it \leftarrow currentVertex$
 7:         if $it$.colour $= 0$ then    ▷ Uncoloured vertices have not been assigned to a group
 8:             $groupCounter{+}{+}$
 9:             Recursive($it$, $g$, $group$, $groupCounter$)    ▷ Traverses entire group
10:         end if
11:         if $group.size() \neq 0$ and $group.size() > 20$ then
12:             $listOfGroups.add(group)$
13:             $group.removeAll()$
14:         end if
15:     end for
16: end procedure

17: procedure Recursive(VertexIterator $it$, Graph $g$, Vector $group$, int $groupCounter$)
18:     $it.colour \leftarrow groupCounter$
19:     $group.add(it)$
20:     for all vertices connected to $it$ on vertex iterator $con$ do
21:         if $con.colour = 0$ then
22:             Recursive($con$, $g$, $group$, $groupCounter$)
23:         end if
24:     end for
25: end procedure
```

procedure ensures that every particle belonging to the same group is processed before the stack is empty. The vertex is passed to the recursive function where the entire group is processed. Each vertex visited has its colour field set to the same integer `groupCounter`, and added to the `group` vector. After each recursive call, `group` is added to `listOfGroups`, and cleared.

On completion of Algorithm 1, `listOfGroups` contains a list of each vertex in each group structure. Groups containing less than the minimum amount of particles per halo are not listed here, as they were not added. This ensures a valid list of extracted structures for the current data.

4.1.5 Conversion to 6D

In a 6D simulation, we make use of the velocity parameters described in Section 4.1.1. It is important to note that in the context of this implementation, the inclusion of these parameters only affects the weighting of the edges, and the linking length. This is because the edge weighting computation is modified to use Equation 3.3. As described in this equation, the linking length becomes 1.

Once the modified distances have been computed and the linking length adapted, we achieve results for a 6D phase-space simulation. All remaining sections of the implementation are unaffected.

4.2 Kd-tree CPU implementation

As in Section 4.1, we begin the kd-tree implementation by pre-processing the data and preparing it for simulation. A kd-tree is constructed on the data to facilitate the use of range queries for finding particles within linking length proximity.

Following this, we use a recursive algorithm to process range queries on each particle in the simulation, using the kd-tree. Each range query discovers all other particles within the linking length of the query particle. Finally, structures constituting dark matter halos are extracted from the range query results.

4.2.1 Data-structures

The data-structure used to store particle information remains the same as that in the minimum spanning tree implementation. The number of particles, separated in to dark particles, star particles and gas particles, and the dimensionality of the simulation are extracted from the data file. Each particle in the simulation is described along the following dimensions:

- mass
- position
- velocity

The construction of the kd-tree follows. The kd-tree contains an array of kd-nodes, where each node relates to a particle in the simulation. Each kd-node contains the following:

- The split axis
- The split value
- The ID linking the node to its corresponding particle

The split axis value indicates the current dimension that the node was split on. That is, whether the node was split on the x, y, or z axis for a 3D simulation. The split value indicates the position value for that node on that split axis. Finally, the ID links the node to the particle it belongs to. This is important, as it allows us to access any additional information about that particle that may be relevant.

While computing range queries, an array of integers and an array of floats are necessary in order to hold the indices of the particles found to be within range, as well as to hold the distances of those particles from the original query node.

4.2.2 Constructing the kd-tree

Approximate Nearest Neighbour (ANN) is a library which supports data structures and methods for exact and approximate nearest neighbour searches [1]. In the context of this implementation, it was used to efficiently build the kd-tree, and perform range queries in 3 dimensions. The algorithm used in ANN for kd-tree construction is based on the algorithm presented in [13]. The procedure makes use of a simple divide-and-conquer technique, which determines an appropriate orthogonal cutting plane on which it splits the points.

The construction of the kd-tree provides the ideal platform on which to perform range queries, as large portions of the kd-tree may be removed from the search should they not satisfy branching conditions.

4.2.3 Range queries

Since ANN does not explicitly define a range query operation, it is necessary to perform a nearest neighbour search with a fixed radius and to add minor modifications. A nearest neighbour operation performed by ANN is given a query point q, a non-negative integer k, an array of point indices `nnIdx` and an array of distances. The procedure computes the k nearest neighbours to q in the point set, and stores the indices of the nearest neighbours relative to the point array `pa` given in the constructor. The nearest neighbour is stored in `nnIdx[0]`, the second nearest in `nnIdx[1]`, and so on. The squared distances to the corresponding points are stored in the array `dists`.

In order to compute a nearest neighbour query with a fixed radius, an additional fixed radius parameter `sqRad` is required. However, the search implemented in this library is not a true fixed-radius search, as it only computes the closest k points that lie within the radius bound. Thus, if the value of k is less than the total number of points in the radius bound, the farthest points within the radius will not be reported. However, regardless of the value of k, the procedure always returns a count of the total number of points that lie within the radius bound. Therefore, in order to produce a true fixed-radius search, we can first set $k = 0$ and run the procedure to obtain the number of points within the radius. The procedure can then be run again with k set to the number of points within the radius [1]. This approach, however, results in running the full length of the procedure twice as many times as would normally be necessary.

This implementation uses a recursive procedure to perform range queries on all points in the simulation, and is described in Algorithm 2.

Algorithm 2 describes the consolidation of halo groups for a given data-set using range queries. Once the kd-tree has been constructed, we iterate over each query point in the data set, and pass it to the recursive function. This function first calls the nearest neighbour search for that particle, and then calls itself for each particle found within the radius bound. When nodes are found, they are coloured as in Algorithm 1, and added to a group vector in a similar fashion.

On completion of Algorithm 2, as in Algorithm 1, `listOfGroups` contains a list of each vertex in each halo. Groups not satisfying the minimum particles per group constraint are not listed here, as they were not added. This ensures a valid list of halos for the current data.

Algorithm 2 Consolidate groups kd-tree

```
1: procedure FindGroupsKD(KDTree $g$, float $sqRad$, Vector $queryPoints$)
2:     vector $group$
3:     vector $listOfGroups$
4:     int $groupCounter = 0$
5:     for all nodes in kd-tree $g$ do
6:         if $node$.colour = 0 then                ▷ Uncoloured vertices not yet processed
7:             $groupCounter{+}{+}$
8:             Recursive($g$, $queryPoints$, $sqRad$, $groupCounter$, $group$, $currentQuery$)
9:         end if
10:        if $group.size() \neq 0$ and $group.size() > 20$ then
11:            $listOfGroups.add(group)$
12:            $group.removeAll()$
13:        end if
14:    end for
15: end procedure

16: procedure Recursive(KDTree $g$, Vector $queryPoints$, float $sqRad$, int $groupCounter$,
    Vector $group$, int $currentQuery$)
17:     $currentQuery.colour \leftarrow groupCounter$
18:     $group.add(it)$
19:     array $ids \leftarrow$ RangeQuery($g$, $currentQuery$, $sqRad$) ▷ Gets ID for all nodes in range
20:     for all IDs in array $ids$ do
21:         if $ids[i].colour = 0$ then
22:             Recursive($g$, $queryPoints$, $sqRad$, $groupCounter$, $group$, $ids[i]$)
23:         end if
24:     end for
25: end procedure
```

4.2.4 Conversion to 6D

The ANN library was modified in a number of ways to add 6D functionality. A procedure was added to retrieve the velocity vector information and pass it to the nearest neighbour routine. Furthermore, the procedure for calculating the distance between particles was modified, replacing the standard 3D euclidean distance calculation with the 6D distance in Equation 3.3.

Converting to 6D phase-space requires minor modifications to the ANN library nearest neighbour routine. Changes to the kd-tree construction were not required, despite the additional velocity dimensions. This is because the search range in 6D phase-space is still a subset of the original 3D linking length. This is shown below.

Consider a combination of Equations 3.2 and 3.3:

$$1 > (\frac{\vec{x}_i - \vec{x}_j}{l})^2 + (\frac{\vec{v}_i - \vec{v}_j}{\psi})^2 > (\frac{\vec{x}_i - \vec{x}_j}{l})^2 \tag{4.1}$$

This is true as the velocity vector term is squared, and thus, always positive. Therefore, the 6D search radius is a subset of the 3D linking length. This means that no particles are missed during the tree branching decisions found in the nearest neighbour search.

4.3 Brute force GPU implementation

This system is based on work done in [30] and begins by pre-processing the data and preparing it for simulation. That is, the data is read from file and stored in memory. The appropriate data-arrays are then allocated on the GPU, and the data is copied to them.

The GPU kernel exploits the speed of shared memory over the latency of global memory to compute the distances between each particle. Should two particles be within the appropriate linking length of each other, the resultant array is modified accordingly.

On completion of the kernel, the resulting integer array, containing grouping information for each particle, is copied back to the CPU. This data is used to consolidate particle grouping on the CPU, allowing the extraction of structures constituting dark matter halos.

4.3.1 Data-structures

Due to the small amount of pre- and post-processing required for a brute force approach, and the simplicity of the algorithm, the data-structures required in this system are relatively straight-forward. As in Section 4.1.1, each particle in the simulation is described as follows:

- mass
- position
- velocity

The data is stored in arrays of type `float4` so as to maximise memory coalescence, as described in Section 3.4.1, before being copied to GPU memory. Here, the data is used to compute a resultant array from which dark matter structures may be extracted.

4.3.2 The brute force GPU kernel

The simulation data is read from file and assigned to host storage arrays before being copied to the GPU. The GPU kernel accepts a number of parameters as arguments: An array of type `float4`, containing the position parameters of the simulation particles; an integer array to hold the results; the total amount of particles to be processed; and an integer denoting the number of particles we wish to store in shared memory at a time. A full description of the kernel is shown in Algorithm 3.

An encompassing `for` loop surrounds the computation performed by the kernel. This loop allows us to perform calculations using the particles currently loaded into shared memory, and governs the time at which this data should be replaced by particles not yet processed. This is necessary because shared memory is not large enough to contain all particles in the simulation at one time. At the beginning of each iteration of the loop, the data stored in shared memory is updated. As each thread is responsible for inserting one particle into shared memory, all threads must be synchronised at this point to ensure that the appropriate data has been copied.

For each load of shared memory, every thread operates on its particular particle, and sequentially computes the distance between this particle, and each element currently in shared memory. If the particles are found to be within the appropriate linking length of the simulation, the result array is updated accordingly. Note that the resultant array is *chronologically* initialised so that each element contains its index. That is, element 0 contains an integer 0, and element 1 contains an integer 1, and so on.

Updating the resulting integer array in such a way that we may later extract structures works as follows: Each element in the array correlates to the same element in the `float4` particle data array. To begin, we conceptualise each particle as its own halo, as well as the *leader* of its halo: This is accomplished by chronologically initialising the integer array as described above. Every time another particle is found to be within range, the two particles are linked. Linking can be conceptualised as merging two separate halos. In order to link, the leaders of both halos must be found. Once they have been discovered, the correlating elements in

Algorithm 3 Brute force GPU kernel

```
1: procedure BruteForce(float4 $particles$, int* $group$, int $n$, int $p$)
2:     allocated $shared$ memory for $p$ particles
3:     float4 $currentPos$
4:     int $i$, $tile$, $j$, $targetPart$
5:     bool $distance$
6:     int $curPosID = currentThread$
7:     for $i = 0$ increasing by $p$ and $tile = 0$ increasing by 1 do
8:         load $p$ particles into $shared$
9:         syncThreads()
10:        for $p$ particles currently in $shared$ increase $j$ do
11:            CheckDistance($distance$)
12:            if within distance then
13:                $target = i + j$
14:                int $targetCur$, $targetBack$, $selfCur$, $selfBack$
15:                $targetCur \leftarrow target$
16:                $targetBack \leftarrow group[target]$
17:                $selfCur \leftarrow curPosID$
18:                $selfBack \leftarrow group[curPosID]$
19:                while $selfCur \neq selfBack$ or $targetCur \neq targetBack$ do
20:                    $targetCur \leftarrow targetBack$
21:                    $targetBack \leftarrow group[target]$
22:                    $selfCur \leftarrow selfBack$
23:                    $selfBack \leftarrow group[selfCur]$
24:                end while
25:                if $selfBack \neq targetBack$ then
26:                    if $selfBack < targetBack$ then
27:                        $group[targetBack] \leftarrow selfBack$
28:                    else if $selfBack > targetBack$ then
29:                        $group[selfBack] \leftarrow targetBack$
30:                    end if
31:                end if
32:            end if
33:            syncThreads()
34:        end for
35:        syncThreads()
36:    end for
37: end procedure
```

the result array are changed to indicate a new halo leader. This process is described in more detail below.

A visual depiction of this is shown in Algorithm 4.1. In order to update the result array, we must first check whether either particle is already linked to another particle. To do this, we check the elements corresponding to each particle in the results array. If particles reference themselves, they are not already linked, as they are still single particle halos. In this case, we simply link the two particles so that they form their own group. To do this, we check which particle has the higher index, and we make it the *leader*. Alternatively, if one of the particles is already linked, it is necessary for us to determine the leader of the group that the particle is linked to. This is accomplished by traversing backward through the particles in the results array until we find a particle referencing itself. We then link this leader with the unlinked particle in the same way as we did when neither particle was linked. The element with the greater index becomes the leader and the two particles are linked. If the particle with the smaller index was the leader, it is no longer considered so, but is instead considered linked to the new leader.

The leader is always the particle with the greater index in order to ensure that we avoid race conditions arising from the out-of-order execution of CUDA threads. Consider the following example: Particle A and particle B are within range of each other, and each is the leader of its own halo. A's thread and B's thread are executed in parallel, and both A and B link to each other. If we did not assign the particle with the highest index value as the leader, but instead, assigned the particle that each thread is comparing itself to as the leader, the two particles would both make the other particle the new leader. This race condition causes a circular reference in the array pointers. However, making the particle with the highest index the new leader ensures that both A and B will choose the same leader, regardless of thread order execution.

Additionally, it is worth noting that only leaders of a halo are ever linked together. When two non-leader particles consider each other, their respective leaders are always searched for and compared instead. Their array element pointer is then appropriately modified to assign a new leader.

Once all particles have been processed, the resultant integer array is copied back to host memory, where the data can be consolidated.

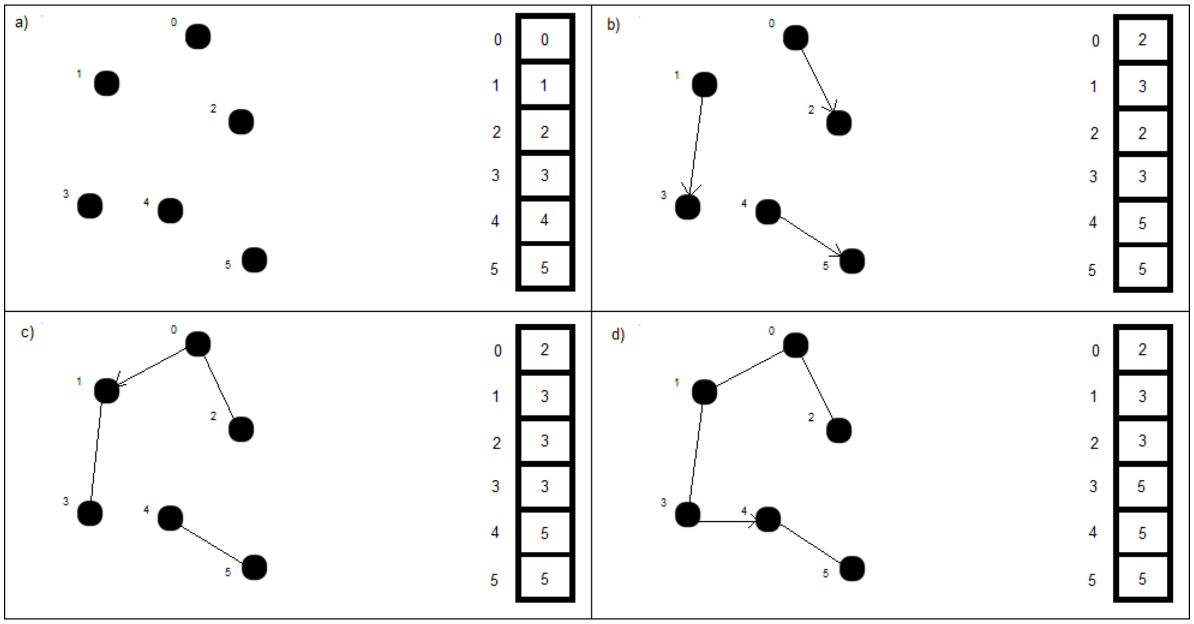

Figure 4.1: *A simplified depiction of how the particles group number is assigned in the array to be returned from the GPU: a) Each particle is considered its own halo, and its corresponding array element points to itself. b) Particles that are within linking length range are linked and merged into one halo. The particle with the greater index becomes the new leader of the halo, referencing itself. The other particle will reference the leader. c) Two halos merge. Each halo searches for its leader (in this case, particles 2 and 3), and the leaders reference the particle with the higher index. Therefore, the two halos merge with 3 as the leader. d) Again, two halos merge. 5 becomes the halo leader. Note that in order to find a leader, one simply needs to iterate through the array pointers until an element references itself.*

4.3.3 Result structure consolidation

Consolidating data from the resultant integer array is accomplished in a similar manner to the group consolidations of the CPU implementations. The method, described in Algorithm 4, accepts three parameters: two zeroed integer arrays, and the result array.

The method iterates over each element in the result array, and keeps track of processed elements by changing the relevant zero value in the `processed` array to 1. If a particle has not been processed, it is passed to the recursive function. This function increments a counter element, signifying 1 additional particle in the halo. The new particle is marked as processed, and the counter is incremented again. The function terminates once an element is found to be referecing itself, and the incrementing counter is added to the `zeros` array

Algorithm 4 GPU group consolidation

```
procedure ConsolidateGroupsGPU(int* $zeros$, int* $processed$, int* $group$)
    int $inc \leftarrow 0$                    ▷ increments amount of particles in current group
    for iterate over all elements in $group$ on $i$ do
        $inc \leftarrow inc + 1$
        if $processed[i] = 0$ then
            Recursive($zeros$, $processed$, $group$, $i$, $inc$)
            $inc \leftarrow 0$
            $zeros$ reset                     ▷ reset array to zeros
        end if
    end for
end procedure

procedure Recursive(int *$zeros$, int* $processed$, int* $group$, int $i$, int $inc$)
    if $processed[i] = 0$ then
        $inc \leftarrow inc + 1$
        $processed[i] \leftarrow 1$
    end if
    if $group[i] = i$ then
        $zeros[i] \leftarrow zeros[i] + inc$
    else
        Recursive($zeros$, $processed$, $group$, $group[i]$, $inc$)
    end if
end procedure
```

element correlating to this particle.

Once this process has completed, the number of halo groups can be assessed by counting the number of elements in the `zeros` array that are not 0. These non-zero elements also indicate the number of particles in each halo.

4.3.4 Conversion to 6D

Converting to 6D phase-space for this implementation requires modifications to data retrieval, the GPU kernel and the distance function.

Additional velocity data is stored in a similar fashion to the position data and is also copied to the GPU prior to kernel invocation. The additional velocity vectors are treated in the same manner as the position data. That is, as they provide additional accuracy when evaluating the linking length, every time the position data is used, the corresponding velocity vector is also required. However, the addition of this data requires more shared memory. As shared memory is already at maximum capacity, it is necessary that half the space is reserved for velocity information. This limits the effectiveness of using shared memory to accelerate our sequential calculations. However, such performance hits are expected when attempting to increase the accuracy of the simulation.

Finally, the distance evaluation function must be changed in order to reflect Equation 3.3. All other sections of the implementation remain unchanged.

4.4 KD-tree GPU implementation

We begin the kd-tree GPU implementation by pre-processing the data and preparing it for simulation in the same manner as Section 4.1. A kd-tree is constructed on the data before being passed, along with a list of query points, to the GPU for computation. Following this, the GPU uses the kd-tree to compute range queries on the data and an array of integers representing each particle's group number is returned to the CPU. Finally, the CPU extracts structures constituting dark matter halos from the resultant array.

This implementation is based on work done by Nghia Ho [15]. The original code provided a basic kd-tree construction on the CPU and the mapping of the tree to the GPU. It also provided a nearest neighbours implementation for the GPU. Since we required range queries, this functionality was removed. The kd-tree construction was modified for efficiency purposes and additional fields were added to the resultant kd-nodes.

4.4.1 Data-structures

The data is read and stored prior to kd-tree construction as described in Section 4.1.1. That is, each particle in the simulation can be described according to the following:

- mass
- position
- velocity

The kd-tree is constructed on this data using only the position parameters in both the 3D and 6D cases. As such, we can define a kd-tree data structure with the following attributes:

- A `kdNode` array consisting of all nodes in the tree
- An `integer` array denoting indices of each node
- A `Point` array consisting of all points in the tree
- An `integer` representing the number of points
- An `integer` denoting the dimension

Each kd-tree node requires the following members:

- An `integer` representing the nodes position in the original data array
- An `integer` denoting the its level in the kd-tree
- Pointers to the node's parent node and its left and right children
- A `float` denoting the value the node was split on
- An `integer` representing where the nodes indices begin in the indices array of the kd-tree
- An `integer` denoting the number of indices the node contains

A kd-tree node with the above description is adequate for the purposes of this implementation. Note that we do not need to store the split axis of each node, as it can be quickly calculated using the nodes current level and the dimension of the tree. Using these data-structures, all data is stored efficiently. It is important that we store only what is required, so as to minimize the amount of memory required on the GPU. Thus, we aim to recompute any derived values, as given the speed of GPU computation when compared to memory latency, this operation can be considered relatively cheap.

4.4.2 Constructing the kd-tree on the CPU

The kd-tree in this implementation was not constructed using the ANN library, as the library does not build the nodes with the required fields for GPU mapping and computation. The original code provided a procedure to construct the kd-tree. However, this procedure was modified for reasons described below. The original code is described in Algorithm 5.

The dimension of the kd-tree and a list of points are taken as parameters. The struct `kdNode` contains the nodes current depth in the tree, the `split_axis` for the current depth, a `rightList` that contains an ID for immediate children on its right side, and a `leftList` that does the same for children on its left side. The root of the tree is initialized first as a starting point, and the rest of the tree is processed recursively.

Algorithm 5 Kd-tree construction

```
1: procedure CONSTRUCT(int $dim$, set $points$)
2:     $vector$ toVisit                                   ▷ Keeps track of nodes still to process
3:     $kdNode\ root$
4:     find $split\_axis$ median in $points$
5:     split $points$ on median into $root.leftList$ and $root.rightList$
6:     $root \leftarrow median$
7:     $root \leftarrow depth$
8:     toVisit.add(root)                                   ▷ Add root to get started
9:     while toVisit.size $\neq 0$ do
10:        $currentNode$ = toVisit[0]
11:        $kdNode\ leftNode,\ rightNode$
12:        $split\_axis \leftarrow (currentNode.depth \mod dim)$          ▷ Determines current axis
13:        $leftNode \leftarrow$ median of $currentNode.leftList$ on $split\_axis$
14:        $rightNode \leftarrow$ median of $currentNode.rightList$ on $split\_axis$
15:        split $currentNode.leftList$ and $currentNode.rightList$ on medians
16:        $leftNode \leftarrow splitLeftLists$         ▷ $leftNode$ gets new $leftList$ and $rightList$
17:        $rightNode \leftarrow splitRightLists$       ▷ $rightNode$ gets new $leftList$ and $rightList$
18:        toVisit.add($leftNode$, $rightNode$)                ▷ Add to process list
19:        toVisit.remove($currentNode$)                       ▷ $currentNode$ is finished, remove
20:    end while
21: end procedure
```

This method was found to be inefficient. Sorting each list on a different `split_axis` for every node is a lengthy process, and so requires a significant amount of computation time. In order to speed up this process, we modified the procedure by using the `nth element` function from the C++ standard template library, which has an average computational complexity of O(n) [11]. This function rearranges the elements in the list in such a way that the element at the n^{th} position in the list is the element that would be in that position in a sorted sequence. In this implementation, the n^{th} element is the length of the list divided by 2. The other elements in the list are not left in any particular order, with the caveat that no element preceding the n^{th} element is bigger than the n^{th} element, and no element following it is less. This is ideal, as the preconditions for splitting the lists for the `left` and `right` nodes are still satisfied, while computation time is minimised.

4.4.3 Range queries on the GPU

Once the kd-tree has been constructed, the appropriate memory space can be allocated in GPU memory and the data copied across. In this implementation, we allocate memory for an array of kd-tree nodes, an array for the indices in the kd-tree, the expected integer result array and an array of points where we can store the original particle coordinates. The array of original particles is used in two ways. Firstly, should we require additional information for a specific kd-tree node, it can be retrieved using the indices array and accessed there. Each kd-tree node contains only the value that the node was split on. That is, if the node was split on the x axis, the split value holds the x value of that node's 3D point. Should we require the y and z coordinates to perform a distance calculation, these values can be accessed in the array of particle coordinates. Secondly, the array doubles as a list of query points. That is, it contains all points for which we need to perform a range query on. The reuse of this array allows us to store the data only once, saving memory space on the GPU. Furthermore, using a single array of integers to store results, where each particle in the simulation requires only one element of the array, minimises the amount of data that needs to be transfered back to the CPU, thus decreasing run-time.

In order to search the entire problem space, a range query must be computed on each particle in the simulation. On GPU kernel initiation, each particle is assigned to a separate thread as a query point. These threads run in parallel and compute each range query at approximately the same time. A description of a single thread range query computation can be seen in Algorithm 6.

Algorithm 6 takes in a number of parameters: The arrays of kd-tree nodes, indices and original points; an integer `cur` denoting the current within-range particle we are considering; the query point for the current thread; an integer `self` to identify the current query points position in the original point array `pts`, a chronologically numbered integer array to hold our results and the current query range we are considering as well as the dimension of the problem.

The procedure begins by creating a stack for simulated recursion. Simulating recursion was a necessity when using CUDA Toolkit version 2.1, as recursion is only supported in version 3.1 and higher. Following this, we begin with the root node of the kd-tree. The current node's

Algorithm 6 Kd-tree GPU kernel

```
1: procedure RangeQuery(const CUDA_NODE* $nodes$, const int* $indices$, const Point* $pts$, int $cur$, const Point $query$, int $self$, int* $group$, float $range$, int $dim$)
2:     float $distance \leftarrow range$
3:     float $d$                                                ▷ Calculated distance
4:     int $split\_axis$, $count \leftarrow 0$
5:     int $toVisit$[cudaStack]
6:     $toVisit[0] \leftarrow 0$
7:     $count \leftarrow count + 1$
8:     $toVisit[count] \leftarrow cur$
9:     while $count > 0$ do
10:        $cur \leftarrow toVisit[count]$
11:        $count \leftarrow count - 1$
12:        $split\_axis \leftarrow nodes[cur].depth \mod dim$
13:        if $nodes[cur].left = -1$ then                    ▷ If current node has no children
14:            for iterate over node indices on $i$ do
15:                int $idx \leftarrow indexes[nodes[cur].indexes + i]$
16:                $d \leftarrow$ CheckDistance($query, pts[idx]$)
17:                if $d <= distance$ and $d \neq 0$ then
18:                    EditGroups($group$, $self$, $idx$)
19:                end if
20:            end for
21:        else
22:            $d \leftarrow$ CheckDistance($query, pts[nodes[cur].me]$)
23:            if $d <= distance$ and $d \neq 0$ then
24:                EditGroups($group$, $self$, $nodes[cur].me$)
25:            end if
26:            if $nodes[cur].split\_value > (query.coords[split\_axis] + distance$ then
27:                $count \leftarrow count + 1$
28:                $toVisit[count] \leftarrow nodes[cur].left$
29:            end if
30:            if $nodes[cur].split\_value < (query.coords[split\_axis] - distance$ then
31:                $count \leftarrow count + 1$
32:                $toVisit[count] \leftarrow nodes[cur].right$
33:            end if
34:        end if
35:    end while
36: end procedure
```

depth is extracted and used to determine its split axis. This is used later when determining branching directions in the tree. The node is then checked to determine whether it is a leaf node. If it is not, we check the distance between the current node and the query point. If this distance is within the query range, we invoke the `EditGroups` procedure shown in Algorithm 6 continued. Since the current node is not a leaf node, the left and right trees below it must be examined to determine whether a possible node within range exists within them. This is accomplished in the following way:

$$currentNode.split\_value > queryPoint.coordinates[split\_axis] + range \tag{4.2}$$

$$currentNode.split\_value < queryPoint.coordinates[split\_axis] - range \tag{4.3}$$

Equation 4.2 determines whether a possible node of interest exists in the left sub-tree, and Equation 4.3 in the right. Should one of these equations not be satisfied, that entire branch of the tree can be ignored. Alternatively, if the current node is a leaf node, we iterate over its indices, checking the distance between the current node and each index.

During distance calculations, should the two particle nodes in question be within range, we invoke the `EditGroups` procedure. This procedure works in a similar way to the result array editing performed in Section 4.3. That is, the results array is edited so that each particle in a specific particle group references the leader of its group. If the particle in question is the leader of the group, it references itself. This setup makes it easy for the CPU to consolidate these groups once the array has been returned, and extract viable structures from them.

On completion of this kernel, we are left with an integer array, where each element represents the particle associated with that element in the particles array, and each element points to the leader of its particle group.

4.4.4 Result structure consolidation

Once the kernel has completed, the resultant array, containing integers indicating particle groupings, is copied back to the CPU, as described in Section 4.3.

However, some additional computation is required to consolidate the data and extract viable resultant structures. This work is accomplished in the same manner as the group consolidation in Section 4.3, and described by Algorithm 4.

4.5 Summary

We implemented four separate halo finding systems as follows:

- Euclidean minimum spanning trees on a CPU
- Kd-trees on a CPU
- A brute-force approach using tiling on the GPU
- Kd-trees on a GPU

Different approaches towards solving the problem were used, together with differing hardware architectures. The systems implemented on the CPU were intended to serve as benchmarks for those implemented on the GPU. Each implementation was modified to handle a 6D phase-space halo finding simulation, and each system successfully computed resultant halo groups comparable to current commercial applications. The run-times of each application, and appropriate speed-up comparisons are presented in the following section.

Algorithm 6 Kd-tree GPU kernel continued

```
37: procedure EditGroups(int* $group$, int $self$, int $target$)
38:     int $targetCur$, $targetBack$, $selfCur$, $selfBack$
39:     $targetCur \leftarrow target$
40:     $targetBack \leftarrow group[target]$
41:     $selfCur \leftarrow curPosID$
42:     $selfBack \leftarrow group[curPosID]$
43:     while $selfCur \neq selfBack$ or $targetCur \neq targetBack$ do
44:         $targetCur \leftarrow targetBack$
45:         $targetBack \leftarrow group[target]$
46:         $selfCur \leftarrow selfBack$
47:         $selfBack \leftarrow group[selfCur]$
48:     end while
49:     if $selfBack \neq targetBack$ then
50:         if $selfBack < targetBack$ then
51:             $group[targetBack] \leftarrow selfBack$
52:         else if $selfBack > targetBack$ then
53:             $group[selfBack] \leftarrow targetBack$
54:         end if
55:     end if
56: end procedure

57: procedure CheckDistance(const Point &$a$, const Point &$b$)
58:     float $distance \leftarrow 0$
59:     for each dimension in points do
60:         float $d \leftarrow a.coords[i] - b.coords[i]$
61:         $distance \leftarrow distance + d^2$
62:     end for
63:     return squareRoot($distance$)
64: end procedure
```

Chapter 5

Testing and results

This chapter describes the results of benchmarking the halo-finding GPU implementations against their CPU counterparts, as well as GPU simulation performance against current commercial halo-finders. Benchmarks were performed on a Linux 64-bit operating system using a Geforce GTX460ti, an older entry-level device, with 1024MB GDDR memory and an Intel Core i5-2400 CPU with 8GB RAM.

We begin by discussing the data used in the applications and how it is structured. Following this, we describe the testing and timing of runs for the MST CPU and brute-force GPU implementations, and examine the results achieved. Next, we present a comparison of results for the CPU and GPU kd-tree implementations, as well as a discussion of the implementation efficiency based on differing kd-tree depths. Finally, the fastest implementation is tested against a current widely-used commercial halo finder.

5.1 Data

The data used to test our implementations is a single file in a *tipsy* binary format. Tipsy binary is an efficient storage format for particle-based data [39].

The specific data used in these comparisons consists of 256 particles, per dimension, in a bounding box of 150 megaparsecs (roughly 490 million light years). That is, a simulation with 256x256x256 data points, or particles of interest. As the run time of the applications is dependent on data-size, each was run multiple times, with a different number of particles. We sub-sampled the data-sets by selecting particles at random. This ensured that the clustering behaviour of the particles did not change. That is, the overall structure was conserved, so as not to compromise the authenticity of the results. This resulted in tests of 100 000 to 500 000 particles for the MST and brute-force implementations, as well as 1 million to 5 million particles for the CPU and GPU kd-tree implementations. Every implementation reads the position and velocity data for each particle to be used in the simulation and stores it. In the case of the 3D simulation, the velocity data is still read, but subsequently ignored. A slice of the data used is visualised in Figure 5.1.

Each implementation was run multiple times for every test and the recorded run times were averaged over the number of runs. Only the mean value was plotted, but the standard deviation for each test is noted. To ensure the correctness of our code, the output of the data from our applications was validated by comparing it to the output from the FOF commercial halo-finder application [39].

5.2 Benchmarks and results

We begin by testing and comparing the run time results for the MST CPU and brute-force GPU applications. Following this, we examine the results of the kd-tree implementations and discuss the impact of kd-tree depth on the results.

5.2.1 MST CPU and brute force GPU results

The final results for the run times of brute-force GPU and CPU MST proof-of-concept applications are depicted in Figures 5.2 and 5.3, and Table 5.1. The applications were tested in increments of 100 000 with data points ranging from 100 000 to 500 000. This range is relatively small for a halo-finding simulation, but the applications did not perform exceptionally well and the number of data points was lowered due to run time constraints.

In Figure 5.2, it is clear that for smaller data sets, the brute-force GPU implementation outperforms its MST CPU counterpart. However, as we increase the number of particles, the difference in performance is far less noticeable. This is due to the non-scalable nature of the brute-force implementation. The brute-force algorithm calculates all pair-wise interactions on the GPU, and each thread calculates interactions for one particle [30]. As the GPU has

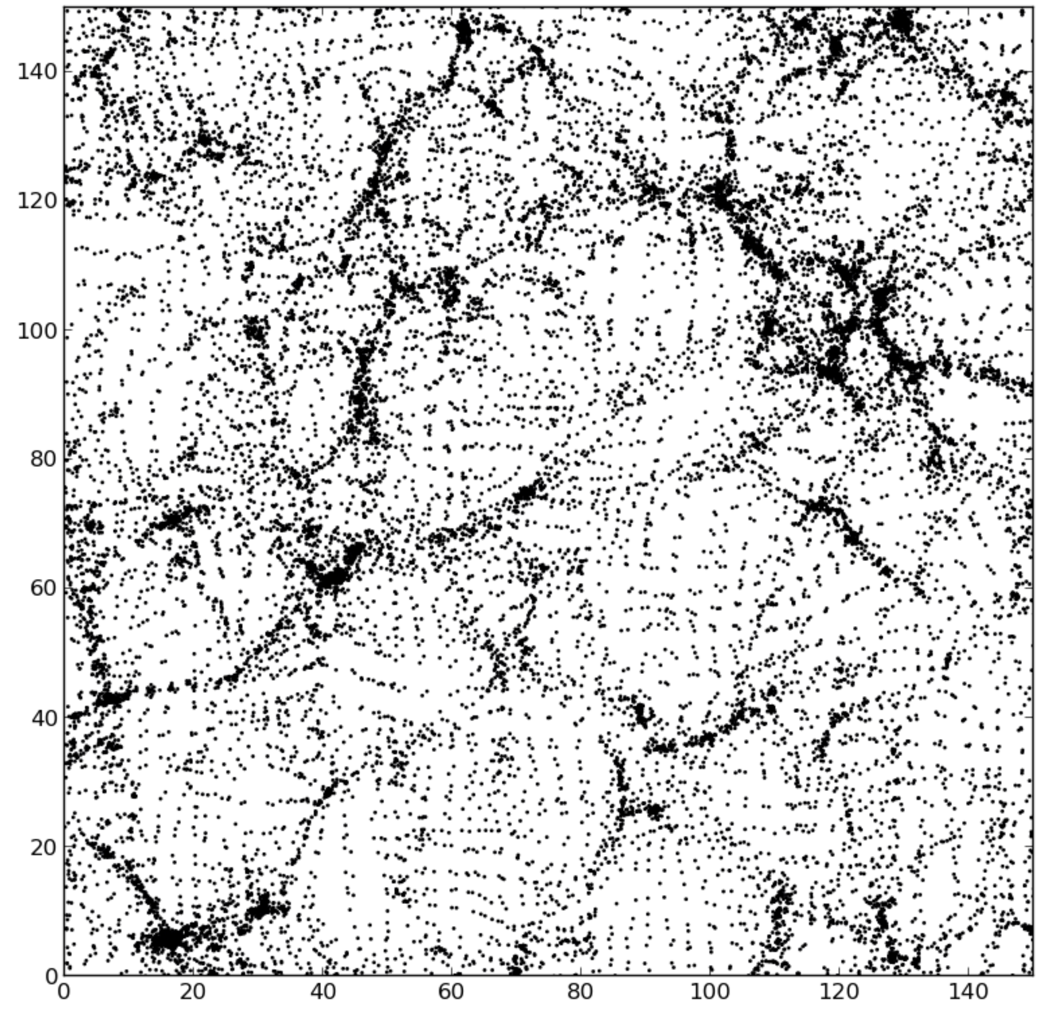

Figure 5.1: *A direct orthogonal projection of the data file representing the 3D data in 2D at a set axis depth. Each dot represents a particle in the data.*

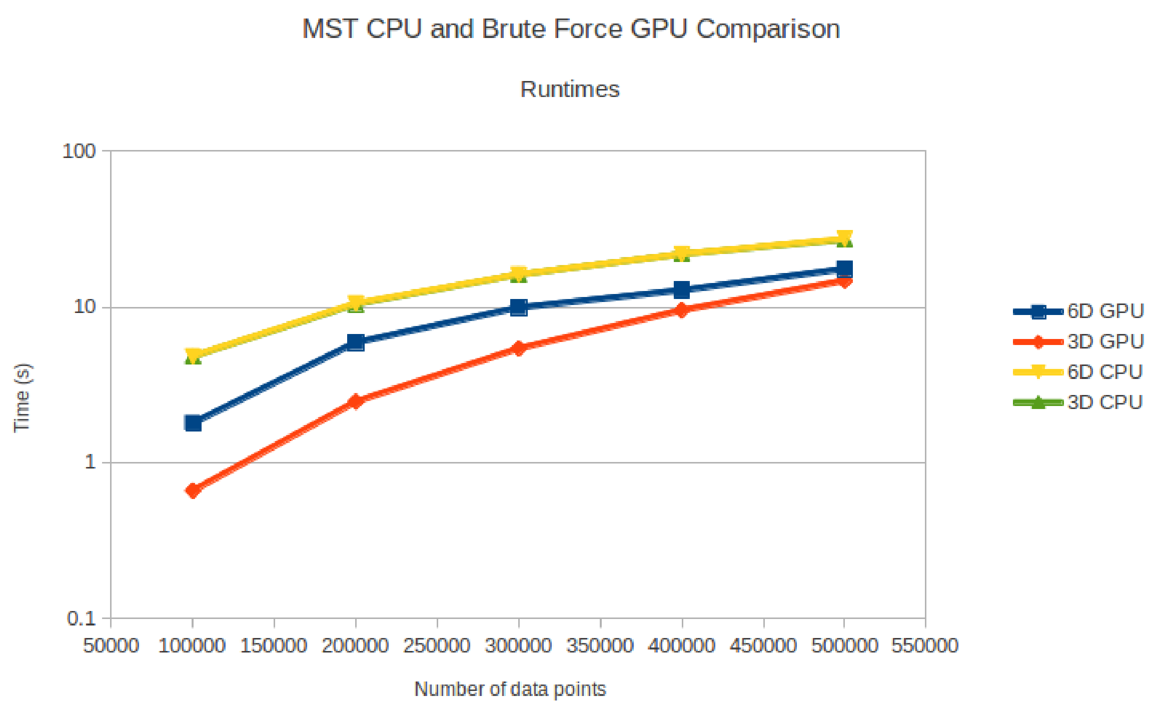

Figure 5.2: *A graph comparing the average run times for the MST CPU and brute-force GPU implementations, in both 3D and 6D, for data points ranging from 100 000 to 500 000. Both CPU applications run slower than their GPU counterparts, while the 3D GPU application runs faster than its 6D partner. This is due to the additional floating-point arithmetic required for a 6D distance calculation (Equation 3.3).*

a fixed thread limit, this approach does not scale well.

A slight difference in run time is present between the 6D and 3D GPU implementations. This is caused by the additional floating-point arithmetic required when determining whether two particles should be linked in a 6D implementation. That is, determining the result of Equation 3.3 for 6D requires floating-point division, whereas Equation 3.2 does not. Calculating floating-point division on a GPU can take up to 36 clock cycles, as opposed to 4 clock cycles for addition and multiplication. This results in significant performance hit for a 6D implementation. Furthermore, a 6D simulation requires additional velocity parameters to perform linking calculations. This additional information must be copied to the GPU alongside the particle position data, whereas a 3D implementation requires only the latter. As data transfer between GPU memory and main memory is relatively expensive, this too results in a small performance decrease.

Data points	**3D CPU (s)**	**6D CPU (s)**	**3D GPU (s)**	**6D GPU (s)**
100000	4.81 ($\sigma = 10ms$)	4.82 ($\sigma = 8ms$)	0.66 ($\sigma = 5ms$)	1.79 ($\sigma = 4ms$)
200000	10.42 ($\sigma = 14ms$)	10.52 ($\sigma = 15ms$)	2.46 ($\sigma = 7ms$)	5.9 ($\sigma = 11ms$)
300000	16.13 ($\sigma = 17ms$)	16.15 ($\sigma = 18ms$)	5.4 ($\sigma = 8ms$)	9.9 ($\sigma = 9ms$)
400000	21.81 ($\sigma = 13ms$)	21.83 ($\sigma = 11ms$)	9.51 ($\sigma = 12ms$)	12.79 ($\sigma = 18ms$)
500000	27.03 ($\sigma = 28ms$)	27.28 ($\sigma = 25ms$)	14.73 ($\sigma = 18ms$)	17.531 ($\sigma = 19ms$)

Table 5.1: *A table comparing the average run times for the MST CPU and Brute Force implementations, for both 3D and 6D, for data points ranging from 100 000 to 500 000. σ denotes the standard deviation for each test.*

Conversely, there is no clear difference between the 6D and 3D implementations for the MST CPU application, due to the CPU's ability to calculate floating-point arithmetic efficiently.

Figure 5.3 depicts the results when comparing the brute-force GPU implementation with the CPU MST application. A fairly large speed-up is noticeable for particle ranges below 300 000 when considering the 3D implementation, with the same being true to a lesser degree for the 6D implementation. Datasets greater than 300 000 do not result in a significant speed-up, due to the implementation's lack of scalability. Extrapolating the curve in the graph, we may conclude that a brute-force approach to a 3D halo-finding implementation results in good speed-ups for small simulations, but is not appropriate otherwise. Furthermore, a brute-force approach to 6D halo-finding does not result in significant performance increases, for either small or large simulations.

5.2.2 Kd-tree CPU and GPU results

The final results for the run times of the CPU and GPU kd-tree applications are now discussed. These applications were tested in increments of 1 million data points, ranging from 1 to 5 million. To begin, we discuss the effect of kd-tree depth optimisations on the application run times. Following this, the final run time results achieved by the applications are examined.

Effect of kd-tree depth on results

Due to GPU global memory limitations discussed in Chapter 4, it was necessary to limit the depth of the kd-tree in our GPU applications for large simulations. The results of such

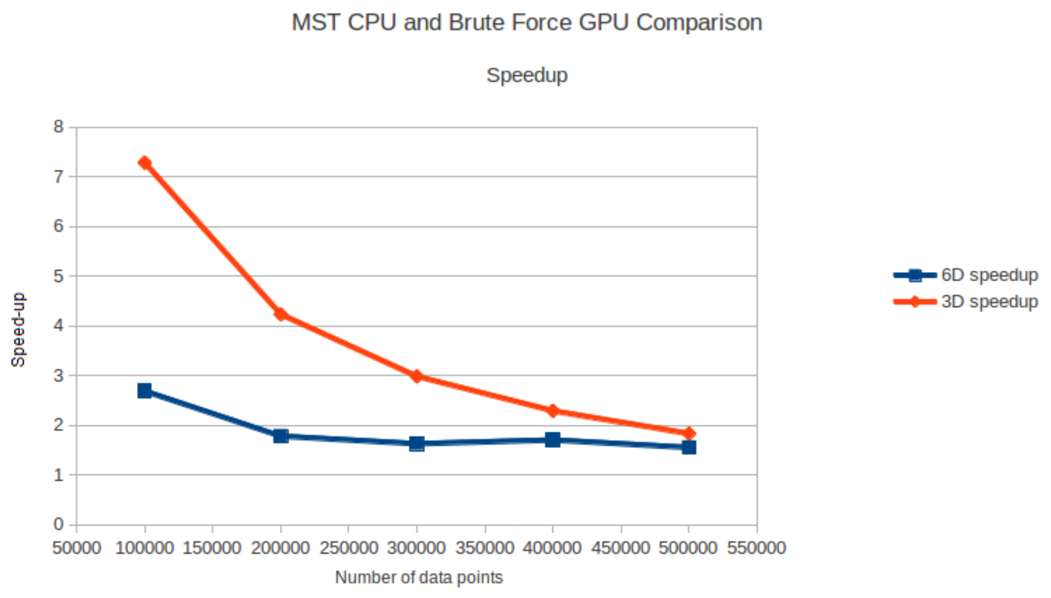

Figure 5.3: *A comparison of the speed-up of the brute force GPU implementation when compared to the MST CPU implementation, for both 3D and 6D, with data sets ranging from 100 000 to 500 000 particles. The speed-up for the 3D GPU implementation over its CPU counterpart is noticeable for smaller data sets, but quickly diminishes as particle numbers increase, implying a non-scalable solution.*

kd-tree depth limits are depicted in Figures 5.4, 5.5, 5.6 and 5.7. For these tests, we use kd-trees of two different depths: A kd-tree of depth 15 and a kd-tree with a depth that allows each data point to be incorporated into the kd-tree as a separate node. Note that in the case where the depth of the kd-tree is too small to incorporate all data points as nodes, overflowing data points are stored as lists in each leaf node. Consequently, these points must be searched iteratively, and thus it is no longer possible to discard parts of the search space.

Figure 5.4 illustrates the difference in time required to copy all data from the host (CPU) to the GPU. In other words, the time required to copy all position and velocity data, as well as each node of the kd-tree constructed on the CPU. Results for the 3D implementation timings were omitted here, as the curves are identical. As expected, the results for a kd-tree of depth 15 do not vary significantly within the 1 to 5 million data point range. This is because the amount of data to be copied increases only slightly as we increase the size of the dataset. That is, for every increment of 1 million, an additional 1 million particle positions and velocities must be copied. Since the number of nodes in the kd-tree remains constant,

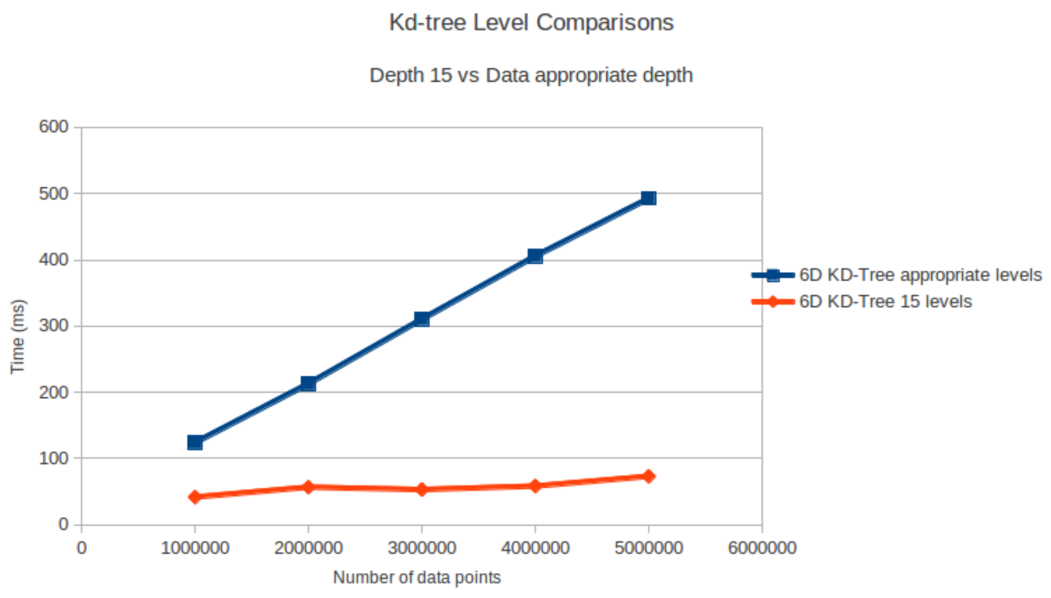

Figure 5.4: *A comparison of the creation times plus the copy time from the CPU to the GPU, for the 6D GPU kd-tree implementation, with different kd-tree depths. Appropriate levels refers to the minimum number of levels required to account for all data points. The 3D implementation timings were omitted as they are identical. When the kd-tree remains at a specific depth, the timings do not vary, as the size of the kd-tree remains static. Alternately, for a varying kd-tree depth, where the depth is determined based on the number of data points in the simulation, the kd-tree is larger for greater numbers of particles. As such, the time required to transfer the data increases.*

the size increases by 1 million integers for dataset increment of 1 million. These additional points are then stored in the tree's leaves.

On examination of the results for a kd-tree of appropriate depth, it is clear that the amount of time required to copy the required data from the host to the GPU increases in a linear fashion. This is because the number of nodes in the kd-tree correlates directly with the size of the dataset. Since the number of data points increases in linear 1 million point increments, a similar linear increase in the required time to copy from host to GPU is expected. Since the time required to copy the particle positions and velocities is not significant, it is clear that this linear increase in time to copy is caused by the additional kd-tree nodes only.

Figure 5.5 depicts the difference in time required to execute the GPU kernel for trees of varying depths. The results of the 3D implementation show that for a small number of data

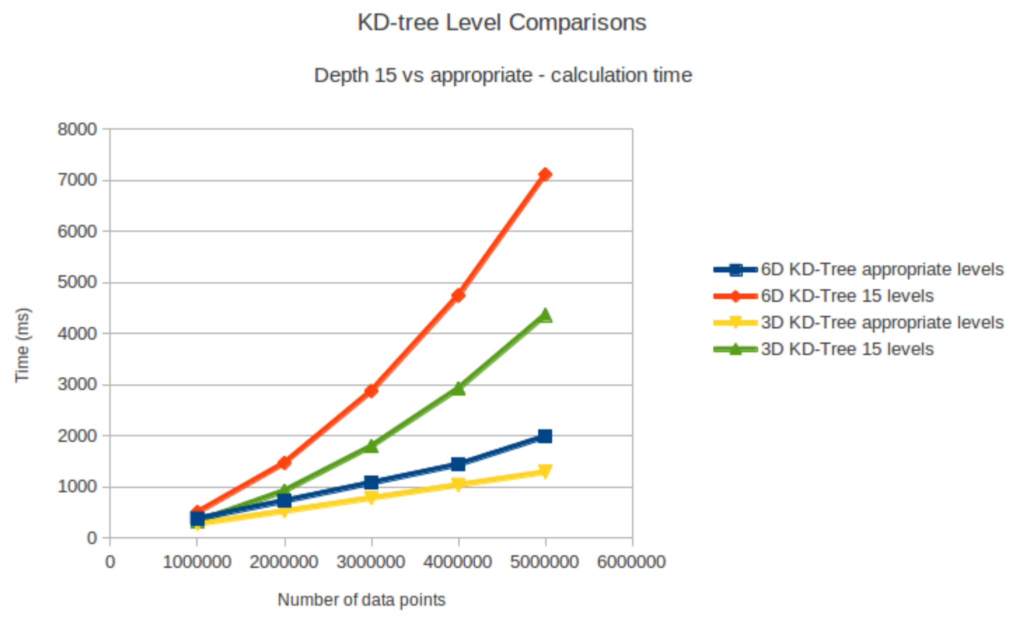

Figure 5.5: *A comparison of the calculation times (excluding tree creation and data transfer), for the GPU kd-tree implementation, with different kd-tree depths. Appropriate levels refers to the minimum number of levels required to account for all data points. Kd-trees with an appropriate depth allow greater potential to remove branches of the tree and hence narrow the search space, decreasing computation time.*

points, the difference between the kernel calculation time for a kd-tree of depth 15, and one of appropriate depth is small. However, as we increase the number of data points, it is clear that the kd-tree with an appropriate depth easily outperforms that of depth 15. As discussed in Section 4, when the depth of a kd-tree is limited, any additional data points that would normally become children of the leaf nodes are added to the leaf nodes as indices. These are then processed sequentially, in a similar fashion to the brute-force GPU implementation. Thus, a kd-tree with an appropriate depth is better able to effectively prune the search space, as the additional nodes may be processed normally. That is, with additional data nodes in the kd-tree, many more branches of the tree may be discarded, as this allows for more left-right branching decisions.

On examination of the 6D implementation, one can see that the results are very much the same, with the exceptions that the run times are slightly higher than the 3D implementations. In addition, the difference in speed-up between the depth 15 and appropriate depth kd-tree implementations varies far more. The reasons for this are discussed later in this section.

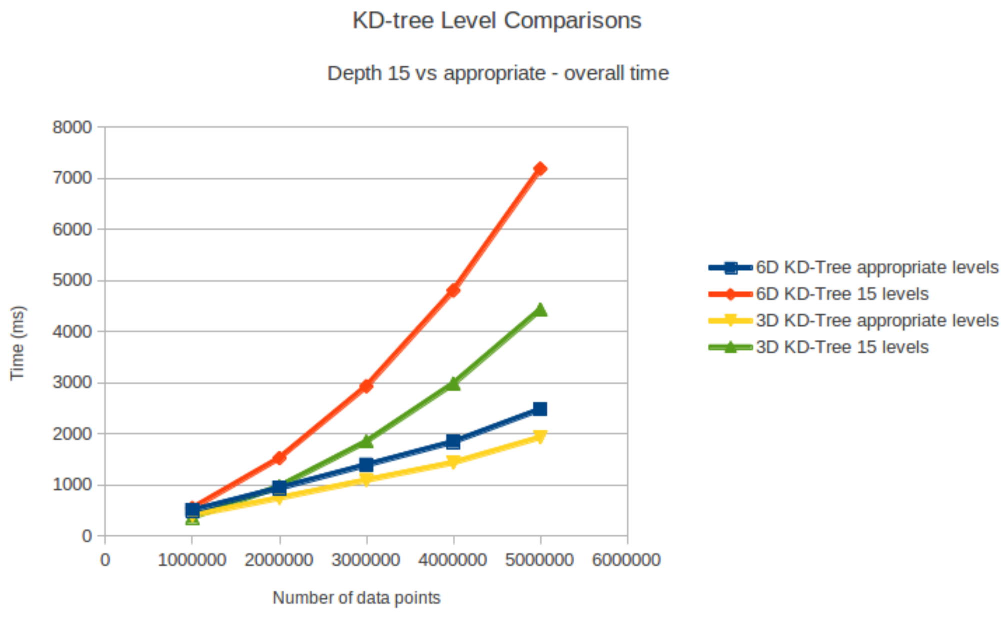

Figure 5.6: *A comparison of the overall run times including data transfer for the GPU kd-tree implementation, with different kd-tree depths. Appropriate levels refers to the minimum number of levels required to account for all data points. Despite the additional creation and data transfer time required for a kd-tree of appropriate depth, the additional search space optimisation that this affords far outperforms a kd-tree with a limited depth.*

Figure 5.6 shows the overall run time for the kd-tree GPU applications with kd-trees of limited and appropriate depths. On further examination, we notice that the graph looks extremely similar to that in Figure 5.5, suggesting that the additional run time required to copy a larger kd-tree from the host to the GPU is not significant when compared to the added search space optimisation gained due to additional kd-tree nodes.

Furthermore, there is a difference in curve shape for the kd-trees of depth 15 and appropriate levels in both the 3D and 6D implementations. As the dataset is increased, the curves for the depth 15 kd-trees become steeper, as opposed to those of the appropriate depth kd-trees, which do so at a significantly slower rate. This suggests that kd-trees of appropriate depth are more scalable than those with a fixed lower depth, resulting in greater speed-ups for large simulations. This is illustrated in Figure 5.7.

Examining the graph, a significant speed-up is visible when the kd-tree depth satisfies the number of data points in the simulation. That is, when the kd-tree depth is high enough

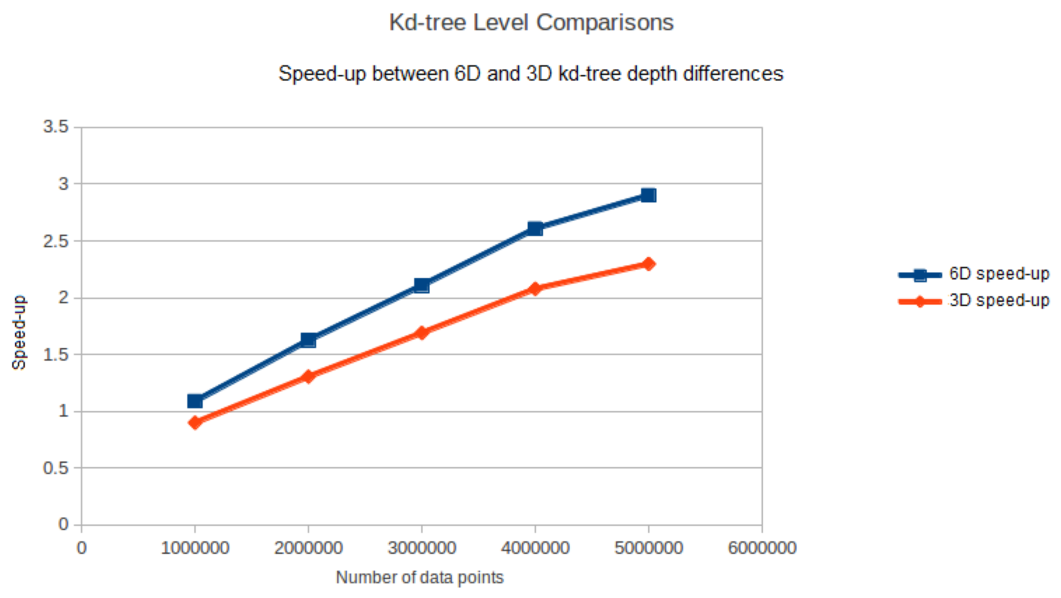

Figure 5.7: *A comparison of the speed-up for the GPU kd-tree implementation, when using different kd-tree depths. Appropriate levels refers to the minimum number of levels required to account for all data points.*

that there are no data points bucketed in the leaf nodes. It is clear that the 6D kd-tree implementation benefits from a greater speed-up than the 3D implementation. This is due to the additional floating-point arithmetic required for a 6D linking criterion calculation. As discussed earlier, incorporating a greater number of data points in the kd-tree increases the potential to eliminate portions of the search space. As a result, fewer costly linking calculations are required, greatly reducing the overall run time, and thus significantly increasing the speed-up.

Moreover, we notice a slightly lower speed-up when increasing from 4 million to 5 million particles. This is due to the hardware used as it limits the kd-tree depth to 22. The GPU used can only store a little over 4 million kd-tree nodes. Since the maximum number of kd-tree nodes in a 22 level kd-tree is 4 194 303, this only impacts the 5 million data point test, as this would require a depth of 23. As such, 805 697 data points are added to the leaves of the kd-tree. These are then processed sequentially in each thread. This slight decrease in speed-up for the 5 million data point case clearly demonstrates the negative effect even one fewer kd-tree level can have on calculation speed.

Finally, the graph curves suggest that completely saturating the kd-tree in either implementation produces a significant speed-up, increasing as we add data points to the simulation. Such a drastic difference in performance indicates that, should we seek maximum speed-up for large simulations, additional GPU devices should be added.

Benchmark results

Figures 5.8 and 5.9, and Table 5.2 describe the final results achieved by the CPU and GPU kd-tree halo-finder implementations, and the speed-up delivered.

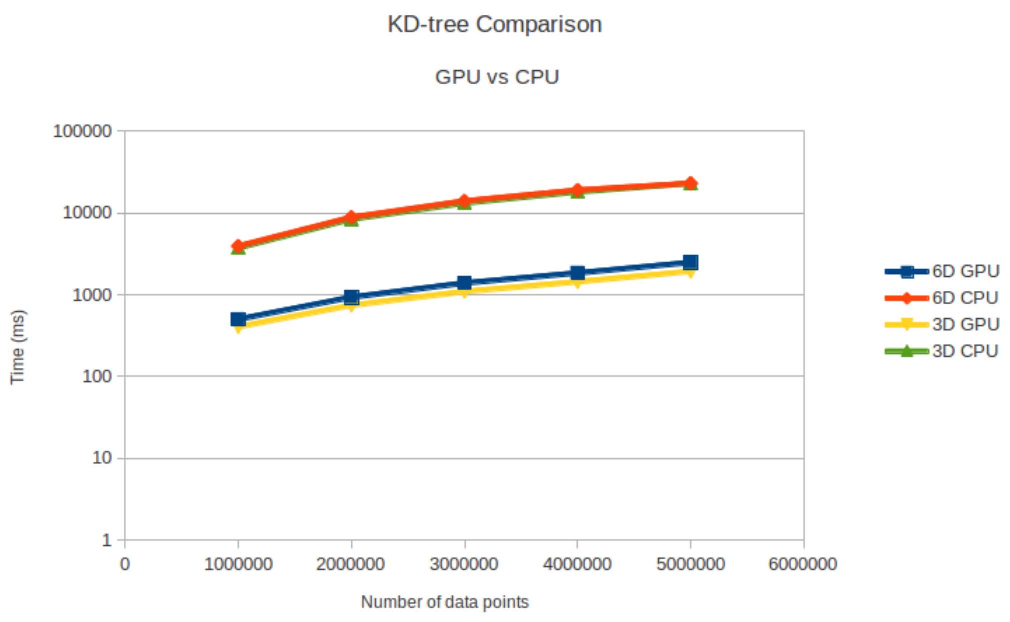

Figure 5.8: *A comparison of the average run times for GPU and CPU kd-tree implementations, for both 3D and 6D phase-space, for data points ranging from 1 000 000 to 5 000 000. The GPU applications clearly perform better than their CPU counterparts.*

In Figure 5.8, we examine the run times produced by the CPU and GPU kd-tree applications, for both 3D and 6D phase-space. To begin, it is clear that there is no significant difference in run time between the 3D and 6D implementations for the CPU application. If we consider that the only difference between these applications is the additional arithmetic operation required to calculate the linking length, and that the CPU is particularly suited to sequential instructions as well as floating-point arithmetic, this result is not unexpected. Therefore, when extrapolating this trend, we may deduce that the difference in run time for both

Data points	3D CPU (s)	6D CPU (s)	3D GPU (s)	6D GPU (s)
1000000	3.717 ($\sigma = 10ms$)	3.925 ($\sigma = 11ms$)	0.405 ($\sigma = 1ms$)	0.5 ($\sigma = 1ms$)
2000000	8.373 ($\sigma = 18ms$)	8.844 ($\sigma = 15ms$)	0.742 ($\sigma = 2ms$)	0.937 ($\sigma = 2ms$)
3000000	13.216 ($\sigma = 13ms$)	14.090 ($\sigma = 12ms$)	1.093 ($\sigma = 4ms$)	1.390 ($\sigma = 3ms$)
4000000	18.072 ($\sigma = 18ms$)	19.922 ($\sigma = 19ms$)	1.435 ($\sigma = 3ms$)	1.842 ($\sigma = 5ms$)
5000000	23.051 ($\sigma = 25ms$)	23.189 ($\sigma = 23ms$)	1.929 ($\sigma = 6ms$)	2.477 ($\sigma = 8ms$)

Table 5.2: *A table comparing the average run times for the GPU and CPU kd-tree implementations, for both 3D and 6D, for data points ranging from 1 000 000 to 5 000 000. σ denotes the standard deviation for each test.*

versions of the kd-tree CPU implementation is not significantly different as we increase the size of the datasets.

The same conclusion, however, does not hold for the GPU implementation. Again, this is consistent with previous results discussed in this chapter. Specifically, since the GPU is not particularly suited to floating point arithmetic, we expect a 6D halo-finding simulation to be out-performed by a 3D version, as it requires additional calculations when computing 6D linking length criteria.

The run time results achieved by our kd-tree CPU application were slightly higher than expected. This may be attributed to running the ANN fixed-radius search twice, as discussed in Chapter 4. Modifying the ANN fixed-radius procedure to dynamically add particles as they are discovered in range, and removing the need for a k particles parameter, may solve this problem. Considering that the fixed-radius search accounts for the bulk of computation time, running this code once, instead of twice, should halve the run time results shown in Figure 5.8.

Despite halving the run times of our CPU kd-tree application, it is clear that this is outperformed by a significant margin by our GPU kd-tree implementation.

Figure 5.9 builds on the results depicted in Figure 5.8, presenting the speed-up achieved for the kd-tree implementations. To begin, it is clear that the speed-up deadlines at 5 million data points. This is due to the kd-tree depth being limited at this point. As discussed in Section 5.2.2, limiting the kd-tree depth significantly affects the speed-up achieved. Despite this, a significant speed-up for 5 million data points is still recorded. Were we to test the application for increasingly larger simulations whilst maintaining a constant kd-tree depth, however, the speed-up would decrease accordingly.

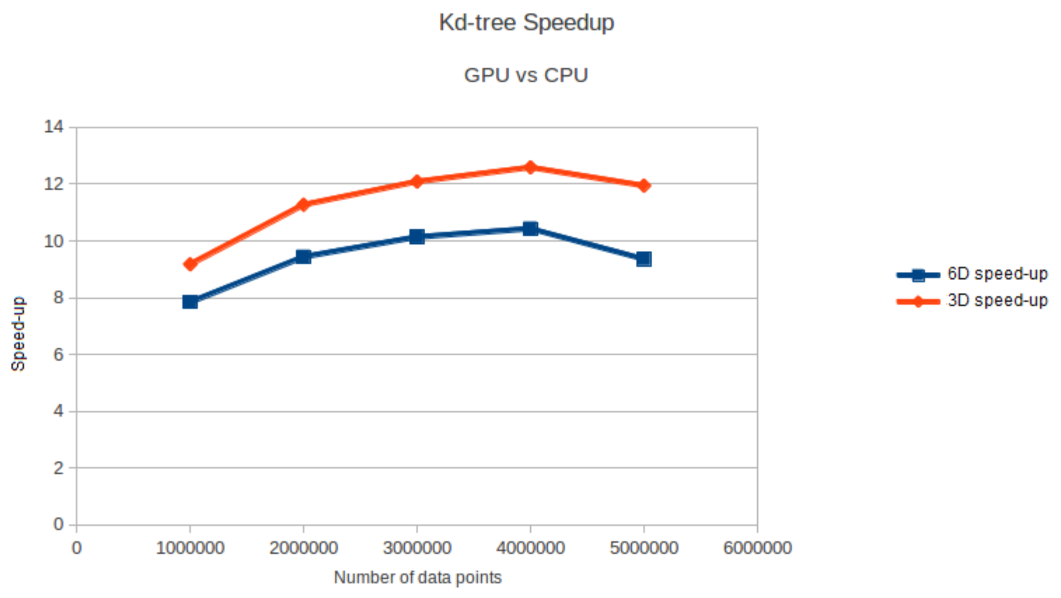

Figure 5.9: *A comparison of the speed-up of the GPU and CPU kd-tree implementations, for both 3D and 6D phase-space, for datasets ranging from 1 to 5 million points. Running these applications on a GPU results in a significant speed-up over the CPU version. At 5 million data points, the speed-up decreases due to memory limitations on the hardware used in these tests. This is due to the limited depth of the kd-tree*

.

Finally, the correlation between the number of data points and the speed-up achieved should be noted. That is, the speed-up achieved is proportional to the number of data points. This excludes the 5 million point dataset, due to its memory limitations. This can be attributed to reducing the search space by discarding branches of the kd-tree. That is, as the size of the dataset increases, a larger portion of kd-tree branches can be discarded. As we increase the size of the dataset, however, although the overall speed-up increases, the amount it increases by becomes less. Although a larger simulation presents additional potential reducing the search space, this reaches an artificial ceiling due to hardware limitations.

5.2.3 Commercial halo-finder results

We tested our 3D GPU kd-tree application against a widely used commercial halo-finder called FOF, produced by the University of Washington N-Body Shop [39]. FOF is a CPU halo-finder that uses the friends-of-friends algorithm with kd-trees to effectively discover halos and particle groups. The application is a good fit for a comparison, as similar methods are used, and the code is thoroughly optimized. Figures 5.10 and 5.11 describe the run time comparison for the GPU and FOF programs, and the overall speed-up achieved between them, respectively.

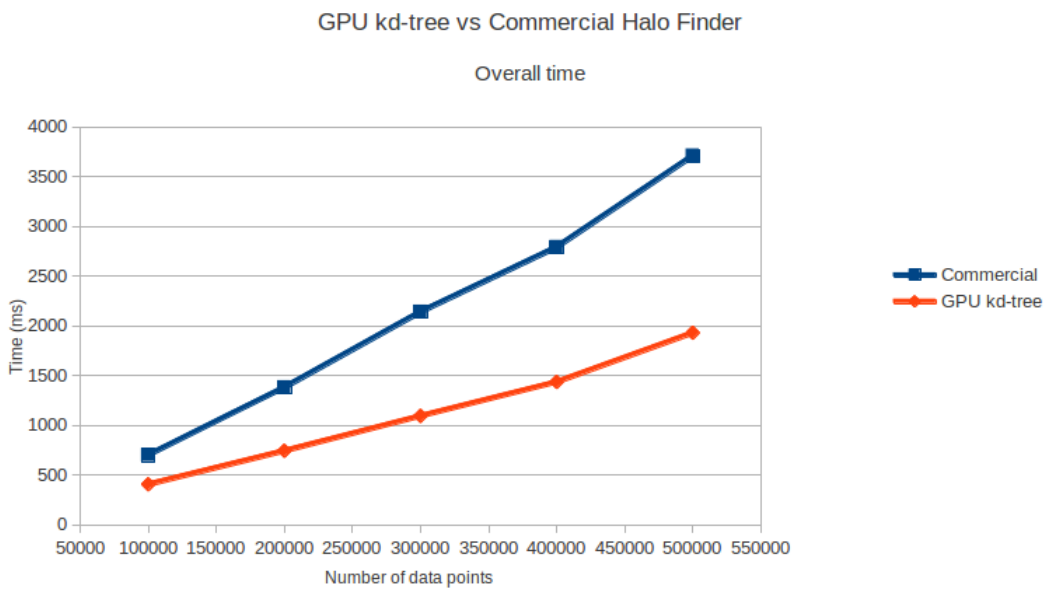

Figure 5.10: *This graph compares the average run times of the GPU implementation and those of the FoF commercial halo finder. This is for 3D simulations, with data points ranging from 1 to 5 million. The GPU application outperforms the CPU halo-finder.*

The run time results of our GPU implementation and that of the FOF halo-finder are shown in Figure 5.10. While our GPU kd-tree implementation did outperform FOF, the margin was less than expected.

On closer analysis, however, we notice that as the size of the dataset increases, the discrepancy between run times for the applications grows. As such, we may extrapolate that for larger simulations, we should gain better performance. That is, our GPU solution seems to scale better.

As with previous tests, speed-up for the GPU kd-tree implementation declines at 5 million data points. As discussed, this result is caused by our hardware's limited memory capacity, and the inability to fully represent the tree in memory. Therefore, for our solution to continue to scale linearly, it is necessary that additional hardware be used. Although potentially costly, this would be necessary only for larger simulations.

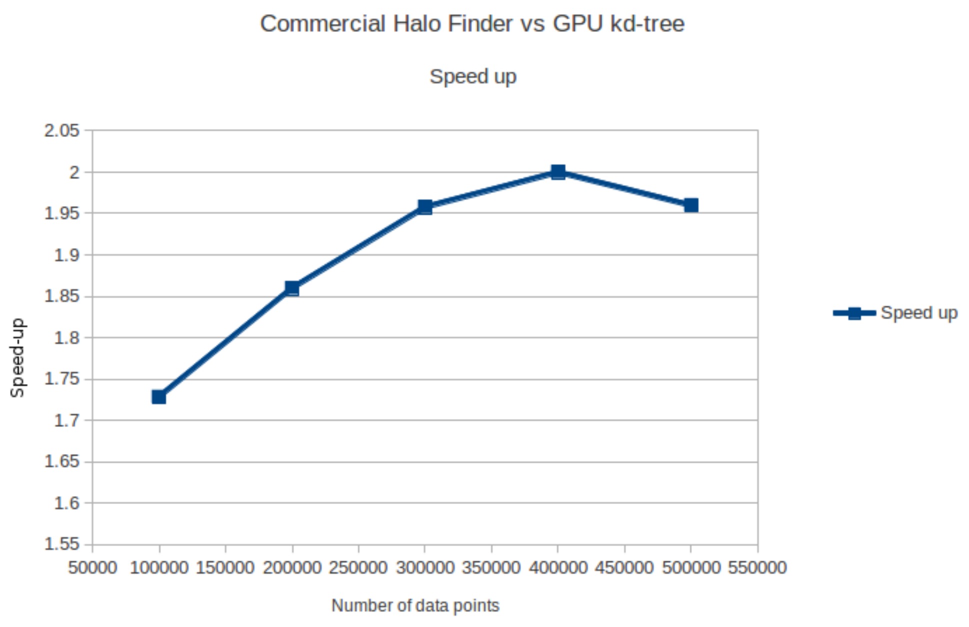

Figure 5.11: *The graph compares the speed-up of the GPU implementations and those of the FoF commercial halo finder. This is for 3D simulations, with data points ranging from 1 to 5 million. As discussed, the drop in speed-up at the 5 million data point is due to memory limitations in the hardware. These results show potential for future research into GPU halo-finders.*

Figure 5.11 shows the speed-up achieved by the GPU kd-tree implementation when compared to the FOF application. As previously mentioned, the speed-up increases with the number of data points, but slows as the GPU memory limit is reached.

Ultimately, the speed-up achieved was not as great as we had hoped. This can be attributed to a number of factors. To begin with, our GPU kd-tree implementation was not fully optimised, and the hardware used was old. Newer hardware is far more powerful, and would significantly improve performance. Additional modifications to the code to ensure optimal exploitation of the GPU architecture were not implemented. That is, lower latency device memory, such as shared and texture memory, maximum instruction- and data-level parallelism were not fully utilised. In addition, profiling and load balancing could also have been performed. In contrast, the FOF application we tested against is extremely well optimised commercial software. As such, we consider the GPU approach to halo-finding using kd-trees to hold great potential. Our research suggests that with a fair degree of optimisation, a GPU kd-tree halo-finder would compute data more efficiently than current CPU-based halo-finders.

5.2.4 Summary

Four applications were compared using datasets ranging from 1 to 5 million points. This compared the speed of computation achieved on a GPU as opposed to CPU. Following this, we measured our most successful implementation against a widely used commercial halo-finder, FOF. The following results were observed:

- *The MST CPU and brute-force GPU applications were slower, non-scalable solutions.* The GPU implementation, however, still exhibited a speed-up of 7x for 3D and 3x for 6D, over the MST CPU application, for smaller simulations. For larger simulations, the speed-ups converged to 2x for both 3D and 6D, indicating a limited degree of scalability.
- The CPU and GPU kd-tree implementations were shown to be reasonably scalable. *The speed-up achieved by the GPU kd-tree implementation is directly proportional to the size of the dataset, provided sufficient memory is available.* When memory becomes constrained, the speed-up levels off due to the inability to correctly represent the tree and thus fully optimise search space traversal.
- An investigation was conducted into the affect of saturating the kd-tree as opposed to limiting its depth according to memory limitations. Fully saturating the GPU kd-tree resulted in a 1-2x speed-up for smaller simulations, and 2-3x speed-up for slightly larger simulations. *Speed-ups thus improved linearly, implying a directly proportionate relationship between saturating the kd-tree and the number of particles in the dataset.*
- The GPU kd-tree application was compared with a current commercial halo-finder which uses similar algorithms. *The GPU application achieved a 2x speed-up on average over the commercial CPU halo-finder.* The graph suggests that speed-up increases with the size of the dataset. It should be noted that the GPU implementation was not fully optimised.

The above results suggest that there is great potential for improvement in run times of existing halo-finders through the use of GPU hardware.

Chapter 6

Conclusion

We proposed an investigation into the potential for leveraging the massively-parallel processing capabilities of entry-level graphics hardware in order to quickly perform halo-finding calculations. This was explored as a feasible alternative to current single- and multi-threaded solutions for large datasets.

As a proof-of-concept exercise, two applications were created. The first made use of Delaunay triangulation and a minimum spanning tree and was implemented on the CPU. This program served as a benchmark for a brute force model GPU approach to the problem. These implementations were intended to encourage a more complete understanding of halo-finding in both a Computer Science and Astronomy context.

The performance results, as expected for proof-of-concept exercises, were poor. Our brute force GPU application outperformed its MST CPU counterpart in both 3D, and, to a lesser extent, 6D, for relatively small simulations. It did not scale well, however, due to the brute-force nature of the solution. Furthermore, the poor performance of the CPU implementation can be attributed to the extensive pre-processing required before the friends-of-friends algorithm could be applied. That is, the formation of a seed graph using Delaunay triangulation, and the reduction of this graph to a minimum spanning tree required considerable calculation.

Following the initial implementations, two additional applications were designed. A CPU program making use of kd-trees and the Approximate Nearest Neighbours library was implemented as a benchmark for a new GPU application, also using kd-trees. These implementations successfully outperformed their proof-of-concept counterparts, despite minimal optimisation. As such, there is still considerable room for improvement when considering our kd-tree GPU implementation, particularly through shared memory optimisation and instruction-level parallelism.

The kd-tree GPU implementation significantly outperformed its kd-tree CPU counterpart by up to 12x for 3D simulations, and 10x for 6D simulations. The graph trend for these applications suggests speed-up is directly proportional to the number of data points. This in turn implies this application is highly scalable. During testing, we noticed discrepancies in expected running time results when the GPU kd-tree was limited due to GPU memory limitations. This was explored further, and we found that saturating the kd-tree on the GPU resulted in an additional 2.5x speed-up for 4 million data points, with a graph trend suggesting increasing speed-ups as data points increased.

Finally, we tested our kd-tree GPU application against the FOF commercial CPU halo finder. A speed-up of 2x was achieved for 4 million particles and results suggest further speed-ups for larger simulations when using hardware that will hold a complete kd-tree with no stacking of leaf nodes. Although not a large improvement, our kd-tree GPU halo-finder was not fully optimised, whereas the FOF commercial halo-finder is extremely efficient.

This research demonstrates that there is considerable potential for an optimised GPU-based halo-finder to greatly accelerate the discovery of galactic structures.

6.1 Future work

Examining the results achieved in this research, it is clear that there exists much potential for a commercial halo-finder leveraging the massively-parallel processing capability of graphics hardware. We consider the following possible areas for future research:

- Due to the memory limitations of our current GPU implementation, it would be a natural addition to extend the application to run on multiple graphics cards, increasing parallelism and memory capacity. In addition, it may be possible to decrease the memory space required for a kd-node on the device by optimising the kd-node structure.
- In our current GPU implementation, the kd-tree is built on the CPU. Zhou et al. [44] describes the construction of a kd-tree on the GPU device. A modified version incorporated into our GPU application could decrease kd-tree construction time and

eliminate the need to copy the kd-tree from host to device, resulting in a lower transfer overhead.

- Other than the obvious implementation extensions mentioned above, our application could benefit greatly from code optimisations such as the employment of shared memory, streamlined memory accesses and greater instruction- and data-level parallelism.
- The tests used a low end commodity graphics card with a limited global memory capacity. A GPU implementation with the above modifications and optimisations would perform significantly better on a modern graphics card.
- It may be appropriate to create a GPU halo-finder based on minimum spanning trees instead of kd-trees. Vineet et al. [41] presents an efficient method to compute MSTs using a GPU. Comparing this implementation with an optimised version of our current GPU application may yield further insights into how best to move forward.

Chapter 7

Appendix

7.1 CUDA Kernel Code

```
/*d_p is the array of particles, parent is the result
   array that will be passed back, and n is the number
   of particles*/
__global__ void 3DGPU(float4 *d_p, int *parent, int n)
   {
     __shared__ float4 shPosition[512];
     float4 myPosition;
     int i, tile, j, target;
     bool distance;
     int self = (blockIdx.x * blockDim.x + threadIdx.x);

     myPosition = d_p[self];

     for (i = 0, tile = 0; i < n; i += 512, tile++) {
         int idx1 = tile * blockDim.x + threadIdx.x;
         shPosition[threadIdx.x] = d_p[idx1];
         __syncthreads();
         for (j = 0; j < blockDim.x; j++) {
                distance = checkDistance3D(myPosition,
                   shPosition[j]);
                if (distance) {
                    target = i + j;
                    int targetCur, targetBack;
                    int selfCur, selfBack;
                    targetCur = target;
                    targetBack = parent[target];
                    selfCur = self;
                    selfBack = parent[self];
                    while (selfCur != selfBack || targetCur
                        != targetBack) {
                        targetCur = targetBack;
                        targetBack = parent[targetCur];
                        selfCur = selfBack;
                        selfBack = parent[selfCur];
                    }
                    if (selfBack != targetBack) {
                        if (selfBack < targetBack) {
                            parent[targetBack] = selfBack;
                        } else if (selfBack > targetBack) {
                            parent[selfBack] = targetBack;
                        }
                    }
                }
            }
            __syncthreads();
        }
    }

    /*pos1 and pos2 are particle data for two particles*/
    __device__ bool checkDistance3D(float4 pos1, float4
       pos2) {
```

```
    float distance = (pos1.x - pos2.x)*(pos1.x - pos2.x
       ) + (pos1.y - pos2.y)*(pos1.y - pos2.y) + (pos1.z
        - pos2.z)*(pos1.z - pos2.z);
    distance = sqrt(distance);
    if (distance < 200 && distance != 0)
        return true;
    else
        return false;
}

*/d_p is an array of particle positions, d_v is an
   array of particle velocities, parent is the result
   array that will be returned, n is the number of
   particles*/
__global__ void 6DGPU(float3 *d_p, float3 *d_v, int *
   parent, int n) {
    __shared__ float3 shPosition[256];
    __shared__ float3 shVelocity[256];
    float3 myPosition;
    float3 myVelocity;
    int i, tile, j, target;
    bool distance;
    int self = (blockIdx.x * blockDim.x + threadIdx.x);

    myPosition = d_p[self];
    myVelocity = d_v[self];

    for (i = 0, tile = 0; i < n; i += 256, tile++) {
        int idx1 = tile * blockDim.x + threadIdx.x;
        shPosition[threadIdx.x] = d_p[idx1];
        shVelocity[threadIdx.x] = d_v[idx1];
        __syncthreads();
        for (j = 0; j < blockDim.x; j++) {
            distance = checkDistance6D(myPosition,
               shPosition[j], myVelocity,
                    shVelocity[j]);
            if (distance) {
                target = i + j;
                int targetCur, targetBack;
                int selfCur, selfBack;
                targetCur = target;
                targetBack = parent[target];
                selfCur = self;
                selfBack = parent[self];
                while (selfCur != selfBack || targetCur
                    != targetBack) {
                    targetCur = targetBack;
                    targetBack = parent[targetCur];
                    selfCur = selfBack;
                    selfBack = parent[selfCur];
                }
                if (selfBack != targetBack) {
                    if (selfBack < targetBack) {
                        parent[targetBack] = selfBack;
                    } else if (selfBack > targetBack) {
                        parent[selfBack] = targetBack;
                    }
                }
            }
            __syncthreads();

        }
        __syncthreads();
    }
}
```

```
/*pos1 and pos2 contain position data for two particles
   , vel1 and vel2 contain velocity data for two
   particles*/
__device__ bool checkDistance6D(float3 pos1, float3
   pos2, float3 vel1, float3 vel2) {
    float distance =
            pow(((pos1.x - pos2.x) + (pos1.y - pos2.y)
               + (pos1.z - pos2.z))
                    * ((pos1.x - pos2.x) + (pos1.y -
                       pos2.y) + (pos1.z - pos2.z))
                    / 200, 2)
                    + pow(((vel1.x - vel2.x) + (vel1.y
                       - vel2.y)
                            + (vel1.z - vel2.z))
                            * ((vel1.x - vel2.x) + (
                               vel1.y - vel2.y)
                                    + (vel1.z - vel2.z)
                                       ) / 200, 2);
    if (distance < 1 && distance != 0)
        return true;
    else
        return false;
}

/*nodes is an array of kd-tree nodes, indexes are the
   indexes of the particles, pts is an array of the
   particles, cur denotes the current particle, query
   denotes the current particle being queried, self
   denotes the particle being searched from, parents is
   the return result array, and link is the linking
   length*/
__device__ void 3DKDTree(const CUDA_KDNode *nodes,
   const int *indexes, const Point *pts, int cur, const
   Point &query, int self, int *parents, float link) {
    float distance = link;
    float d;
    int split_axis;
    int count = 0;
    int to_visit[CUDA_STACK];
    to_visit[0] = 0;
    count = count + 1;
    to_visit[count] = cur;
    while (count > 0) {
        cur = to_visit[count];
        count = count - 1;
        split_axis = nodes[cur].level % KDTREE_DIM;
        if (nodes[cur].left == -1) {
            for (int i = 0; i < nodes[cur].num_indexes;
                i++) {
                int idx = indexes[nodes[cur].indexes +
                   i];
                d = Distance3DKDTree(query, pts[idx]);
                if (d <= distance && d != 0) {
                    performCalc3DKDTree(parents, self,
                       idx);
                }
            }
        } else {
            d = Distance(query, pts[nodes[cur].me]);
            if (d <= distance && d != 0) {
                performCalc3DKDTree(parents, self,
                   nodes[cur].me);
            }
            if (nodes[cur].split_value > (query.coords[
               split_axis] - distance)) {
                count = count + 1;
                to_visit[count] = nodes[cur].left;
            }
```

```
                if (nodes[cur].split_value < (query.coords[
                   split_axis] + distance)) {
                    count = count + 1;
                    to_visit[count] = nodes[cur].right;
                }
            }

        }
    }

    __device__ void performCalc3DKDTree(int *parent, int
       self, int target) {
        int targetCur, targetBack;
        int selfCur, selfBack;
        targetCur = target;
        targetBack = parent[target];
        selfCur = self;
        selfBack = parent[self];
        while (selfCur != selfBack || targetCur !=
           targetBack) {
            targetCur = targetBack;
            targetBack = parent[targetCur];
            selfCur = selfBack;
            selfBack = parent[selfCur];
        }
        if (selfBack != targetBack) {
            if (selfBack < targetBack) {
                parent[targetBack] = selfBack;
            } else if (selfBack > targetBack) {
                parent[selfBack] = targetBack;
            }
        }
    }
```

```
__device__ float Distance3DKDTree(const Point &a, const
    Point &b) {
     float dist = 0;
     for (int i = 0; i < KDTREE_DIM; i++) {
          float d = a.coords[i] - b.coords[i];
          dist += d * d;
     }
     return sqrt(dist);
}


/*nodes is an array of kd-tree nodes, indexes are the
   indexes of the particles, pts is an array of the
   particles, cur denotes the current particle, query
   denotes the current particle being queried, self
   denotes the particle being searched from, parents is
   the return result array, velocity is the velocity
   data of the query particle, vel is the velocity data
   of the cur particle, and link is the linking length
   is position space, and velLink is the linking length
   in velocity space*/
__device__ void 6DKDTree(const CUDA_KDNode *nodes,
   const int *indexes, const Point *pts, int cur, const
   Point &query, int self, int *parents, Point *velocity
   , const Point &vel, float link, float velLink) {
     float distance = 1;
     float d;
     int split_axis;
     int count = 0;
     int to_visit[CUDA_STACK];
     to_visit[0] = 0;
     count = count + 1;
     to_visit[count] = cur;
```

```
        while (count > 0) {
            cur = to_visit[count];
            count = count - 1;
            split_axis = nodes[cur].level % KDTREE_DIM;
            if (nodes[cur].left == -1) {
                for (int i = 0; i < nodes[cur].num_indexes;
                     i++) {
                    int idx = indexes[nodes[cur].indexes +
                       i];
                    d = Distance6DKDTree(query, vel, pts[
                       idx], velocity[idx], link, velLink);
                    if (d < distance && d != 0) {
                        performCalc6DKDTree(parents, self,
                           idx);
                    }
                }
            } else {
                d = Distance(query, vel, pts[nodes[cur].me
                   ], velocity[nodes[cur].me], link, velLink
                   );
                if (d < distance && d != 0) {
                    performCalc6DKDTree(parents, self,
                       nodes[cur].me);
                }
                if (nodes[cur].split_value > (query.coords[
                   split_axis] - link)) {
                    count = count + 1;
                    to_visit[count] = nodes[cur].left;
                }

                if (nodes[cur].split_value < (query.coords[
                   split_axis] + link)) {
                    count = count + 1;
                    to_visit[count] = nodes[cur].right;
                }
            }

        }
    }

    __device__ void performCalc6DKDTree(int *parent, int
       self, int target) {
        int targetCur, targetBack;
        int selfCur, selfBack;
        targetCur = target;
        targetBack = parent[target];
        selfCur = self;
        selfBack = parent[self];
        while (selfCur != selfBack || targetCur !=
           targetBack) {
            targetCur = targetBack;
            targetBack = parent[targetCur];
            selfCur = selfBack;
            selfBack = parent[selfCur];
        }
        if (selfBack != targetBack) {
            if (selfBack < targetBack) {
                parent[targetBack] = selfBack;
            } else if (selfBack > targetBack) {
                parent[selfBack] = targetBack;
            }
        }
    }

    __device__ float Distance6DKDTree(const Point &pos1,
       const Point &vel1, const Point &pos2, const Point &
       vel2, float link, float velLink) {
        float pos = (pos1.coords[0] - pos2.coords[0]) + (
           pos1.coords[1] - pos2.coords[1]) + (pos1.coords
           [2] - pos2.coords[2]);
        float vel = (vel1.coords[0] - vel2.coords[0]) + (
           vel1.coords[1] - vel2.coords[1]) + (vel1.coords
           [2] - vel2.coords[2]);
        pos = (pos/link)*(pos/link);
        vel = (vel/velLink)*(vel/velLink);
        return pos + vel;
    }
```

References

[1] ANN, Approximate Nearest Neighbors. http://www.cs.umd.edu/ mount/ANN.

[2] BGL, Boost Graph Library. http://www.boost.org.

[3] Cgal, Computational Geometry Algorithms Library. http://www.cgal.org.

[4] Aarseth, S. J. *Gravitational N-Body Simulations.* Cambridge University, 2003.

[5] Agarwal, P. K., Edelsbrunner, H., Schwarzkopf, O., and Welzl, E. Euclidean minimum spanning trees and bichromatic closest pairs. *Discrete Comput. Geom. 6*, 5 (Aug. 1991), 407–422.

[6] Behroozi, P. S., Wechsler, R. H., and Wu, H.-Y. The rockstar phase-space temporal halo finder and the velocity offsets of cluster cores. *The Astrophysical Journal 762*, 2 (2013), 109.

[7] Bothun, G. Galaxy formation, July 2013. http://zebu.uoregon.edu/2002/ph123/lec08.html.

[8] Cormen, T. H., Stein, C., Rivest, R. L., and Leiserson, C. E. *Introduction to Algorithms*, 2nd ed. McGraw-Hill Higher Education, 2001.

[9] Dagum, L., and Menon, R. OpenMP: An industry-standard API for shared-memory programming. *IEEE Comput. Sci. Eng. 5*, 1 (Jan. 1998), 46–55.

[10] de Berg, M., van Kreveld, M., Overmars, M., and Schwarzkopf, O. *Computational Geometry: Algorithms and Applications*, second ed. Springer-Verlag, 2000.

[11] Ellis, M. A., and Stroustrup, B. *The Annotated C++ Reference Manual.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[12] Fernando, R. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics.* Pearson Higher Education, 2004.

[13] Friedman, J. H., Bentley, J. L., and Finkel, R. A. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw. 3*, 3 (Sept. 1977), 209–226.

[14] Glendenning, N. K. *Compact Stars: Nuclear Physics, Particle Physics and General Relativity*, vol. 1. Springer, 2000.

[15] Ho, N. Kd-tree on gpu for 3d point searching, Mar. 2011. http://nghiaho.com/?p=437.

[16] JasonSmith, M. Kd-trees, Jan. 2007. http://ldots.org/kdtree/.

[17] Jones, M. H., and Lambourne, R. J. *An Introduction to Galaxies and Cosmology*, vol. 1. Cambridge University Press, 2004.

[18] Kim, J., and Park, C. A new halo-finding method for n-body simulations. *The Astrophysical Journal 639*, 2 (2006), 600.

[19] Kirk, D. B., and Hwu, W.-m. W. *Programming Massively Parallel Processors: A Hands-on Approach (Applications of GPU Computing Series)*, 1st ed. Morgan Kaufmann, Feb. 2010.

[20] Klypin, A., Gottlber, S., Kravtsov, A. V., and Khokhlov, A. M. Galaxies in n-body simulations: Overcoming the overmerging problem. *The Astrophysical Journal 516*, 2 (1999), 530.

[21] Knebe, A., Knollmann, S. R., Muldrew, S. I., Pearce, F. R., Aragon-Calvo, M. A., Ascasibar, Y., Behroozi, P. S., Ceverino, D., Colombi, S., Diemand, J., Dolag, K., Falck, B. L., Fasel, P., Gardner, J., Gottlber, S., Hsu, C.-H., Iannuzzi, F., Klypin, A., Luki, Z., Maciejewski, M., McBride, C., Neyrinck, M. C., Planelles, S., Potter, D., Quilis, V., Rasera, Y., Read, J. I., Ricker, P. M., Roy, F., Springel, V., Stadel, J., Stinson, G., Sutter, P. M., Turchaninov, V., Tweed, D., Yepes, G., and Zemp, M. Haloes gone mad: The halo-finder comparison project. *Monthly Notices of the Royal Astronomical Society 415*, 3 (2011), 2293–2318.

[22] Kormendy, J. Dark matter halos of disk galaxies, Feb. 2000. http://chandra.as.utexas.edu/ kormendy/dm-halo-pic.html.

[23] Lacey, C., and Cole, S. Merger rates in hierarchical models of galaxy formation. ii: Comparison with n-body simulations. *Arxiv preprint astro-ph/9402069* (1994).

[24] Lam, S. K. Cuda performance: Maximizing instruction-level parallelism, Sept. 2013. http://continuum.io/blog/cudapy_ilp_opt.

[25] Maciejewski, M., Colombi, S., Alard, C., Bouchet, F., and Pichon, C. Phase-space structures i. a comparison of 6d density estimators. *Monthly Notices of the Royal Astronomical Society 393*, 3 (2009), 703–722.

[26] Maciejewski, M., Colombi, S., Springel, V., Alard, C., and Bouchet, F. R. Phase-space structures - II. Hierarchical Structure Finder. *Mon. Not. Roy. Astron. Soc. 396* (July 2009), 1329–1348.

[27] N-Body Shop. Friends-of-friends halo finder, 2013. http://www-hpcc.astro.washington.edu/tools/fof.html.

[28] NVIDIA Corporation. *CUDA C Best Practices Guide*, 4.0 ed. 2701 San Tomas Expressway, Santa Clara 95050, USA, May 2011.

[29] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*, June 2011.

[30] Nyland, L., Harris, M., and Prins, J. Fast N-Body Simulation with CUDA. In *GPU Gems 3*, H. Nguyen, Ed. Addison Wesley Professional, Aug. 2007, ch. 31.

[31] Pamplona, V. Cuda, Dec. 2008. http://vitorpamplona.com/wiki/Cuda.

[32] Pharr, M., and Fernando, R. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005.

[33] Planelles, S., and Quilis, V. ASOHF: a new adaptive spherical overdensity halo finder. *Astron. Astrophys. 519* (Sept. 2010), A94+.

[34] Seymour, M. D., and Widrow, L. M. Multiresolution analysis of substructure in dark matter halos. *The Astrophysical Journal 578*, 2 (2002), 689.

[35] Snir, M., Otto, S., Huss-Lederman, S., Walker, D., and Dongarra, J. *MPI-The Complete Reference, Volume 1: The MPI Core*, 2nd. (revised) ed. MIT Press, Cambridge, MA, USA, 1998.

[36] Sparke, L. S., and Gallagher, J. S. *Galaxies in the Universe: An Introduction*, vol. 1. Cambridge University Press, 2007.

[37] SUMMERS, F., DAVIS, M., AND EVRARD, A. Galaxy tracers and velocity bias. *Astrophys 454*, 1 (May 1995), 1–14.

[38] MULTIDARK DATABASE. Halo finders, Dec. 2013. http://www.multidark.org/MultiDark/Help?page=halofinders.

[39] N-BODY SHOP, UNIVERSITY OF WASHINGTON. FOF, Mar. 2013. http://www-hpcc.astro.washington.edu/tools/fof.html.

[40] TRIMBLE, V. Existence and Nature of Dark Matter in the Universe. *Ann.Rev.Astron.Astrophys. 25* (1987), 425–472.

[41] VINEET, V., HARISH, P., PATIDAR, S., AND NARAYANAN, P. J. Fast minimum spanning tree for large graphs on the GPU. In *Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), HPG '09, ACM, pp. 167–171.

[42] VOIT, G. M. Tracing cosmic evolution with clusters of galaxies. *Reviews of Modern Physics 77* (Apr. 2005), 207–258.

[43] WILT, N. *The CUDA Handbook: A Comprehensive Guide to GPU Programming*, first ed. Addison-Wesley, 2013.

[44] ZHOU, K., HOU, Q., WANG, R., AND GUO, B. Real-time kd-tree construction on graphics hardware. In *ACM SIGGRAPH Asia 2008 papers* (New York, NY, USA, 2008), SIGGRAPH Asia '08, ACM, pp. 126:1–126:11.