Working With Incremental Spatial Data During Parallel (GPU) Computation

By:
Robert Chisholm

Supervised By:
Dr Paul Richmond & Dr Steve Maddock

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy

The University of Sheffield
Faculty of Engineering
Department of Computer Science

24th January 2020

Abstract

Central to many complex systems, spatial actors require an awareness of their local environment to enable behaviours such as communication and navigation. Complex system simulations represent this behaviour with Fixed Radius Near Neighbours (FRNN) search. This algorithm allows actors to store data at spatial locations and then query the data structure to find all data stored within a fixed radius of the search origin.

The work within this thesis answers the question: What techniques can be used for improving the performance of FRNN searches during complex system simulations on Graphics Processing Units (GPUs)?

It is generally agreed that Uniform Spatial Partitioning (USP) is the most suitable data structure for providing FRNN search on GPUs. However, due to the architectural complexities of GPUs, the performance is constrained such that FRNN search remains one of the most expensive common stages between complex systems models.

Existing innovations to USP highlight a need to take advantage of recent GPU advances, reducing the levels of divergence and limiting redundant memory accesses as viable routes to improve the performance of FRNN search. This thesis addresses these with three separate optimisations that can be used simultaneously.

Experiments have assessed the impact of optimisations to the general case of FRNN search found within complex system simulations and demonstrated their impact in practice when applied to full complex system models. Results presented show the performance of the construction and query stages of FRNN search can be improved by over 2x and 1.3x respectively. These improvements allow complex system simulations to be executed faster, enabling increases in scale and model complexity.

I, the author, confirm that the Thesis is my own work. I am aware of the University's Guidance on the Use of Unfair Means (www.sheffield.ac.uk/ssid/unfair-means). This work has not been previously been presented for an award at this, or any other, university.

Name:	Robert Chisholm
Signature:	
Date:	24th January 2020

Acknowledgements

I would like to acknowledge & extend my gratitude to the following persons who have made the completion of this research possible:

Dr Paul Richmond for sharing his GPU passion and knowledge.
Dr Steve Maddock for his regular discretionary guidance correcting my grammar and 'logic train'.
Dr Daniela Romano for the earlier supervision she provided.
Peter & Mozhgan for general support and rubber duck debugging around the office.
My family for the unconditional support.
The many friends who got me into fitness, and kept me sane: Aisha, Andres, Andy, Bader, Basti, Chris, 'Wolf', James, Jeremy, Kieran, Roman, Sunny & countless others (also Ben).
GBC for the spirited conversation.
Becca and the many others, not listed, who have also contributed to my general well being.

Contents

List of Figures

List of Tables

List of Acronyms

ACS Atomic Counting Sort. VII, VIII, 59, 67, 68, 70, 72, 74–83, 117, 118, 123

AoS Array of Structs. 27

API Applications Programming Interface. 27

ASICs Application-Specific Integrated Circuit. 29, 30

AVL A form of binary tree that stores additional information, allowing it to maintain balance.. 17

BVH Bounding Volume Hierarchy. 17

CA Cellular Automata. 8

CPU Central Processing Unit. 1, 11, 20, 21, 25–30, 34, 35, 37, 38, 45

CUDA Compute Unified Device Architecture. 22–28, 39

CUDPP CUDA Parallel Primitives. 38, 39

ECC Error Correction Code. 24

fp32 32 bit Floating Point; single precision decimal. 24

fp64 64 bit Floating Point; double precision decimal. 24

FPGAs Field Programmable Gate Arrays; Programmable integrated circuits organised in a matrix. 29

FRNN Fixed Radius Near Neighbours. I, VI, VII, 2–12, 14–20, 25, 29–31, 36–38, 40, 41, 44–53, 57, 58, 61, 62, 64–66, 71, 72, 74, 82–90, 92–95, 97, 98, 101, 102, 104–106, 111, 113–116, 118, 119, 121, 123–129

GPGPU General Purpose computation on Graphics Processing Units. IV, 3, 6, 10, 11, 20, 21, 25, 27, 29, 34, 35, 39, 47, 127

GPU Graphics Processing Unit. I, IV, V, VII, 1–6, 9–11, 20–30, 32, 33, 35–40, 42–50, 57–59, 62, 64–67, 72, 76, 78, 82, 84, 87, 88, 90, 95, 100, 104, 111, 114, 116–119, 121, 124, 126–128

HPC High Performance Computation. 6, 20, 24, 25, 29, 30, 117, 128

LOS level of service. VIII, 122–124

MAS Multi-Agent Systems. 4, 30, 39, 42, 62, 71, 78, 85, 90, 94, 95, 113, 117, 118, 124

MD Molecular Dynamics. 32, 38, 46, 128

MIC Many Integrated Core; Intel's (Xeon Phi) compute architecture. 29

MPI Message Passing Interface. 21

nvcc NVIDIA CUDA Compiler. 26

NVLink NVLink is the brand name for NVIDIAs high-speed GPU interconnect technology.. 21

NVSwitch NVSwitch is the brand name for NVLink in a switch arrangement to enable communication between upto 16 devices simultaneously.. 21

OpenACC Open Accelerators. 27

OpenCL Open Computing Language. 27, 28

OpenGL Open Graphics Library. 27

PBM Partition Boundary Matrix. VII, 41–44, 59, 65–70, 86, 103–105, 113

PCIe PCI Express. 21

SETI The Search for Extra-Terrestrial Intelligence. 28

SFU Special Functions Unit. 24

SIMT Single Instruction Multiple Threads. 21, 22, 26, 32, 35, 39, 43, 58, 66, 84, 87, 100, 118

SLI Scalable Link Interface (SLI) is the brand name for NVIDIAs multi-GPU technology.. 21

SM Stream Multiprocessor. 21–26, 116

SoA Struct of Arrays. 27

SPH Smoothed-Particle Hydrodynamics. VII, 10, 11, 32, 38, 46–49, 68, 74, 82, 90

STR Sort-Tile Recursive. 18

USP Uniform Spatial Partitioning. I, VII, VIII, 2–6, 8, 9, 11, 33, 36, 38–41, 43–50, 52, 58, 59, 64–68, 71, 73, 75, 78, 80, 82–85, 97, 98, 105, 106, 115, 126, 128

WEE The average percentage of active threads in each executed warp. Such that a value of 100% indicates no divergence.. X, 89, 93–95, 115

Chapter 1

Introduction

1.1 Background

Complex systems can be identified by their large number of autonomous actors often working arranged in organisational hierarchies. It is the interaction of these many actors that combine and feedback into the system in a continuous cycle which produces high-level emergent behaviour. This process allows many small independent actions to combine, producing visible and robust high-level patterns of emergent behaviour.

These complex systems are found everywhere in daily life, both in our surrounding environment and among the processes that permit our bodies to continue functioning. These systems exist from scales not visible to the human eye, such as the network of neurons within a brain, to those which extend across continents, like the organisational behaviour of ant colonies. Furthermore, abstract complex systems can also be found, such as the global economy.

It is often impractical to directly study complex systems because the systems are too large to control, too expensive to measure or simply too harmful to modify. Understanding how a system's chaotic low-level interactions affect its high-level behaviour is of interest to many researchers in fields from Astronomy to Social Science.

The introduction of general purpose programming of Graphics Processing Units (GPUs) in 2001 opened up the highly scalable and parallel architecture for the development of complex system simulations. The hierarchical structures, comprised of many similar actors, that form complex systems are ideal for highly parallel computation and, as such, most complex system models can be developed to target GPUs. However, developing parallel algorithms for use with GPUs introduces challenges such as communication between individuals, conflict resolution for limited resources and a requirement to optimise for hardware with different performance-critical features than those present in CPUs.

There are now many frameworks that reduce the skills required to implement models of complex systems, although these primarily target CPUs with a minority capable of utilising

GPUs (e.g. FLAMEGPU [137]). However, there is a trade-off between general applicability and performance: the more general a framework the less it can take advantage of domain specific optimisations. This is particularly apparent when we consider the spatial data structures required by many complex simulations, which is the focus of this thesis.

Many complex systems contain mobile spatial actors that are influenced by their neighbours whether they are particles, people, vehicles or planets. At each timestep, actors first survey neighbours within a spatial radius before deciding their change of state. This process is called Fixed Radius Near Neighbours (FRNN) search. Complex system models require FRNN search to be performed over a variety of distances and some models may share additional data alongside neighbour locations. The performance of the data structure used to manage this dynamic spatial data and deliver FRNN search is an active area of research [149, 44, 81].

Most data structures for storing spatial data are highly application specific. Whilst kd trees are well suited to delivering Nearest Neighbour (NN) search, they are unsuitable for returning the larger numbers of neighbours required by general case FRNN search. Conversely, any data structure suitable for providing FRNN search is unlikely to outperform kd trees when used specifically for NN search.

Applications utilising static spatial data have many options covering a wide range of use-cases available to them [51, 110, 53, 63, 144] (discussed in Section 2.2), with many GPU alternatives (discussed in Section 2.4). Many of these static techniques, however, utilise costly optimisation strategies at construction to ensure queries are performant, making them undesirable for highly dynamic data.

There are few data structures capable of handling dynamic spatial data on GPUs (also discussed in Section 2.4). This has led most complex simulations handling dynamic spatial data to rely on Uniform Spatial Partitioning (USP) to provide FRNN searches.

FRNN search using the USP data structure consists of two distinct stages: construction and querying. Despite its use with complex system simulations, representing data with frequent incremental movement, it is a static data structure requiring a full reconstruction when any item moves, irrespective of the degree to which the data has moved.

The construction process of USP decomposes the continuous environment into a discrete grid of bins, sorts all items according to their bin and builds an index to each bin's storage in memory. Existing research towards construction has primarily targeted the sorting stage [149]. However attempts to produce a more efficient parallel sort, with the same handling of pre-sorted data as serial insertion sorts, have also reduced FRNN search to an approximate technique [86, 85, 41].

Most research focus around improving the performance of FRNN search with USP has attempted to improve the query stage, whereby all items within a Moore neighbourhood (the set of directly adjacent discrete spatial bins) of the search origin's bin are depth tested to ensure they fall within the fixed radius. These approaches have predominantly attempted to improve the coherence of the highly scattered memory accesses whilst items from different bins are being accessed in parallel. However, recent improvements to GPUs have significantly improved the caching of memory reads reducing the impact of such techniques.

Researchers have used many names to describe the USP data structure, including uniform subdivision, grid method and cell list [104, 56, 149, 85]. Yet FRNN search still remains one of the most costly aspects of complex system simulation [44], with many avenues left to be investigated to further advance the performance of both the construction and query stages of FRNN searches of dynamic spatial data on GPUs. In particular, the research in Chapter 2 highlights advances to GPU hardware, high levels of divergence and redundant memory accesses as areas for further investigation.

This thesis seeks to address the challenges faced when handling the dynamic spatial data used in complex system simulations on GPUs, in particular the process of FRNN search. Existing techniques used for handling spatial data and optimising GPU algorithms have been investigated. This, in combination with an understanding of how FRNN search is currently performed, including state of the art approaches for optimisation, has been used to develop several novel optimisations. These optimisations are then shown to be beneficial to a wide range of research disciplines by facilitating the production of larger and faster complex system simulations.

1.2 Aims & Research Question

The aim of this thesis is to investigate and improve upon the performance and use of data structures capable of providing FRNN searches for complex system simulations on GPUs. This work will demonstrate how the performance of any GPGPU applications requiring FRNN searches of dynamic spatial data can be improved.

This research answers the question: What techniques can be used for improving the performance of FRNN searches during complex system simulations on GPUs?

In particular this thesis will address the question by focusing on modern GPU approaches to sorting at the construction stage, and reducing divergent code and redundant memory accesses at the query stage.

The evaluation of improvements to FRNN search will be carried out using both theoretical analysis and benchmarks, to assess the impact across a wide range of complex system simulations. Whilst analysis can be used to initially reduce the scope, it is necessary to also apply techniques in practice using benchmarks, as the complex architecture of GPU hardware leads to performance which does not always map directly to abstract theoretical understanding. Benchmarks must be carried out in a diagnostic form, to enable understanding of how varying properties such as population size and density, which can affect performance, are handled by optimisations. Additionally, any developed techniques must be evaluated with real modelling scenarios, as it is important to demonstrate their feasibility in practice.

1.3 Contributions

This thesis makes the following novel contributions in delivering the above objectives:

- **C1** - Description and performance evaluation of a novel optimisation for construction of the USP data structure: knowledge of atomic operation performance in modern GPU hardware enables the sorting algorithm central to construction of the USP data structure to be changed. By instead using atomic counting sort, an intermediate stage of construction is produced as by-product of the sort.
- **C2** - Description and performance evaluation of a novel optimisation for the query operation of FRNN search using the USP data structure: understanding of the FRNN search's query operation allows the removal of redundant operations, providing benefits via the removal of divergent code.
- **C3** - Description and performance evaluation of a second novel optimisation for the query operation of FRNN search using the USP data structure: theoretical analysis of the trade-off between redundant area accessed and subdivision resolution leads to an optimisation which adjusts the resolution of the environment's subdivision. This optimisation has the effect of reducing the volume of redundant messages accessed when mapping the radial search area to a set of bins.
- **C4** - Demonstration of the potential of these optimisations, in combination, for improving the performance of complex system simulations, by comparing the performance of FRNN search within FLAMEGPU models before and after application of the optimisations. Performance was shown to be improved for all but the most extreme cases of low density and device under utilisation

1.3.1 Publications

The work in this thesis has led to the following publications:

- R. Chisholm, P. Richmond & S. Maddock - A Standardised Benchmark for Assessing the Performance of Fixed Radius Near Neighbours (2016)[38]
- R. Chisholm, S. Maddock & P. Richmond & - Improved GPU Near Neighbours Performance Through Proportional Bin Widths (2020) [37]

1.4 Summary of Chapters

- Chapter 2 (Background Literature) introduces: the requirement of FRNN search in Multi-Agent Systems (MAS); how MAS requiring FRNN search are most suited to execution GPU; and spatial data structure background literature which demonstrates why USP is the most appropriate data structure for GPU FRNN search. It then investigates the specification and operation of the USP data structure and FRNN search algorithm, introducing state of the art optimisations, performance considerations and further applications of the data structure.

- Chapter 3 (Benchmark Models) provides a specification and justification for the models used to assess the optimisations presented within the following chapters.
- Chapter 4 (Optimising Construction) demonstrates a potential optimisation for the construction of the USP data structure which takes advantage of recent GPU hardware advances, providing Contribution C1.
- Chapter 5 (Optimising Queries: Reducing Divergence) details the relative costs of the elements of the FRNN query algorithm, identifying reducing divergent code leads to a potential optimisation, providing Contribution C2.
- Chapter 6 (Optimising Queries: Surplus Memory Accesses) builds on the optimisation from the previous chapter and addresses the redundant memory accessed during the FRNN query algorithm, delivering a further optimisation that works in combination with the prior optimisations, providing Contribution C3.
- Chapter 7 (Case Studies & Discussion) the contributions from this thesis are applied to two potential applications, demonstrating how the knowledge contained in this thesis can be applied in practice, providing Contribution C4.
- Chapter 8 (Conclusions) collates the core insights and developments from this thesis to provide a concise reference of how FRNN search on GPUs can be best applied to simulations of complex systems.

Chapter 2

Background Literature

This chapter provides general context and background to Fixed Radius Near Neighbours (FRNN) search and how it is applied. Furthermore, it explores methods which can be used for storing and handling spatial data. The work in this chapter has been used to shape the research question.

Section 2.1 introduces the field of complex system simulations. These simulations usually rely on spatial communication, often via FRNN search, ranging from scales not visible to the human eye, like cellular interaction, to communication across a global scale. As such, they are a key application for which FRNN is used.

Section 2.2 defines and introduces spatial data structures, beginning by providing examples from literature of serial techniques and how they manage storage of and access to spatially located data. Many of these data structures would not be suitable for providing FRNN search, however, they provide an important understanding of techniques for handling spatially located data.

With High Performance Computation (HPC) as the primary platform of complex system simulations, Section 2.3 delivers an overview of HPC techniques. The focus of this section is the architectural considerations of GPGPU and how this affects algorithm development, as this is the platform of focus within this thesis.

After which Section 2.4 introduces parallel capable data structures for handling spatially located data. Most of these we term static, where updating elements requires rebuilding the whole data structure. Dynamic data structures, where elements can be introduced and updated incrementally could be most suitable for complex system simulations, however, hardware and architectural limitations of GPUs make them more difficult to deliver.

Following the previous sections which give context to the handling spatial data in high performance simulations Section 2.5 explores Uniform Spatial Partitioning (USP), which has been shown to be the most prominent technique, for handling FRNN search in depth and state of the art techniques used to improve it's performance.

The final section of the chapter concludes by summarising areas for further research, which have been highlighted by the literature, leading to the following chapters which investigate them.

2.1 Complex Simulations

Complex simulations are one of the primary fields where FRNN search is utilised. Many of these simulations model the behaviour of a large number of mobile actors that interact with neighbouring actors and the local environment. These simulations regularly utilise high-performance computing environments, where improvements to performance allow models to increase in both scale and complexity.

The following subsections provide some examples of complex simulations which stand to benefit from improvements to the handling of FRNN search.

2.1.1 Pedestrian Modelling

Pedestrian & crowd modelling are concerned with the replication of the behaviour of pedestrians in reaction to each other and the environment. Models can be separated into two approaches: accurate (data driven) & aesthetic (visually driven). Whilst they are concerned with different properties for evaluation and performance considerations, they both require the replication of natural pedestrian behaviour.

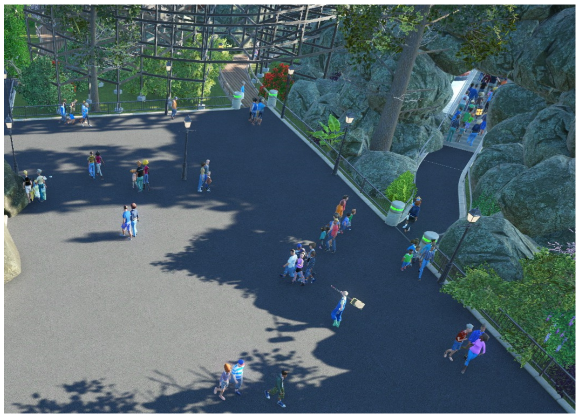


Figure 2.1: Games such as Planet Coaster aim to simulate visually realistic and appealing crowds. *Screenshot taken within the game Planet Coaster.*

Common to both of these fields is the need for performant and often large scale crowd simulations. Accuracy driven implementations are expected to run faster than real-time to allow hundreds of simulations to be completed to account for stochastic behaviours and to potentially review many architectural designs. Visually driven implementations are typically required with video games and overall require the algorithms controlling crowds to run concurrently in real-time with the various other methods required to make a game function. The crowd included in a game must be visually convincing, but performant enough that it does not impact on the performance of the game. This has previously been achieved by either reducing the size of crowds or utilising simpler rules of motion to benefit performance [48].

Earlier data driven pedestrian models initially used a macroscopic approach based on fluid dynamics techniques [73, 28, 155]. Modern techniques however primarily take a microscopic approach, modelling each individual pedestrian. The first of these techniques, the theory of social forces, was proposed by Helbing and Molnar in 1995 [72]. The social forces model proposes that pedestrians are influenced by 4 forces. This technique has been applied to a range of applications including evacuation and detection of suspicious behaviour [145, 61, 111, 71]. Hardware limitations have also led some to simplify modelling by reducing a continuous environment into a discrete grid via Cellular Automata (CA) [162, 132, 168].

Of the styles of modelling available for crowds, the microscopic nature of the continuous social forces model provides the highest fidelity. Within a social forces model, it should be apparent that pedestrians are essentially spatial actors that smoothly avoid collisions under regular circumstances. For pedestrian actors to handle collision avoidance, it becomes necessary for them to access data related to neighbouring pedestrians. This, coupled with their mobile nature, makes them suited to benefit from improvements to FRNN search.

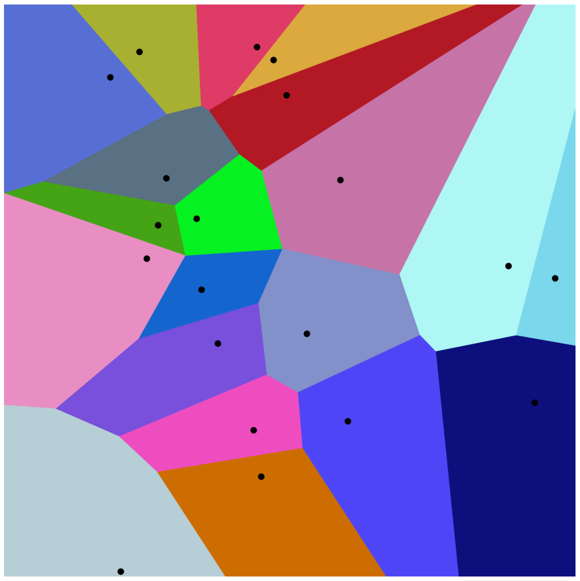


Figure 2.2: An example 1st order Voronoi diagram, each polygon corresponds to the area closest to the contained seed (black dot).
CC by-sa 4.0, wikipedia:Balu.Ertl

Microscopic pedestrian models often use spatial subdivision to store their spatial actors. This stems from Reynolds's original 'flocking' paper which paved the way for much of the early pedestrian simulations [136]. The simplest example is that of Uniform Spatial Partitioning (USP), whereby the environment is decomposed into a uniform grid. Each pedestrian is then stored into the bin within which its location resides. Pedestrians regularly move, changing their location. As a static data structure it must be fully rebuilt each time step to accommodate these movements [58]. There are, however, more complex solutions, such as the Multi-actor Navigation Graph (MaNG), developed by Sud et al [148]. This technique utilises first and second order Voronoi diagrams to identify and encode the clearance between actors into the data structure, however, it still requires full reconstructions each iteration. The use of Voronoi diagrams, which partition space into polygons of area closest to a point (Figure 2.2), acts as a pre-processor by calculating pairwise proximity information which is stored within the MaNG. Paths of maximal clearance are calculated as a by-product of the data structure's construction. However, the time complexity for construction of the MaNG on a discrete grid of resolution $m\ x\ m$ is presented as $O(m^2 + n \log m)$. The m^2 component is unlikely to scale effectively for larger environments. Their largest experiment demonstrated 200 actors on a grid of 1000 x 1000 resolution, producing a graph of 1927 vertices and 14669 edges, suggesting their technique is also limited to smaller crowds.

2.1.2 Traffic

Traffic modelling concerns the replication of the behaviour of vehicles on the road network. It has been used to predict the effect that network changes and events will have on traffic. There is also the wider field of multi-modal transport modelling that seeks to combine traffic, pedestrian, train and other transport models towards a unified model of transport systems [32]. However, this subsection will focus on traffic modelling, as pedestrian modelling has been covered above, and the techniques used in the implementation of other transport models are mostly analogous to either traffic or pedestrian models.

Whilst existing traffic models are often concerned with models, within counties, of specific areas and roads, there is always demand for increasing performance which would permit models to be executed more frequently or at a greater scale. For example, within the UK, local authorities and Highways England both utilise various traffic models for evaluation of traffic within their jurisdictions [13].

There are several approaches to traffic modelling: macroscopic, mesoscopic, microscopic and hybrid techniques [94]. The macro, meso and micro techniques are each concerned with modelling traffic at different levels of detail. Macro is concerned with modelling traffic on a per-road basis, meso is concerned with traffic in a grouped or platoon of vehicles basis, and micro is concerned with modelling the interaction of individual vehicles. As the modelled focus becomes smaller (roads, groups, vehicles), more actors must be simulated, leading to increased fidelity at a higher computational cost. The hybrid approach seeks to utilise aspects of microscopic models to enhance macroscopic models, whilst avoiding the full computational cost required of microscopic models.

Whether the operations are occurring for each road (macroscopic) or each vehicle (microscopic), the uniform nature of actors makes them ideal for GPU acceleration. Of these techniques, microscopic simulation requires intense computation and simulates many mobile actors (vehicles), therefore it is a suitable candidate to benefit from FRNN search.

There are a number of available commercial tools for traffic modelling (e.g. AIMSUN [22], Saturn [65] & VISSIM [50]). However, of the tools available, most underlying traffic models were initially developed decades ago. As such they are primarily macroscopic (due to the available computational power at the time making microscopic simulation less feasible). The open-source microscopic traffic model, Simulation of Urban MObility (SUMO), currently represents lanes as vectors, where each element is a vehicle [123]. Whilst this is intuitive, due to the nature of cars being constrained to a graph, this does not align vehicles with other vehicles in neighbouring lanes.

The USP data structure used to provide FRNN search for pedestrian models is parallel multi-map, a key-value store whereby keys can map to multiple values, and hence can instead be used to represent data tied to discrete locations such as edges of a graph. This could instead be used to represent cars within a road network, similar to how SUMO operates. Whilst this use case is unlikely to utilise FRNN search, it would still benefit from improvements to how the USP data structure is constructed.

However, shared vehicle-pedestrian space models are increasingly required within modern urban design. A consequence of the inclusion of pedestrians within transport models is the growing demand for the representation of continuous spatial actors within traffic models, which is provided by FRNN search.

2.1.3 Smoothed-Particle Hydrodynamics

Smoothed-Particle Hydrodynamics (SPH) is a computational technique used for the simulation of fluid dynamics. The technique was first developed in 1977 at the Institute of Astronomy in Cambridge, UK, applying the Newtonian equations for particles in order to solve astrophysics problems [54, 105]. Since then SPH has been further applied to Astrophysics, alongside other fields including Ballistics, Graphics, Oceanography, Solid mechanics & Volcanology [116]. Figure 2.3 shows an example of SPH being used to simulate blood motion.

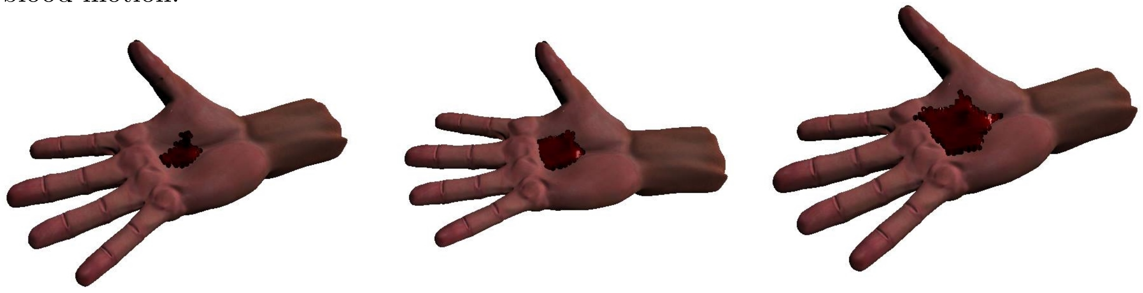


Figure 2.3: Simulation of blood flowing over a hand which was generated using SPH.
Used with permission of Matthew Leach, University of Sheffield

SPH decomposes a fluid into a discrete set of particles. A variable 'smoothing length' is used, such that any particle's properties can be calculated from the neighbouring particles that reside within range of the desired particle. Each particle's contribution to the calculation of a property is weighted according to its distance from the particle whose properties are being computed. A Gaussian function or cubic spline is often used to calculate this weight.

The representation of particles within fluids consists of many thousands of mobile particles, each surveying their neighbouring particles. This shows that, similar to pedestrian and crowd modelling, SPH would benefit from a spatial data structure capable of providing FRNN searches. However, whilst pedestrians are most often modelled in 2 dimensions, SPH is almost exclusively carried out in 3 dimensions.

Due to the high numbers of particles that are following the same rules, this makes SPH ideal for GPU computation. With the advent of General Purpose computation on Graphics Processing Units (GPGPU) software, many SPH simulations have been accelerated to real-time, and some researchers are now taking advantage of multi-GPU implementations to achieve even larger simulations [45].

Harada et al's early SPH GPU implementation simply stored particle identifiers inside a bucket texture, whereby the environment is decomposed into a regular grid of buckets and each particle is assigned to a bucket which is represented as a pixel within the texture [66]. More recent implementations using GPGPU techniques utilise spatial subdivision to provide FRNN search, similar to that used in pedestrian modelling. Researchers in the field of SPH have found that the performance of query stage of FRNN search can be improved through the utilisation of space-filling curves [56, 141] – these are explained in more depth in Section 2.5.

2.1.4 General Approaches

Outside of specific models, there are also frameworks and toolkits that ease the development of multi actor simulations for researchers unfamiliar with advanced programming concepts. Many of these offer functionality for handling dynamic and continuous spatial data which can be utilised to implement the models described above. However, their chief concern lies with providing an intuitive interface for users to develop models with. This creates a trade-off whereby generality is gained at the cost of domain-specific optimisations. These frameworks provide an alternate perspective from which to consider whether a spatial data structure capable of providing FRNN searches would have a simple enough interface and generality to be applicable for their users.

There are many agent based modelling frameworks (NetLogo [153], MASON [106], Repast [117], FLAME [92], etc). The majority of these target a single machine and are limited to execution on a CPU. A small number have been extended to utilise distributed computing [39]. More recently FLAMEGPU has provided a GPGPU agent based framework, allowing domain experts with limited programming expertise to utilise GPU computation [92]. The FLAMEGPU framework currently handles spatial data using the same spatial partitioning method used by pedestrian & crowd modelling, USP, whereby spatially located actor messages are partitioned into a uniform grid every time-step, thus allowing neighbourhood searches.

Currently, most existing actor modelling frameworks target CPU implementations. The scalability of GPGPU has been shown [46] and this has catalysed the growth of GPU computation. Therefore it is clear that improvements to techniques for working with spatial data on GPUs will become more beneficial over time.

2.1.5 Summary

This section has shown the breadth of fields whereby dynamic spatial data is handled during complex simulations, and how many of these have started to adopt the use of GPU computation within the past decade to boost performance. It has also detailed some existing techniques used by these implementations for handling spatial data, identifying USP as the common data structure currently used for providing FRNN search. This provides confidence that an improvement to the performance of FRNN searches would be capable of impacting a wide audience.

2.2 Spatial Data Structures

There are many existing techniques used for working with spatial data, covering a plethora of applications. The majority of these data structures are static, relying on a complete reconstruction to accommodate the movement of even a small number of elements. The section below discusses existing data structures, examining the underlying techniques used for organising spatially located data and the functionality they provide. Whilst not all of these data structures provide the FRNN search functionality required of complex system simulations, it is valuable to understand how they handle spatial data to determine whether the techniques are applicable to the delivery of FRNN searches.

The following subsections discuss some of the techniques used by hashing and tree data structures. These two families of data structure are both highly suitable for storing spatial data.

2.2.1 Hashing Schemes

Hash-Tables, Hash-Maps and Hash-Sets are a family of data structures that fundamentally rely on hashing to provide an index that is used in the storage and retrieval of data from an array. Hashes are good for working with any data whereby random access is required and keys do not form a contiguous block or regular pattern. This makes them suitable for storing spatially located data. Some hashes used with spatial data allow data to be stored coherently in bins of their local neighbourhood. Whilst this locality benefits memory accesses, most hashing data structures rely on data being uniformly distributed. There are many different methods for implementing hashing data structures and each implementation has slightly different properties that affect the performance of insert and query operations.

Hashing structures are inherently statically sized. Items are placed into an array according to how their hash indexes into the array. This does however mean that once the load-factor (the ratio of items to maximum items) becomes too high, collisions are increasingly likely and the data structure must be rebuilt with larger storage (some implementations may also rebuild if an item's hash exceeds a given number of collisions during insertion). This process can be time consuming when working with unknown or variable quantities of data, as every item must be re-hashed. The action of growing a hash-table requires memory for both the old and new hash-tables simultaneously, which can easily lead to exceeding memory bounds when working with large tables or within tight memory constraints. Similarly, as items are deleted, it may be necessary to rebuild the data structure to reduce its memory footprint.

Good Hash Functions

In most cases a hash function should deliver approximate uniform hashing to reduce collisions. It is most common for input keys to be reduced to either be provided as or coerced into an integer. This could take the form of reducing spatial coordinates to a single integer. It can be important that the range of keys is assessed prior to coercion, to ensure the coercion itself is unlikely to generate collisions. The integer form of the key is then mapped to an index for storage via a hash function. There are various techniques for creating an appropriate hashing function.

When working with known data there are techniques for creating perfect hash functions, where all hashes are unique. These are more applicable to the handling of spatial data within complex systems, as continuous locations may be discretised and often environments are a known finite scale. If a 2-dimensional environment is bounded by a square boundary the environment can be discretised with a uniform grid with cells indexed linearly, providing a means of coercion from a 2-dimensional continuous location to a single integer.

Collision Resolution

When the input set of items to a hash function is unknown it is not possible to generate a hash function which avoids multiple inputs producing the same hash. Furthermore, some hashing data structures allow multiple values to be attached to the same input. There are two approaches to handling these collisions, whilst retaining efficient insert and lookup performance: separate chaining and open-addressing.

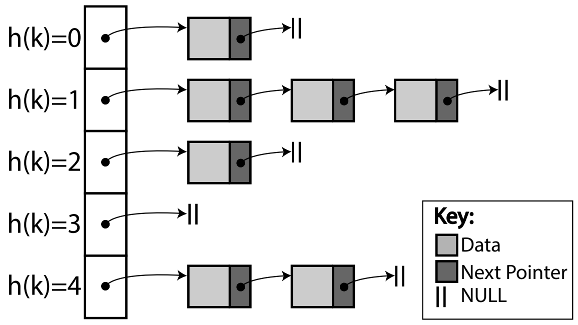


Figure 2.4: A visual representation of a hash table that utilises separate chaining.

Separate chaining is a technique whereby items with the same hash are stored in a list at that address (Figure 2.4). This technique requires the load-factor of a table to be maintained at a level such that chains do not become long [95].

In contrast, open-addressing techniques seek to place all items within the same address space. This can lead to primary and secondary clustering, whereby many items collide before a suitable location is found. It also becomes necessary within open-addressing schemes to replace deletions with a special delete value. If deleted items were instead marked as empty, searches would fail to find items that had previously collided with the now empty location. As such, deletions may not speed up searches under open-addressing.

To place items under open-addressing, probing is used, whereby the value of the hash is incremented with a fixed counter (linear probing, Figure 2.5), the second power of a counter (quadratic probing) or a hash of a counter (double hashing). This occurs until a unique index is found. Quadratic probing using triangular numbers rather than a counter avoids secondary clumping issues, assuming a good initial hash function, leading to improved insertion at a high load-factor [93].

There are numerous other techniques of collision resolution that build on separate chaining and open-addressing such as cuckoo hashing [128], hopscotch hashing [75], Robin Hood hashing [34] and coalesced hashing [158]. These techniques utilise eviction of existing items and additional hashing functions to reduce the worst case search time. However, the best and average cases are often outperformed by common techniques (e.g. linear probing), so use is largely limited to tasks requiring their worst case search guarantees.

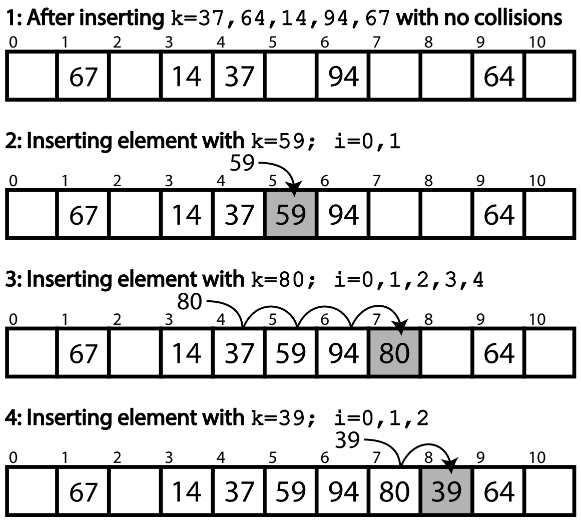

1: After inserting k=37,64,14,94,67 with no collisions
0 1 2 3 4 5 6 7 8 9 10
67 14 37 94 64
2: Inserting element with k=59; i=0,1
59
67 14 37 59 94 64
3: Inserting element with k=80; i=0,1,2,3,4
80
67 14 37 59 94 80 64
4: Inserting element with k=39; i=0,1,2
39
67 14 37 59 94 80 39 64

Figure 2.5: A visual representation of a hash table that utilises linear probing using the hash function $h(k,i) = ((k+i) \; mod \; 11)$.

When dealing with spatial data in continuous space and hashing data-structures, it becomes necessary to discretise the space so that queries can be performed by referring to an area, rather than requiring the exact coordinates. Within a complex system simulation, this is likely to lead to multiple actors sharing the same key, hence a multi-map or form of separate chaining would be more favourable for retrieving all elements at a given discrete location. This is a common feature within parallel implementations of FRNN search using uniform spatial subdivision, discussed further in Section 2.4.

Some applications of hashing data structures, e.g. nearest neighbour search (which returns a single result), are instead concerned with ensuring that collisions occur for pairs within a fixed proximity. This approach is termed locality-sensitive hashing and works around the basis that if two points are close together in high dimensional space they should remain close after projection to a low dimensional space. However, far apart points may also become close after this same projection (e.g. the poles of a sphere, where from the side they appear separated, but from above or below they overlap).

Cao et al provide a thorough review of locality-sensitive hashing techniques [31], in particular, Rasmus Pagh's proposed locality-sensitive hashing scheme, which does not produce false negatives [127]. Their theoretical technique improves over earlier approximate methods, with little or no cost to the efficiency (measured in required memory accesses). The approach hashes keys with multiple hashing functions, where each represents different projections, to construct a bitmask identifying projections whereby the points appear within the desired range. The innovation allows projections to be selected such that all possible ranges are covered, such that false negatives are eliminated. This work leaves open both the questions whether it can be applied to areas beyond Hamming distance, and whether the gap in efficiency can be completely closed, or proven necessary.

Importantly, locality-sensitive hashing schemes are concerned with encoding the separation across a population. This contrasts with FRNN search which seeks to identify each point's separation from every other point independently. Hence, this approach is unlikely to out-perform perfect hashing via the linear discretisation of a continuous environment into a grid, which is far more appropriate for FRNN search and the low dimensionalities found within complex system simulations.

Summary

This subsection has explained the central components of hashing data structures and how they can be applied to the storage of spatial data. However, in order to maintain spatial coherence, open-addressing schemes must be avoided due to their reliance on uniformly distributed keys. It is worth considering the performance benefits of spatial coherence, both in the sense of how it facilitates neighbour access and may improve performance via coalesced memory operations.

Of the hashing techniques covered, spatially coherent perfect hashing, whereby continuous space is discretised into a linearly indexed uniform grid appears most appropriate for retaining the locality information required by FRNN search. Other techniques such as locality-sensitive hashing may have applications within algorithms requiring proximity information, e.g. nearest neighbour, however, the difference in algorithm focus would lead to significant inefficiencies if applied to FRNN search due to the high number of queries required.

The challenges introduced by implementation of a hashing scheme in parallel are explained in Section 2.4.1, which includes greater detail on spatial techniques.

2.2.2 Trees

A tree in computing consists of a root node, which is the parent of one or more child nodes. Subsequently each of these child nodes may have one or more children of their own, and so on. A node with no children is referred to as a leaf node.

The maximum number of children a node may have is referred to as a tree's branching factor. This value controls how the tree will branch out as nodes are inserted. Different types of tree have different branching factors.

There are many classes of tree; search trees (2-3, 2-3-4, AVL, B, B+, Binary Search, Red-Black, etc), heaps (Binary, Binomial, etc), tries (C-trie, Hash, Radix), spatial partitioning trees (BSP, k-d, Octree, Quad, R, R+, etc) and several other miscellaneous types.

Spatial Trees

There are several forms of spatial tree and their purposes vary from rendering to providing access to large databases of spatial data.

BSP trees are the data structure used in binary spatial partitioning, an algorithm for the recursive subdivision of space by hyperplanes into convex subspaces. First published in 1980 by Fuchs et al [53], BSP trees provide rapid access to the front-to-back ordering of objects within a scene from the perspective of an arbitrary direction. This makes them ideal for rendering, whereby the visibility of faces within a scene must be computed regularly. The binary spatial partitioning algorithm splits entities that cross the partitioning hyperplanes, which can cause a BSP tree to have far more nodes than entities present in the original input.

k-d trees are binary trees (branch factor of two) that form a special case of BSP tree. Published by Bentley in 1975 [26], they are used for organising elements in a k-dimensional space. They are still widely used today, often to provide approximate k-nearest neighbour search [84, 79], which has similarities to FRNN search. However, k-nearest neighbour search is usually applied to machine learning which often operates in much higher dimensionalities than required of complex system simulations.

Every non-leaf node acts as an axis-aligned hyperplane, splitting all elements below the node on the chosen axis from those above the node. The layers of a k-d tree must cycle through the axis in a continuous order (e.g. x-axis, y-axis, z-axis, x-axis..), such that all nodes at a specific height partition the same axis. Depending on the implementation, elements may be stored in all nodes or only leaf nodes.

The structure of k-d trees makes them efficient for performing nearest-neighbour searches (which provide a single result, as opposed to FRNN searches) in low dimensional spaces, such as collision calculations. In high dimensional spaces ($n < 2^k$ where n is the number of elements & k the number of dimensions) most elements will be evaluated during a search.

Quadtrees are tree data structures in which every parent node has 4 children, most often used for partitioning 2 dimensional space. They were first defined in 1974 by Finkel & Bentley, as a means for storing information to be retrieved using a composite key [51]. Each leaf node holds a bucket, when a bucket reaches capacity it splits, creating 4 children.

The most recognisable form of quadtree is that of a region quadtree (Figure 2.6), whereby partitioning creates 4 nodes of equal volume. Region quadtrees are often used for representing variable resolution data. The point quadtree is used to represent 2 dimensional point data and requires that a point lie at the center of each subdivision.

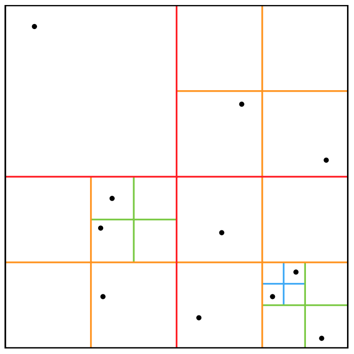


Figure 2.6: A visual representation of region quadtree. The black outline shows the root node, whose child nodes have a red outline, this continues through orange, green and blue.

Octrees are a 3 dimensional analogue of quadtrees, first described by Meagher in 1980 [110], although the initial quadtree paper [51] had stated the ease in which quadtrees could be scaled to higher dimensions. Every parent node within an octree has 8 children, allowing them to be used for partitioning 3 dimensional space. Octrees are different from k-d trees in that an octree splits about a point whereas a k-d tree splits about a dimension. Octrees are often used for volume rendering and ray tracing.

R trees are a balanced tree data structure used in the storage of multi-dimensional data. They were first proposed by Guttman in 1984 as a dynamic structure for spatial searches [63]. First proposed in 1984, it has formed the basis for many variants [144, 24, 20, 130, 125] which have since been applied to a variety of problems, such as to providing k-nearest neighbour search [82] which relates closely to FRNN search. The strategy behind R trees is to group nearby objects under their minimum bounding rectangle (hence the R, in R tree). Data structures which partition based on object proximity are known as Bounding Volume Hierarchy (BVH) structures. As all objects are contained within bounding rectangles, any query that does not intersect the rectangle cannot intersect any of the contained objects.

R trees are implemented such that all leaf-nodes reside at the same height (similar to B trees used in databases), which allows data to be organised into pages, allowing easy storage to disk and fast sequential access. Additional guarantees are provided as to the occupancy of leaf nodes, which improves query performance by guaranteeing leaves are at least 30-40% occupied. However, challenges are introduced in trying to maintain balance whilst reducing the empty space covered by rectangles and inter-rectangle overlap.

There are several variants of the R tree which are designed to improve certain qualities at the cost of others. R+ trees [144] are a variant that permit elements to be stored in multiple leaf nodes, whilst preventing none leaf nodes from overlapping and providing no guarantees to the occupancy of leaf nodes. R* trees [24] use an insertion algorithm that attempts to reinsert certain items when an insertion causes a node to overflow and an alternate node split algorithm which improves queries at the cost of slower construction. Priority R trees [20] are a hybrid between the R tree and k-d tree, providing a guarantee of worst case performance. However, this comes at the cost of stricter rules that must be enforced during construction. M trees [130], whilst not directly related to R trees, can be visualised as an R tree whereby rectangles have been replaced with circles.

Techniques for Balancing

Balanced trees are trees where all leaf nodes are at the same height. When data is only stored in leaf nodes, this enables queries to occur in constant time. A tree in its most unbalanced state is effectively a linked-list, requiring a worst case of every node being considered during queries. As such it is important to ensure that trees are created and remain balanced, or as quasi-balanced as feasible.

Self-balancing binary trees, such as Adelson-Velsky & Landis (AVL) trees [12] and red-black trees [59], operate by storing additional balancing information at each node. On insertion, a new node surveys the balancing information of its lineage. From this it is able to determine its own balancing information and update a parent's if necessary. When another node's balancing information is changed, its child subtrees may need to be rotated.

A rotation of performs a static transformation to the subtree's connectivity, to create a quasi-balanced state.

Spatial trees most often use algorithms at construction to ensure balance, these differ based on both the requirements of the particular tree and distribution of data. For example the algorithms of Sort-Tile Recursive (STR) [100] and Hilbert sort [89] can both be used for construction of R-trees, however STR should be used for uniformly distributed data, whereas Hilbert sort performs best on highly skewed data. However, as the scale of the region being queried increases the difference to performance reduces.

Balancing has the effect of increasing the cost of insertion/deletion and/or construction, to provide significantly faster queries. In most applications of FRNN search within complex system simulations, each actor inserts a single message and performs a single query, such that an expensive construction may be unsuitable for the relative quantity of queries.

Summary

This section has shown that trees make up a wide array of data structures, many of which are useful for handling different forms of spatial data. There are techniques used by binary trees that permit the maintenance of balance during inserts, however, when the branching factor increases due to multi-dimensional data this becomes infeasible.

Of these spatial trees, k-d trees and the family of R-trees appears most appropriate for delivering FRNN search, as evidenced by their use in providing k-nearest neighbour search. k-d trees appear most widely used for applications similar to FRNN search, however operating approximate methods in high dimensional space, which does not align with the requirements of complex system simulations. R-trees on the other hand are most often used with large spatial databases which are too large for available memory, whilst this has parallels with distributed workloads it may limit suitability to FRNN search in complex system simulations. In general for trees in this section, whilst able to provide FRNN search, the cost of balancing, depth of the leaf nodes and parallel suitability may all be prohibitive for use within complex system simulations.

Section 2.4 expands on this by exploring the application of trees to parallel architectures.

2.2.3 Kinetic Data Structures

Independent of the previously discussed data agnostic data structures, there exist kinetic data structures as coined by Basch and Guibas [23, 60]. These data structures seek to take advantage of the knowledge that entities in physical space often move along continuous trajectories. Kinetic data structures intend to exploit this knowledge to improve efficiency, whilst still allowing entities to change direction without an expensive update.

Within their research, they applied these data structures to numerous spatial problems including Convex hulls[1], Closest pair and Minimum spanning trees[2]. Whilst of their own admission not all solutions are efficient, they demonstrate the theoretical complexity of their techniques when applied to variations of the closest pair problem with moving points.

Research around kinetic data structures has continued, with Rahmati and Chan demonstrating further improvements for the closest pair problem[134]. Their technique acts by maintaining a set of clusters of the moving points. Their clustering follows the technique of Hershberger [76], which clusters in strips perpendicular to the x axis.

In 1 dimension, it can be thought as maintaining the leftmost and rightmost members of each set via a kinetic tournament (a priority queue based on trajectories). When two sets collide, their outermost points are exchanged. If a set becomes too wide (the separation between the outermost points is greater than 1), its rightmost point is separated into a singleton set[3]. Similar rules apply for merging sets, insertion and deletion, however, in order to maintain efficiency, more complex rules must be applied.

Of note within Hershberger's research is that to utilise their data structure in 2 dimensions points must be projected into the two axes separately. They clarify that due to the complexity of the rules required by their 1-dimensional solution a significant breakthrough would be required to apply their technique directly to a two dimensional scheme.

The research surrounding kinetic data structures appears limited to the theoretical domain, with papers highlighting favourable complexities via O notation, and no evidence of implementation or performance metrics.

2.2.4 Summary

This section has explored techniques capable for handling spatial data. These can be divided into hashing, trees and kinetic data structures. Whilst kinetic data structures appear limited to the theoretical domain, hashing and trees have been used for decades to store spatial data.

Due to the diverse range of applications for spatial data, not all data structures covered are appropriate for providing FRNN search within complex systems simulations. However, this section has highlighted relevant techniques such as perfect hashing which can be used to decompose a continuous space into a coherent key space. Similarly, trees have been shown to provide a good representation of locality, with k-d trees and R trees capable of delivering FRNN search.

A key difference between hashing data structures is how directly data can be accessed. Hashing often provides near direct access to items, whereas trees require traversal which may require iterating over many branches. Considering, each technique scaled up to many

[1]The smallest convex set (of points) which contains the entire set.

[2]A tree, without cycles, which is a subset of the edges of a graph and connects all vertices together for the minimum possible total edge weight.

[3]A set with exactly one element, also known as unit set.

FRNN queries being performed in parallel, the many additional memory accesses required to traverse a tree appear may act as a handicap.

Section 2.4 expands on this section by exploring how these and other data structures are used in parallel and in the provision of complex system simulations.

2.3 High Performance Computation

The previous section's spatial data structures, whilst optimised for performance in single threaded execution, are still often unsuitable for operating at scales involving potentially millions of actors. In order to scale to support millions of actors concurrently, complex system simulations require higher performance than can be provided by a single thread of execution. This section discusses the hardware methods used to achieve the high performance necessary to execute simulations at scale and considerations which impact code development for parallel and distributed architectures.

Due to the thermal limitations which have largely stalled increases to the speed of processors (Moore's law), it has become necessary to utilise parallel forms of computation to improve the throughput of algorithms which demand high performance. Parallel algorithms can be decomposed by a programmer in one of two styles: task-parallel or data-parallel. Task-parallelism decomposes a problem into threads which operate independently, often on distinct tasks, communicating when necessary. Comparatively, data-parallelism decomposes a problem according to the data set being operated on, such that each thread performs the same algorithm on a different block of data. Task-parallelism is most often found in the form of multi-core CPUs and computing clusters. Data-parallelism is most often associated with General Purpose computation on Graphics Processing Units (GPGPU) computation due to the dedicated architecture found in GPUs. Modern CPU and GPU processors also take advantage of pipeline parallelism, a specialised form of task parallelism, where concurrent independent instructions are executed to reduce idle time.

The following subsections discuss some of the techniques used to provide HPC. Greater focus is applied to GPGPU acceleration as it corresponds to the hardware most appropriate to complex system simulations.

2.3.1 General Purpose computation on Graphics Processing Units

GPGPU has been greatly popularised in the past decade, moving from programmers repurposing the rendering pipeline to dedicated languages, frameworks and specialised hardware options. This has led to the power of GPU computation being leveraged to speed up processing within many research domains. The more energy efficient architecture of GPUs has made GPGPU a more affordable alternative to the more traditional HPC offerings [126]. GPUs are increasingly used within the world's most powerful super computers as a single GPU can often provide a significant speed-up to otherwise CPU limited code [11]. The performance increase from multi-core CPU to GPU varies widely between algorithms.

Some research suggests 100-fold increases can be achieved, whereas Intel's own research insists if code is correctly optimised for multi-core CPUs, this falls to a maximum of 5-fold benefit. This can be attributed to memory bandwidth for problems which when optimised are found to be memory bound. There is certainly potential for the lesser awareness of and greater difficulty in applying CPU optimisation techniques having an effect on the divide [97].

It is possible to leverage multiple GPUs simultaneously to further increase performance. NVIDIA GPUs have NVLink and NVSwitch functionality which allows multiple GPUs to work in parallel. With this technology, devices are able to communicate directly via the high bandwidth interconnect, essentially allowing multiple GPUs to act as a single device. Without this technology the programmer is required to pass data between devices via the slower PCI Express (PCIe). This can be achieved using Message Passing Interface (MPI), a standardised protocol for message passing during parallel computation [6]. There are many implementations of MPI available and they can be used from passing messages between individual GPUs in the same machine to passing messages between many machines over the internet. Earlier GPU interconnect technologies used to improve the performance of graphical rendering, NVIDIA's Scalable Link Interface (SLI) and AMD's Crossfire functionality, have not been used for GPGPU computation.

GPUs do not have direct access to the memory of the host system, although programmatically the combined host and device memory are able to be represented using a single unified address space. Transferring data between the host system and a GPU occurs over the PCIe bus and incurs latency and throughput limitations. Volkov and Demmel found these limitations to be a latency of $15\mu s$ and throughput of $3.3GB/s$ [161]. However, their research targeted PCIe 1.1 hardware, whereas current GPUs use PCIe 3.0 which has greatly improved throughput. This latency is 150 times the documented latency of CPU memory accesses [101], therefore it is necessary to ensure that enough processing is occurring on the GPU to offset any costs incurred by this bottleneck. A version of NVIDIA's NVLink technology, available in high end enterprise solutions, can transfer data between CPU and GPU at throughputs 5-12 times faster than those offered by PCIe, thus reducing this gap [4].

The following subsections provide an understanding of the underlying architecture found in GPU hardware, frameworks used to utilise this hardware and a case study of the performance gains achieved with GPGPU computation on several algorithms.

GPU Architecture

In a broad sense each GPU consists of several Stream Multiprocessors (SMs) and multiple GBs of global memory. Each SM contains multiple schedulers and caches, where the schedulers actually issue the instructions to perform the execution. However, the Single Instruction Multiple Threads (SIMT) execution model is used, so each instruction is executed for a group of multiple threads, referred to as a warp. If one thread within a warp needs to branch, then every thread must branch, and any that are not required to follow this code path remain idle until the branch returns (see Figure 2.7). This general architecture is the same for all GPUs. The remainder of this section provides specific architecture details of NVIDIA GPUs for completeness.

Each NVIDIA GPU consists of 1-72 SMs and several GBs of global memory (DRAM). There are also several (significantly smaller KB scale) caches within each SM. These caches are used for: accessing read-only texture memory; reading global memory; storing data within constant, local and shared memory.

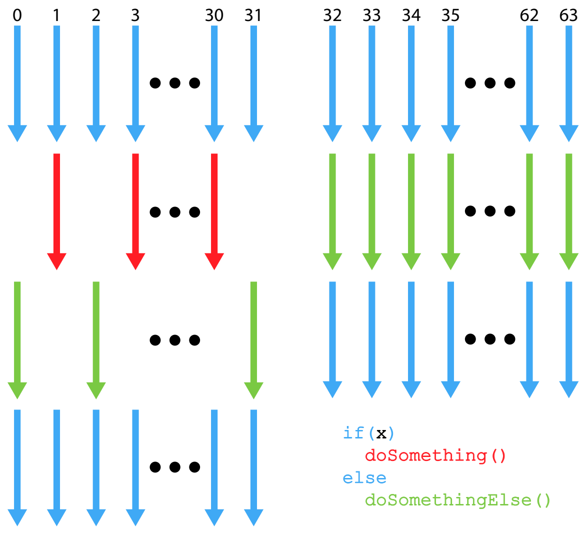


Figure 2.7: Illustration of the SIMT parallel execution model used by GPUs. The threads within the first warp (threads 0-31) contain multiple values of x, therefore both branches must be executed separately. All threads within the second warp (threads 32-63) have the value $false$ for x, so only the green branch is executed.

When using CUDA, a function to be executed on the device is called a kernel. When a kernel is launched, the programmer must specify how many threads to launch, which controls how many instances of the code within the kernel are executed. The number of threads is defined by providing grid and block dimensions. Grids and blocks are both 1-3 dimensional structures, where each index within a grid is a block and each index within a block is a thread. At runtime blocks are assigned to SMs, and are then further partitioned into warps of 32 threads that simultaneously execute. If less than 32 threads are required the resources for 32 threads are still used but the excess threads are masked from execution, remaining idle. Blocks must be capable of executing independently, as they may execute in any order across multiple SMs.

Since the introduction of the Kepler architecture (2012) [1] each SM has consisted of four schedulers capable of issuing dual instructions. This feature allows a scheduler to issue two consecutive instructions to a warp simultaneously if the instructions do not interact. Dual instructions cannot be dispatched for two different warps simultaneously. Each scheduler is capable of managing several warps (from different kernels) concurrently and instructions can be dispatched for a queued warp while the previous warp waits. This allows latencies to be hidden if enough warps are managed concurrently. The number of consecutive warps that can be managed by each scheduler is limited by the available resources required by each warp (e.g. registers or shared memory). There is also a device specific hardware limitation of the maximum threads per SM. Current devices have this limit at 2048, which divides (32 threads and 4 schedulers) to 16 warps per scheduler [114].

Each executing thread is allocated up to 255 32-bit registers. However, each SM only has 64,000 registers available [118, 122][4]. Therefore, as the number of warps each SM manages increases, the available registers per thread decreases. If a thread requires more local memory than is available within the registers, memory accesses will spill from the registers into the L1 cache, increasing the latency of memory accesses. The size of the L1 cache varies between architecture and configuration from 16-128KB. If the L1 cache is filled, memory accesses subsequently spill into the L2 cache further increasing the latency of memory accesses. The L2 cache is shared by all SMs and has a size from 256-6144KB. Furthermore, should the L2 cache become filled, memory accesses will be deferred to the global memory. Global memory has a latency 100x that of the L1 cache, therefore large quantities of register spilling can become a significant overhead.

Whilst a thread is executing it also has access to both global memory and texture memory. Global memory accesses are always cached inside the L2 cache. When accessing large amounts of read-only data, utilisation of the texture cache can provide faster memory reads. Devices prior to CUDA compute 3.5 are required to manually copy data to texture memory to make use of this cache, whereas 3.5 and later devices will automatically make use of this feature, instead referring to it as the read-only cache, where the compiler detects its suitability. Use of `const` and `__restrict__` qualifiers will assist in compiler detection, and `__ldg()` can be used to force data load via the read-only cache. Architectures since Pascal have combined the texture cache with the L1 cache, reducing the significance of texture memory in many cases.

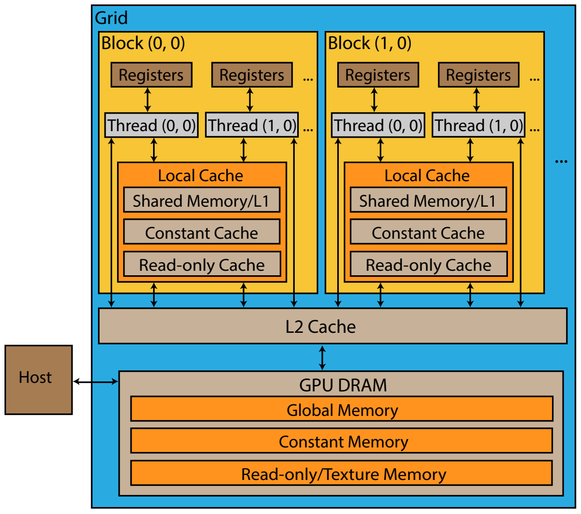


Figure 2.8: Illustration of the memory hierarchy present within the Kepler GPU architecture. The newer Maxwell architecture has moved the L1 cache so that it is shared with the read-only cache.

There are two additional types of memory available to threads: constant and shared memory. However, they both require special consideration. The constant cache can only process one request at a time. Memory accesses to the same address are combined, therefore the effective throughput is divided by the number of separate requests. The constant cache is 64KB in size, and cache misses must be loaded from device memory.

Shared memory on the other-hand is on-chip memory that is shared between all threads within a warp. This memory has higher bandwidth and lower latency than both local and global memory. Shared memory is, however, divided into equally sized memory banks. Memory banks are organised such that consecutive 32-bit words are assigned to separate

[4]Per SM register capacity is currently the same for all GPU architectures (Kepler through Turing).

banks. Each memory bank can only service a single 32-bit word per request, therefore if multiple threads attempt to access different addresses within the same bank, accesses to each address will be serialised. Each SM has access to 16-64KB of shared memory, depending on architecture and configuration.

Each SM also contains a mixture of Single Precision (fp32) units (often referred to as CUDA cores or stream processors) and Double Precision (fp64) units. These units are capable of performing 32bit and 64bit arithmetic, respectively. Special Functions Unit (SFU)s are able to perform 'fast math' operations on fp32 values[5]. The quantities of these three units varies between architecture, chip and product line. For example, the workstation and HPC product lines (Quadro and Tesla respectively) generally contain Error Correction Code (ECC) memory and have a higher ratio of fp64 units to fp32 units than the domestic counterpart (GeForce). These quantities affect the operations that each device excels at.

Additionally, as the price for a GPU increases, the memory bus width usually increases too. The memory bus width refers to the number of bits of global memory that are read per transaction. Therefore, if data requested by a thread is a contiguous block of 128 bits, it will be read in one transaction by a memory bus of 128 bits width, rather than the multiple transactions required if the data were scattered or the memory bus smaller. Use of read-only caches can reduce the influence of this on scattered reads, unnecessary data is cached when read, and later instructions that request cached addresses then do not incur the latency of a global memory read transaction.

Additional Fixed Function Features of GPUs

As GPU technology inches closer to the limits of Moore's Law, which predicted regular increases in transistor density on processors, architectural trends have started to move towards task specialisation. New GPU architectures, released by NVIDIA, increasingly target the demands of common GPU applications. Despite this specialisation, these features are also often made available through CUDA, allowing developers to find ways to apply this specialisation to speedup applications from unrelated domains.

Tensor cores, available since the release of the NVIDIA's Volta architecture in late 2017, are engineered to accelerate the training and inference of neural networks [120]. They were designed in response to the explosive growth in research surrounding deep learning, engineered specifically to execute the 4x4 matrix calculations central to both the training and inference of neural networks. The specialisation of Tensor cores to training and inference has delivered peak performance improvements of 12x and 6x respectively over the previous Pascal architecture. The more recent T4 Turing Tensor Cores, with greater support for multi-precision calculations required by inference have enabled upto

[5]CUDA provides several low precision intrinsic functions such as `__sinf()` and `__logf()` that execute in significantly less instructions than their more precise and versatile counterparts.

32x inference calculation throughout over the Pascal when working with lower precision data [121].

Following the introduction of Tensor cores, CUDA 9 provided access to the special functionality of Tensor cores to perform matrix multiplication. The matrix multiplication is available both at the intended matrix size (4x4) and for arbitrary sized matrices, enabling application to fields such as linear algebra. By instead utilising the tensor cores for mixed precision calculations, performance can be improved by 6x and 3x over single and half precision respectively when utilising previous methods on the same hardware [109]. The cost of mixed precision however is an error rate which increases with the size of input matrix, due to the rounding from single to half precision. Techniques have been proposed to mitigate these precision errors, so it is expected that HPC developers will increasingly look to take advantage of Tensor cores in their applications.

RT cores, first available in July 2019 with the introduction of NVIDIA's Turing architecture, were developed to enable faster and higher fidelity graphical realism in commercial rendering and videogames. In combination with machine learning noise reduction, RT cores have enabled real-time ray-traced graphics on domestic GPUs. RT cores have also enabled significant speed improvements for commercial graphical applications, by calculating 10^9 rays per second, allowing scenes to be rendered 10x faster than when using prior GPU hardware [121]. The technology behind RT cores is highly specialised for the task of bounding-volume hierarchy traversal, which is central to the process of ray-tracing [121]. Whilst bounding-volume hierarchies such as Octrees are highly applicable to spatial tasks such as FRNN, at the time of writing, September 2019, RT cores have not yet been made accessible to CUDA developers.

Optimisation

In addition to obvious considerations of the architecture introduced above, there are further techniques used to improve the performance of GPGPU applications. With the primary application of GPGPU being HPC, optimisation guidance is far more prevalent than for CPU optimisation, although in many cases GPU optimisation techniques are relevant to CPU optimisation.

Traditionally in CUDA, occupancy refers to the number of active warps divided by the maximum number of warps. Ensuring maximal device occupation, so that the hardware within the architecture is utilised as much as theoretically possible, is a primary target of optimisation. This can also be referred to as thread-level parallelism. Occupancy is bounded by the limit of 2048 threads per SM and the capacity of registers and caches. Providing SMs with multiple warps or instructions to execute simultaneously helps hide memory latencies, as other instructions can be executed whilst waiting for memory.

Optimal grid and block dimensions can increase occupancy via thread-level parallelism and reduce execution times in some scenarios by 23% [154]. Whilst it is theoretically possible to calculate the best grid and block dimensions for a kernel when applied to a specific device, in most cases that can be avoided. It is often far simpler to simply benchmark the kernel against a large variety of dimensions to find the the best performing combination, which better accounts for missed variables that may affect occupancy. When purely trying to increase occupancy, register spilling is quite a realistic risk — typically occupancy above 50% does not translate to increased speed-up. CUDA 6.5 introduced functionality for automatically optimising block dimensions for occupancy [67], however, these are not a complete solution to optimisation as they do not account for kernels instead bounded by memory accesses. Using the CUDA compiler (NVIDIA CUDA Compiler (nvcc)) option `-Xptxas -v, -abi=no` will print the number of bytes of local memory used by each kernel, providing better information for manually partitioning kernels. The CUDA profiler also contains several counters that can aide in identifying register spilling [113].

Algorithm 1 Pseudo-code for a vector addition kernel without instruction level parallelism.

```
addition_kernel(a,b,c)
    i = thread.id
    c[i] = a[i] + b[i]
```

Algorithm 2 Pseudo-code for a vector addition kernel that has been optimised with x2 instruction level parallelism.

```
addition_kernel_ilp_x2(a,b,c)
    //Read
    i = thread.id
    ai = a[i]
    bi = b[i]
    j = i + (grid.width * block.width)
    aj = a[j]
    bj = b[j]
    //Compute
    ci = ai + bi
    cj = aj + bj
    //Write
    c[i] = ci
    c[j] = cj
```

Instruction-level parallelism provides an alternate approach for optimising memory access bounded algorithms — increasing performance whilst reducing shared memory traffic and the number of active warps. Instruction-level parallelism seeks to decrease occupancy by performing enough work within each thread to hide memory latencies without SMs switching active warps. Additionally, the increased utilisation of registers instead of shared memory allows the avoidance of bank conflicts, which occur when accesses to shared memory are misaligned and result in serialised accesses.

In essence, instructions are separated and ordered within a kernel, so that operations are not performed until required data is in local memory. This allows the device to avoid waiting for memory returns or the minor overhead of rotating between warps. The pseudo-code in Algorithms 1 and 2 provides an example of how x2 instruction-level parallelism may be performed, although in practise using x4 or x8 (or even as high as register capacity permits) is likely to provide even greater results, due to the number of clock cycles required to perform accesses. In an ideal scenario instruction-level parallelism was able to achieve 87% memory peak when copying 56 floats per thread. This was higher than the 71% achieved by the core CUDA function `cudaMemcpy`. [159]

As GPUs follow the SIMT execution model, the hardware is optimised towards static branch prediction where each thread within a warp should perform the same operation (CPUs have more advanced branch prediction). To aid branch prediction, the structure of code can be simplified by unrolling loops so that multiple iterations are combined into a single iteration. Unrolling a loop can reduce branching penalties, allowing improvements to performance. Volkov identified that a mutated form of loop unrolling can be used to additionally provide the benefits of instruction level parallelism [160].

As stated earlier, the memory bus width is the number of contiguous bits accessed per read/write operation to DRAM. For this reason it becomes necessary to lay out large collections of data as a Struct of Arrays (SoA) rather than an Array of Structs (AoS). This layout ensures that neighbouring threads, accessing the same member of a struct specific to the thread, are accessing contiguously stored data. Reading/writing data in this format is considered a coalesced read/write operation and the resulting global memory bandwidth occupancy is maximal. CUDA compute 3.0 and higher devices are slightly less affected by this optimisation, due to more advanced striding, however, it is still a worthwhile technique due to it having similar relevance to CPU optimisation.

Frameworks for GPGPU Development

There exist two distinct frameworks for GPGPU application development, OpenCL and CUDA. In terms of performance, both frameworks are capable of achieving similar benchmarks once tuned to the targeted device, as the code produced by each framework is very similar at a device level before compiler optimisations [90, 47]. Additionally, OpenGL and DirectX both provide functionality within their graphics APIs for compute shaders. However, there is not much evidence of widespread use of compute shaders past 2010, likely due to the rise of the dedicated frameworks OpenCL and CUDA. An optimised OpenGL (compute shader) scan operation was found to execute 7 times slower than an optimised CUDA implementation. This difference was attributed to CUDA compiler optimisations not available to compute shaders [68]. Compute shaders may receive a resurgence with the release of Vulkan, as this aims to introduce lower-level controls into the graphics APIs, benefiting GPGPU by allowing better integration with graphical outputs.

OpenCL is an open standard maintained by the Khronos Group, the same consortium that manages the OpenGL standard, capable of targeting any parallel architecture that has appropriate vendor provided drivers. At this point in time OpenCL supports: both AMD and NVIDIA GPUs; Intel CPUs and integrated GPUs; several IBM products.

CUDA by comparison is a proprietary offering, restricted to NVIDIA devices. CUDAs's primary advantage has been in the greater level of libraries to assist computation and use of hardware features unique to NVIDIA GPUs. It is also easier to optimise between devices, due to the architecture uniformity among NVIDIA devices, whereas OpenCL supports a diverse array of hardware architectures, each requiring different optimisation priorities.

Alternatively, OpenACC is a compiler directive standard for accelerating serial code through the use of GPGPU [7]. Similar to OpenMP, programmers can label their code with simple compiler directives which are then processed at compile time to include the necessary GPGPU code. Directive standards provide a simple technique for accelerating existing code, however, directives used incorrectly can provide suboptimal and potentially slower results.

Case Study of GPU Optimisation

Grauer-Gray et al [57] used GPU processing to accelerate four computational finance applications – Black Scholes, Monte-Carlo, Bonds & Repo – from the library QuantLib. They compared applications speed when: executed serially; executed in parallel on an 8-core CPU; executed on a K20 GPU when rewritten with CUDA and OpenCL; executed from generated GPU code from the directive based accelerators HMPP and OpenACC. Their results showed that the GPU programs executed at over 100 times that of the sequential CPU and 10 times the parallel CPU code for the Black Scholes algorithm. The speeds of the GPU code varied between sources, although there is nothing to suggest each could not be rewritten to achieve similar speed. For the Monte-Carlo program, they found, for programs actioning over 5000 samples, there was again a speed-up of at least 75 times sequential CPU code and 10 times parallel CPU code. This increased to 1000 times speed-up against sequential CPU code when compared for processing 50,000 samples. Once 500,000 samples was reached, however, the speed suddenly dropped off. Profiling identified the cause of this was due to cache misses from the random number generation. Restructuring code to make better use of the cache would likely be capable of alleviating this fall off. The bonds application only managed an 80 times speed-up compared with sequential CPU code, which is likely due to the amount of divergent branching within the algorithm.

2.3.2 Cluster, Grid & Distributed Computing

A computer cluster is a local collection of connected computers that work together as a single system. Clusters are often deployed as they can provide improved performance, availability and cost over a singular machine of similar specification. A beowulf cluster specifically refers to a cluster assembled from consumer grade computers.

Grid and distributed computing are the techniques where large groups of computers at different locations (potentially including clusters) work in parallel on the same computational projects. In the case of grid computing, the groups of computers are often owned by various research groups, who each share access with others. In the case of distributed computing, public engagement is often leveraged, encouraging people to install the necessary software on their computers (and even phones), so that idle processing power can benefit research in fields such as medical simulations (Folding & Home) and SETI [16]. Distributed computing marketing often neglects to inform users that use of their ‘spare’ processing power will increase power usage.

When using computer clusters, it is often the case that due to the cost of the architecture such a cluster is shared between several research departments or universities. Similar to grid computing this is likely to lead to a schedule system, whereby jobs must be queued, removing potential for real-time applications.

Distributing applications across multiple machines introduces much higher latencies for sharing of data. Whilst CPU and GPU cache times are measured in terms of nanoseconds (10^{-9}), access across network interfaces is measured in milliseconds (10^{-3}), so latencies are in the order of 1 million times higher. Whilst distributed systems are often designed to minimise this latency, large parts of it simply cannot be reduced. This has a significant negative impact on the performance of algorithms whereby the division of a problem requires frequent live data to be shared across boundaries.

To maximise performance of FRNN search in an environment with a uniform distribution, and hence frequent sharing in any division, it is necessary to limit the impact of these latencies, such that a single device approach should be used as far as viable. Modern GPUs are capable of hosting millions of concurrent threads and multiple GPUs can be used together in the same machine as an interim step.

2.3.3 Many Integrated Cores

Similar to GPU compute, Intel offered Many Integrated Core (MIC) architecture supercomputing cards (Xeon Phi), which were intended as a direct competitor to GPGPU, consisting of up to 72 cores capable of executing the x86 instruction set. The potential performance difference between MIC and GPU is negligible, with each performing best in the algorithms their architectures best suit, with GPU providing double the bandwidth of MIC for random data access [152]. After the debut of MIC in 2011, most of the product line has been discontinued as of 2018 due to limited interest.

2.3.4 Architectural Specialisation Away From GPUs

Similar to NVIDIA's movements towards task specific chips within their GPUs, interest in Field Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs) are integrated circuits, designed or programmed for highly specific tasks. The specialisation of such hardware can provide task specific high performance with lower energy requirements than more general hardware.

Whilst FPGAs have been considered for use in HPC and accelerating complex system simulations, mostly predating the growth in GPGPU development, the desire to accelerate training of neural networks saw a reemergence of interest in applying the technology to HPC workloads [74, 14, 164].

Companies such as Google, Amazon and Facebook, with greater financial resources, have taken to designing their own ASICs for neural networks in partnership with existing semiconductor manufacturers. Furthermore, seeking to improve both speed and energy consumption, Bitcoin miners have largely switched to bespoke ASICs developed by companies such as Bitmain and Bitewei [151, 70].

Google's 'Tensor processing unit' ASIC, developed specifically for Google's TensorFlow machine learning framework, have been made available commercially to purchase or use via cloud computing services. Testing of first generation Tensor processing units showed their inference performance to be more than 15x that of NVIDIA's K80 GPU, with 30x the operations per Watt (energy efficiency)[87].

It currently remains unclear how competitive the latest iteration of Tensor processing unit is versus NVIDIA's similarly specialised Volta and Turing architectures. However, with both architectures initially releasing around 2017, the use of specialised hardware for machine learning can still be considered in its infancy.

Due to the specialisation of ASICs, currently they are unlikely to fit the requirements of most MAS, however it is possible wider adoption of ASICs for machine learning workloads will encourage other investors in HPC to invest in task specific hardware, increasing the chances of specificity relevant to MAS.

2.3.5 Summary

This section has shown that among the HPC offerings, GPUs provide an accessible and alternative to traditional HPC systems, capable of providing up to 10x speed-up vs parallel CPU algorithms. Growth in task specific hardware, largely driven by machine learning, may eventually benefit other fields. However, the current investment required to develop and manufacture high performance ASICs suggests it is unlikely that they will be developed to target complex system simulations.

Due to the high level of communication found within complex system simulations and during FRNN search, it is pertinent to avoid introducing unnecessary latency through the chosen HPC architecture. Therefore models should be confined to a single GPU as far as possible. Once model scales exceed a single GPU, multi-GPU implementations should be used, restricting the highest latencies, from a distributed architecture, as a last resort should model scales require it. However, the ability for a single GPU to handle upwards of 10 million threads provides a large buffer space.

Optimisation is important when developing HPC applications, of which there are many optimisation techniques and architecture features to consider when writing algorithms for GPUs. These techniques range from interlacing instructions to ensuring memory is accessed appropriately, helping maximise utilisation of the available hardware.

2.4 Parallel Spatial Data Structures

With an understanding of how HPC utilises parallel algorithms, it is possible to discuss the techniques used to apply spatial data to parallel specific nuances such as the added requirements of conflict resolution and synchronisation.

The most obvious of these nuances is that of arbitration when concurrent threads wish to access the same resource. In multi-threaded CPU systems this is usually synchronised via locking protocols utilising MUTEX, whereby a thread must wait to acquire a lock to a resource before accessing it. Other solutions make use of atomic operations, whereby a limited set of thread-safe operations are serialised to avoid potential race conditions. In addition to these hardware feature, algorithms can often be restructured to avoid race conditions on write operations through increased communication. A potentially important feature of restructuring algorithms in this manner, is the avoidance of stochastic operations which can produce differences in results across repeated executions.

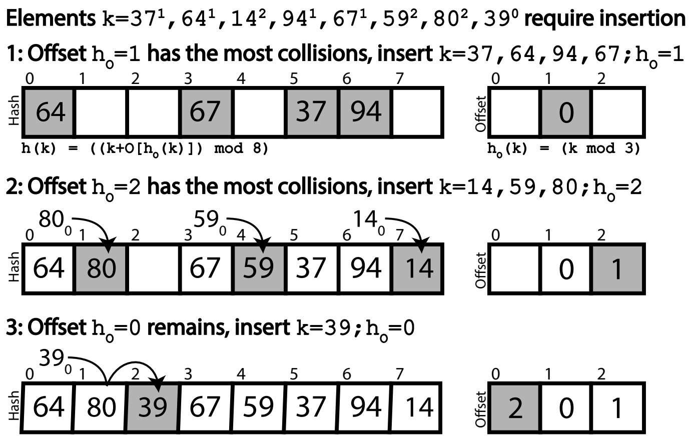
The following subsections discuss some of these existing spatial data structures which have been implemented for parallel architectures. Whilst these data structures are not all suitable for providing the FRNN searches that this thesis is concerned with, the underlying techniques that have been used for handling these parallel specific nuances may prove relevant.

2.4.1 Hashing

Hashing structures are a powerful data structure for memory-bounded algorithms, as their almost direct access to elements can provide an advantage over many ordered structures. Below are details of techniques that have been used and how they handle arbitration.

Perfect Spatial Hashing

Lefebvre & Hoppe presented a scheme for generating perfect spatial hashes of point-data, capable of maintaining spatial coherence [98]. Their technique combines two imperfect hash functions and an offset table to provide queries in exactly two memory accesses. The offset table is compacted by the need to only store a single byte for 15-25% of the number of elements to be stored. The use of a perfect hash however makes the construction of the table significantly more computationally expensive. Figure 2.9 demonstrates the construction algorithm which is explained below.



Elements k=37¹,64¹,14²,94¹,67¹,59²,80²,39⁰ require insertion
1: Offset h_o=1 has the most collisions, insert k=37,64,94,67;h_o=1
Hash
Offset
h(k) = ((k+O[h_o(k)]) mod 8)
h_o(k) = (k mod 3)
2: Offset h_o=2 has the most collisions, insert k=14,59,80;h_o=2
3: Offset h_o=0 remains, insert k=39;h_o=0

Figure 2.9: A visual representation of a perfect hash table that utilises linear probing and a secondary offset table. The element 37^1 denotes that 37 has the offset hash 1.

The algorithm for constructing the table first calculates how many of the items hash to the same location within the offset table. The 8-bit offsets are then generated, starting from the most common offset, placing elements into the hash-table. They found this to work appropriately with suitably sized offset tables, however, they noted that backtracking could be used were the algorithm to reach a stalemate state.

In order to maintain coherence within the main hash-table, when selecting offset values, the most coherent offset is chosen. To accomplish this, the inputs which share the same offset table hash are temporarily mapped into the main hash-table to identify how many of their spatial neighbours would be their neighbours within memory at each offset. This algorithm is simplified with two heuristic rules: (i) the offset values of neighbouring entries in the offset-table are first attempted (due to the coherence of the offset-table); (ii) following this free neighbouring slots to spatial neighbours that have already been placed into the hash-table are attempted.

Their data structure is compact and utilises a low number of memory accesses when performing queries. However the construction times for their hash-tables are costly. Using GPU hardware (from 2006), their algorithm was capable of constructing a fast table of 11,000 elements in 0.2 seconds, however, the optimised table takes 0.9 seconds. Whilst this performance is likely to have improved in the decade since their research, it is unlikely to provide fast enough constructions for real-time motion at an interactive frame-rate as is required for complex system simulations. This is highlighted by their note that future work could involve extending the hashing to support efficient dynamic updates.

Uniform Spatial Partitioning

As seen in Sections 2.1.3 & 2.1.4, the use of spatial subdivision into a uniform grid of bins is a common technique used for managing spatial data in complex system simulations. The technique is known by many names: uniform subdivision [104], spatial partitioning [138], grid approach [56], spatial subdivision [85], cell list [139], cell linked list [124] and likely others. Within this thesis, (uniform) spatial partitioning is used. It has been used in complex system simulations to represent a range of spatial actors across fields such as Nano robotics, Crowd dynamics, SPH and Molecular Dynamics (MD) [103, 91, 124, 55].

The spatial partitioning data structure functions by decomposing the environment into a uniform grid of bins, each given a linearly assigned id number. Spatially located entities are therefore able to identify their containing bins, allowing them to be sorted and stored in order of their bins' ids. Additionally an index must be constructed, to map each bin's id to its data within the main storage array. This storage is compact, containing no additional buffer space, ideal for the memory limitations of GPU computation. However, this therefore requires that the entire data structure be reconstructed to insert a new entity, or move an existing entity (to a different bin).

Although this data structure is not dynamic its construction primarily utilises the highly optimised parallel primitive operations sort and scan. This enables the construction algorithm to be well suited to the SIMT architecture of GPUs.

In order to access entities within a spatial area, stored within the spatial partitioning data structure, the environmental bins which contain any of this spatial area are identified, and their contained entities are then all individually tested against the search area.

Section 2.5 explores the USP data structure in greater detail, providing analysis and discussion of the existing research.

2.4.2 Parallel Trees

The advantage that trees have over hashing data structures is their provision of links for traversing between data – the order of this data is dictated by the particular tree. This comes at the cost of multiple memory accesses to locate nodes by following the tree structure, whereas hashing structures are often able to provide direct access. The sections below provide a discussion of the implementations behind these data structures.

Parallel k-d Trees

As k-d trees are primarily applied to the ray tracing process of rendering, they have been implemented many times over to utilise parallel computation as researchers strive for real-time ray-tracing of complex scenes.

Shevtsov et al have described how k-d tree construction is easily made parallel by partitioning the the input geometry between available threads and only synchronising between each stage of construction [146]. The alternative technique is that of constructing multiple sub-trees, however, this requires an initial decomposition to prevent the sub-trees from overlapping. Calculating a balanced decomposition of the input geometry is computationally intensive, whereas the alternative of partitioning space is likely to lead to poor load balancing. Their implementation was configured to utilise a quad-core processor and only calculate approximate bounding boxes. This allowed it to outperform an existing optimised single threaded accurate k-d tree construction by over 100x, constructing a k-d tree of over 1 million triangles in 0.45 seconds. Developers at Pixar have used k-d trees in parallel for the simulation of agent-based crowds, requiring it for the complex rules of inter-agent and environmental interaction [62].

Popov et al utilised a k-d tree with 'ropes' between adjacent nodes to perform efficient GPU ray tracing [133]. By having each node maintain a link to a neighbouring node from each face, the necessity of the stack as is traditionally used during ray-tracing k-d tree traversal is removed. This makes the technique more suitable for GPU computation. Once the initial node of entry has been identified, the ray tracing algorithm is now able to compare a ray trajectory against a node. If it does not intersect with any of contained geometry, the face of the k-d tree which the ray exits via is identified and the rope to the neighbouring node attached to that face is utilised till the ray collides or exits the scene. Further speedup is achieved by grouping similar rays into packets, such that rays passing through the same node can share computation. This however requires the use of a heuristic, such as surface area heuristic (SAH), to select the best node when rays exit via different faces. They compared their single and packet ray techniques against the OpenRT ray tracing library [43] and found they were able to trace at 60% and 550% higher FPS respectively.

Parallel Matrix Tree

In 2011, Andrysco & Trioche developed a dynamic tree implementation, capable of parallel construction and representing any tree with a constant branching factor and regular structure [19]. Their research was concerned with improving the representation of Octrees and k-d trees [18], whereby each layer of a tree is represented as a matrix, such that their structure is regular, removing the need for scattered allocations, allowing them to better take advantage of parallel hardware when used for ray-tracing. In essence, the bottom layer of the tree is assembled as an array, containing each of the non-empty nodes in a known order. The tree structure above is then easily computed in parallel.

The results show that it was able to construct k-d Trees for 17,000 triangles in 0.002 seconds, with 174k triangles only taking 0.008 seconds. Similarly testing showed that the implementation significantly reduced the nodes traversed during multiple ray-tracing samples, compared to the existing state of the art.

Parallel R-Trees

R-trees are primarily used with large databases of spatial data, as such early work targeted the primary bottle neck I/O. This research consisted of multiplexing the R-tree across multiple disks, by using a branching factor equal to the number of available disks, and each child node of the same parent maps to a unique disk [88].

Luo et al implemented the first comprehensive GPGPU R-tree [108]. Their experiments found the GPGPU implementation to perform constructions 20 times faster, and queries upto 30 times faster than existing sequential algorithms when working with on-chip data.

Their technique for construction was to only parallelise the two operations of sorting and packing, which are heavily used in sequential construction algorithms, whilst utilising the CPU for the remaining sequential operations. This enabled their construction to produce R-trees of identical quality to existing algorithms, avoiding a limitation of earlier parallel techniques that operate by merging smaller trees.

Their technique for parallel queries was to represent the R-tree as two linear arrays, one providing an index mapping the nodes to their parents and the other to storing rectangle information for each node. They then utilise a two tiered parallel approach to queries. Each block of threads performs a different query, and multiple blocks can be launched to perform multiple queries concurrently. Their threads within each block are able to work together performing a breadth first search, using a frontier queue to communicate [107]. However in order to take advantage of memory coalescing, each thread is responsible for an entry, rather than a node. This includes threads responsible for empty entries. Further performance was gained by storing the top levels of the R-tree within the constant cache, which reduced the execution time of queries by 10%.

Whilst their parallel query technique has high utilisation of threads when query rectangles have significant overlaps, they found it unable to fully utilise resources when queries did not have many overlaps. To remedy this they were able to encode their queues such that each block could handle multiple queries.

Parallel Quadtrees & Octrees

Burtscher and Pingali described a GPGPU implementation of the Barnes Hut n-body algorithm [30], which uses a parallel octree that holds a single body within each leaf node. The tree nodes and leaves are both stored in the same array, whereby leaves index from 0, and internal nodes index in reverse from the end of the array. They also have a single array per field (Struct of Arrays), to take advantage of memory coalescing.

They construct their octree via an iterative insert algorithm. Each thread inserting a different body attempts to lock the appropriate child pointer via an atomic operation. If the lock is gained and child pointer is empty, the body is simply added. Otherwise the node is partitioned and both the existing and new bodies are inserted. A memory fence operation is then executed to ensure other threads are made aware of the new tree structure before the lock is released. Other threads that fail to gain a lock, continue to retry until success is gained. The SIMT paradigm, heavily reduces the retries that must occur, as the thread divergence causes threads which failed to gain a lock to wait for any successful threads within a warp to complete, reducing the number of failed memory accesses. The `__syncthreads()` operation is used, to ensure that warps whereby no locks are gained do not utilise the GPU with unnecessary memory accesses. This then iterates until all bodies have been inserted.

They also detail the techniques that they use for traversing and sorting the tree, where the sorting is necessary to speed-up the neighbourhood searches when calculating the forces applied to each body. However they do note that some of their techniques are constrained by the lack of global memory caching in earlier GPUs.

In summary their complete algorithm provided around 10x speed-up over an existing CPU algorithm. As would be expected the tree construction and neighbour search kernels both consumed the most GPU time.

Jian et al extended this to produce a faster GPGPU quadtree implementation (CUDA-quadtrees) whereby the tree was contained within on-chip shared memory rather than global memory [83]. In order to construct quadtrees, each block of threads builds a quadtree for a different region, with each thread performing a different insertion. Notably they used a similar locking technique to that of Burtscher and Pingali, with the key difference that each block constructs a different quadtree in shared memory, which is copied to global memory after construction. Similarly, searching the quadtrees involves loading the related trees back into shared memory, and performing a parallel depth first search. When comparing their quadtrees against the official implementation of Burtscher and Pingali's octree, they found their implementation to perform significantly faster (100x) in their construction experiments which extended to 1,000,000 items and around 10x faster in searches.

Octrees have also been demonstrated as a more performant alternative to USP when performing the similar k-nearest neighbours algorithm [99, 15]. The non-uniform partitioning allows k neighbours to be found within skewed distributions more efficiently than uniform subdivision approaches.

It seems clear that in cases of skew a lone actor could be required to access a much larger Moore neighbourhood in order to find additional neighbours. The effect of this would be a high overhead of bin accesses before the k neighbours are found.

In contrast, the fixed radial cutoff of FRNN search ensures that the numbers of bins to be accessed is always constant. As such, it is unlikely to see the benefits of an Octree approach. Although, the Octree traversal task specific RTX cores found in some recent GPUs for ray tracing may in future provide acceleration which changes this dynamic. However, these are not currently available to utilise via CUDA.

Fast Approximate Near Neighbour Graphs

To improve performance of spatial algorithms, especially in high dimensionality space as required by AI applications, many have resorted to approximate methods. Harwood and Drummond have presented Fast Approximate Near Neighbour Graphs (FANNG) [69] for the nearest neighbour algorithm (which returns a single neighbour as opposed to FRNN search).

Their technique relies on building a graph, where edges exist such that if the graph has an edge from $p1$ to $p2$, there is no requirement for an edge from $p1$ to any vertex $p3$, that is closer to $p2$ than $p1$. The construction of this data structure requires iterating all possible edges to filter the desired ones.

They state their technique can be extended to approximate k-nearest neighbours, which is closer to the FRNN search algorithm. This operates by instead returning the list of the k closest vertices found during graph exploration (ordered via a priority queue).

The GPU performance of their implementation's recall is only compared against a linear search, with as much as a 5x speedup. However, the significance of this result is to be expected when comparing against a brute force technique. The brute force instead occurs in the pre-processing to generate their graph. As their paper does not discuss construction performance, it is fair to assume their technique is not intended for real-time reconstruction.

Dynamic Bounding Volume Hierarchies

Whilst not intended for spatial point data as with the previously covered spatial data structures, Larson and Möller utilised a dynamic data structure for use in collision detection [96]. Their data structure is a hierarchy of axis-aligned bounding boxes which enclose polygonal meshes.

Initially a single root node exists enclosing each dynamic object. Then as the simulation progresses, these trees are incrementally rebuilt to represent changes to the spatial configurations of the represented objects.

During the collision detection stage of execution, all nodes within the hierarchy which are accessed are marked as active. This allows updates to the hierarchy to explicitly target reconstructions (referred to as refits) to the subtrees containing active nodes. This refit process begins from the active leaf nodes, and works its way upwards.

During the process of refitting, the difference between the volume of a parent node and the sum of its children's volumes is used to determine whether a subtree has become invalid. Invalid subtrees are then later repartitioned to reduce bounding overlap during the collision detection query stage.

They stated among their results that their data structure's lazy refits would be challenged by unnatural motions whereby initially neighbouring primitives moved in opposite directions. They also found that its primary bottleneck was when an object was colliding with itself. It was not stated why this creates a bottleneck, however, such a state would likely cause all of the nodes within the hierarchy to be marked as active. This shows that such a structure, whilst dynamic, is better suited to cohesive movements whereby many elements have similar trajectories, and performance quickly degrades when trajectories are highly variable between elements within the same local area. Hence, this style of lazy refits may have applications to ordered behaviour, however, appears unsuitable for the chaotic behaviour found within complex system simulations.

Spatial Trees in Complex System Simulations

Dmitrenok et al performed a review of spatial trees in the simulation of biological tissue [44]. Concerned with simulations of cell behaviour using the BioDynaMo platform, which has a center-based representation of cells and intends to utilise a cloud approach to parallelism [29], they identified spatial search and reinsertion as the most expensive stage of simulation. In particular they require FRNN search for detecting collision between up to 1,000,000 cells. They focus on three styles of tree: Octrees, K-D trees and R trees.

In discussing results, they noted the trade-off of tree depth. As depth is increased search time shortens, however, insertion time (construction of the tree) increases. Primarily visible from their results is that R tree is outperformed in both search and insertion for 10,000 nodes so much that it did not warrant testing at higher capacities. Overall, however, they found that searching provided the bottleneck, with KD trees providing a total performance around 2x faster than Octrees in the configurations and search ranges tested.

Despite BioDynaMo being intended for cloud parallelisation, their experiments were limited to a basic CPU with 4 threads, so it is hard to gauge the impact that latency from the intended distribution would have on performance. Furthermore, these results are not indicative of how the trees may perform on the highly parallel architecture of GPUs.

2.4.3 Parallel Verlet Lists

Verlet lists have also been used as a potential alternative for providing FRNN search. In contrast to USP, once the FRNN search has been performed each actor stores it's list of neighbours; essentially caching the search result [157]. However, this technique relies on an additional step on top of USP and FRNN search to produce the data structure [156]. Furthermore, the primary value of Verlet lists lies in reducing the need to re-perform the neighbour list construction, only rebuilding each neighbour list when its central actor has moved a set distance. This scales cost with the frequency of list reconstruction alongside the inflated memory usage, due to each element holding a list of its unique neighbourhood, hence limiting the applicability to systems of low entropy and scale [165].

Verlet lists have primarily been used in CPU code, as many components of the overall technique are easily mapped to a thread parallel architecture. Lipscomb et al have developed a GPU parallel implementation of the Verlet list neighbour algorithm for use with MD simulations [102]. Their technique primarily takes advantage of the primitive scan and pair sort functionalities provided by CUDA Parallel Primitives (CUDPP), to calculate neighbour lists on the GPU, bypassing the latency experienced if otherwise calculating them on the CPU.

The results of Lipscomb et al showed a peak speedup over CPU of 30x when processing an MD system of 10219 particles when compared with the CPU implementation. This improvement scales with the count of particles, with a mere 76 particles the CPU implementation performed fastest.

Similarly, Páll and Hess modified Verlet lists to overcome the limiting factor imposed by the regular reconstruction [129]. Their approach adds a buffer distance so that neighbour lists can include actors slightly outside the range. Additionally, actors are processed in clusters of 2-8 to improve alignment of threads under the parallel architecture of CPUs and GPUs. They conclude that their method is only appropriate for cheap neighbour interactions, as the redundant interaction calculations due to the buffer distance can outweigh other improvements.

Others, such as Howard et al, have explored further techniques for GPU optimisation of Verlet lists [80], however, their optimisations address requirements of MD applications, such that different entities have different search radii. Their approaches include stencilling, whereby cells of the Moore neighbourhood are skipped, and linear bounding volume hierarchies are introduced.

2.4.4 Neighbourhood Grid

Joselli et al have described a data structure, neighbourhood grid. This data structure has been designed to optimise SPH by assigning each particle to its own bin, rather than the uniformly sized bins found in USP [86]. This binning system instead creates an approximate spatial neighbourhood, which has allowed performance improvements of up to 9x when compared to USP methods [85].

By storing particles to unique bins within a regular grid, neighbourhood searches can be carried out surveying a constant radius of directly adjacent bins. With a radius of 1 cell in 3 dimensions this reduces all neighbourhood searches to checking 26 bins, and in 2 dimensions only 8 bins. This provides constant time queries, rather than queries that increase with the density of particles.

As particles move, the data structure must be sorted such that spatial neighbours remain close within the grid. A bi-tonic sort was used in their development and testing, which sorted each dimension in a separate pass, however, the focus was on their data structure and the sorting algorithm used is independent of that. Earlier sorts are not repeated if the 2nd or 3rd passes also make changes. They found that this would impact the performance whilst only correcting around a single percentage of particles.

This method sees great performance improvements due to the imposed uniformity which reduces divergence and is favoured by the SIMT architecture of GPUs. However, this uniformity comes at the cost of an approximate method, incapable of providing the precision needed by many MAS. Furthermore, despite demonstrating pedestrians, it is not explained how their approach would handle less uniform and skewed distributions. In these cases, some actors would not map cleanly to the grid whilst retaining locality.

2.4.5 Primitives

There are several parallel primitive libraries available for use with CUDA. These each provide GPU optimised implementations of algorithms common to most GPGPU tasks (e.g. sort, reduce, partition, scan) in a templated fashion. These allow developers to utilise highly optimised algorithms, without needing to spend their own time profiling. Thrust [5] and CUDPP [3] both consist of C++ template libraries. Their algorithms overlap in many cases and provide similar performance (this may vary per algorithm). However, Thrust is primarily targeted at CUDA programmers, providing container classes to abstract the sharing of data between host and device, whereas CUDPP is capable of being used by non-GPU code.

CUB [2] instead targets every layer of CUDA programming, allowing primitives to be applied at warp, block and device-wide scopes. This lower-level architecture allows the algorithms to be used inside kernels, removing any overhead of additional kernel launches. CUB's documentation details the performance of its device level algorithms against those available in Thrust across several CUDA-capable devices. CUB is shown to range from matching Thrust's performance to handling twice as many inputs per second across the tested devices.

2.4.6 Summary

This section has shown that there are a number of parallel (GPU) data structures for use with spatial data and they are often maximally performant under limited circumstances, leaving space for improvements in many areas. However, due to the diverse applications of these data structures, not all are suitable for providing FRNN searches.

Handling construction in parallel has been performed using bespoke and parallel primitive algorithms (e.g. sort) to manage arbitration. Others have subdivided construction into smaller units to be later joined. This, however, is unlikely to scale well across highly parallel architectures such as GPUs, and requires data to be ordered prior to subdivision.

USP has been most widely used across fields spanning to provide FRNN search for complex system simulations [103, 91, 124, 55]. However others have considered the applicability of k-d trees and r-trees [62, 44]. One reason for this is likely the ease with which USP parallelises, whereas approaches utilising trees require greater levels of data specific optimisation to produce an efficient tree. However, many of these complex simulations utilising FRNN search highlight it as the mostly costly component of the models.

Whilst USP and k-d trees have both been used for FRNN search in GPU parallel complex system simulations, the consensus shows USP as the predominant data structure for this application. Section 2.5 explores the construction and query stages of FRNN search using USP in greater detail, and highlights state of the art approaches for improving its performance.

2.5 The Uniform Spatial Partitioning Data Structure

Following the earlier sections of this chapter, which have provided an overview of data structures and techniques capable of handling spatial data, this section delivers an in depth assessment of the uniform spatial partitioning (USP) data structure, the primary data structure used to provide FRNN searches across a range of fields and the focus of this thesis.

The USP data structure is a static multi-map with a known key space, which uses a coherent perfect hashing scheme to map continuous spatial locations to its discrete keys. However, the construction of the data structure, and how it is queried to deliver FRNN search on GPUs, does not share many parallels with traditional dynamic hashing data structures explored earlier in this chapter.

The USP data structure represents the environment by subdividing it into a grid of uniformly sized bins. Bins are assigned indices, which are calculated based on their grid position (e.g. $pos_y dim_x + pos_x$). This discretisation of the environment allows entities, with continuous coordinates, located within the environment to be stored according to the discrete bin in which they reside.

Unbounded environments may either perform clamping or wrapping to ensure that all locations are first mapped to the regular environment bounds, so that they can be assigned to discrete bins. However, the optimal choice of solution is dependent on the application. As such, this thesis focuses only on regular bound environments.

The delivery of FRNN search can be divided into two stages: construction and querying. The incremental nature of the complex system simulations relying on FRNN search leads to both stages being required each time-step. As such, the performance of both stages significantly impacts the overall runtime of a simulation.

The remainder of this chapter first details the memory structure of the USP data structure. This is followed by explanations of the two stages of operation, construction and querying, highlighting their possible areas for improvement. After which, prior research from literature which has sought to improve the USP data structure is discussed. The chapter concludes with a summary of the important highlights.

2.5.1 Memory Layout

Figure 2.10 illustrates a simple environment and its associated USP data structure [58]. 10 items are spatially stored in an environment of $4R \times 4R$ scale (shown top left), the discretisation of this continuous environment and how it is mapped to the USP data structure is then demonstrated. Similarly, those items which fall within a FRNN query of the source item are highlighted. In a complex system simulation, the environment would likely be many times larger than shown with thousands of actors performing queries concurrently.

The USP data structure utilises two arrays, the storage array, which provides a compact store of all neighbour data stored within the data structure, and the Partition Boundary Matrix (PBM), which provides an index to map the boundaries of each bin's contained data within the neighbour data array.

Elements within the storage array are stored in the order of their containing bin's index. Where multiple elements exist within the same bin (e.g. bin 6 of Figure 2.10), the sub-ordering of elements within a bin is undefined and may vary according to the sort implementation used. Queries of the data structure always consider all elements within a given bin rather than isolating specific elements, hence, the order of elements within a bin does not matter.

The amount of memory required by the USP data structure can therefore be calculated as the total required for both the storage array and the PBM. The size of the neighbour data array is simply that of all the neighbour data to be stored, as this array contains no buffer space.

There are two options for PBM layout, described in the following section. The memory requirements of each are tied to the number of bins. The first style of PBM requires two unsigned integers (or unsigned shorts) for each bin to form an array for start indices and an array of end indices of each bin within the storage array. As the end index of one bin is equal to the start index of the following bin, a more compact PBM is possible, requiring a single unsigned integer per bin, plus an additional unsigned integer. These integers denote the starting index of the corresponding bin's data within the storage array and the

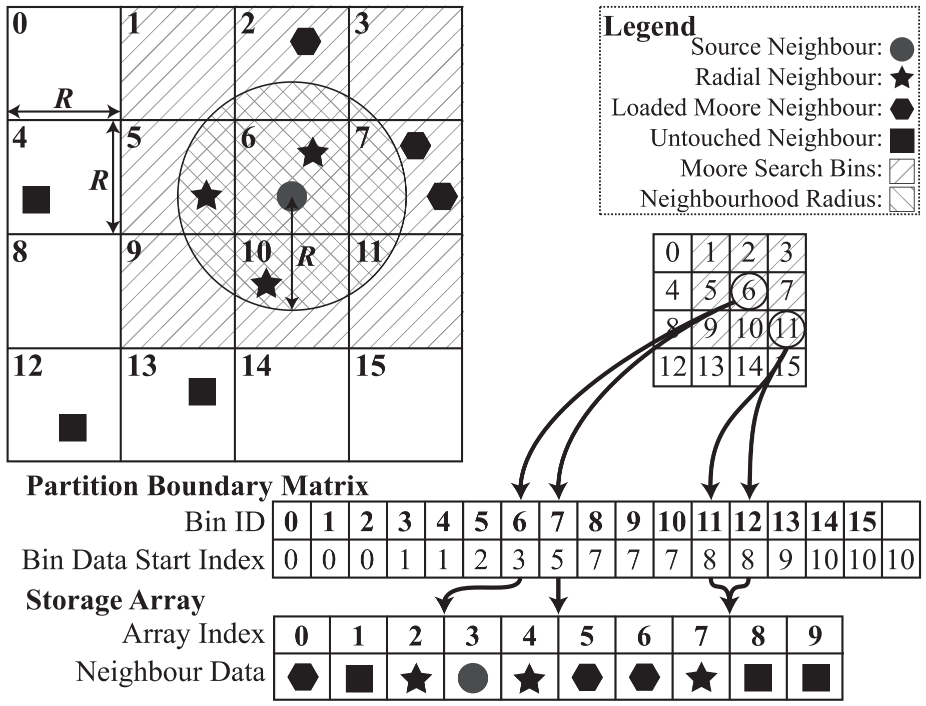


Figure 2.10: Visual representation of an environment and how its data is stored and accessed under GPU uniform spatial partitioning. The PBM acts as an index to each bin's subset of the neighbour data array. Bin ID and Array Index denote the indices of elements within the arrays, and are not stored in memory.

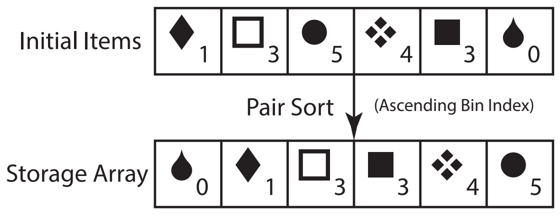
additional index at the end will always hold the length of the storage array. This compact technique allows the bounds of any bin's data within the storage array to be identified by reading two neighbouring values from the PBM, the first of which is the bin's index ($pbm[bin_i]$, $< pbm[bin_{i+1}]$). This is demonstrated for bins 6 and 11 within Figure 2.10, for bin 6: PBM indices 6 & 7 point to the range $[3, 5)$ within the storage array, which contains the two items found within bin 6 of the environment.

The total number of bins within the environment is highly dependent on the level of discretisation required. If bin widths are a small proportion of the environment dimensions, the total quantity of bins can easily reach the millions (256^3) or even billions (1024^3) in 3D due to the cubic relationship. This is both harmful to the memory utilisation of the PBM and construction time, as in such a case most bins are likely to remain empty.

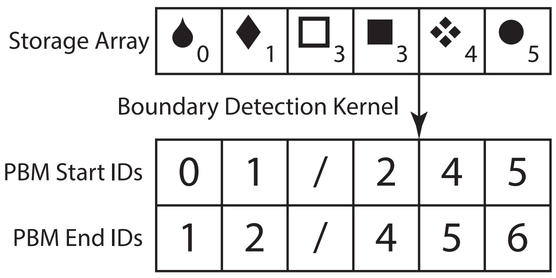
2.5.2 Construction

Each time an element stored within the data structure moves spatially, the data structure must be reconstructed. In practice within MAS this leads to the data structure being reconstructed each time-step, after actors update their locations. Therefore it is important that the cost of construction is minimised.

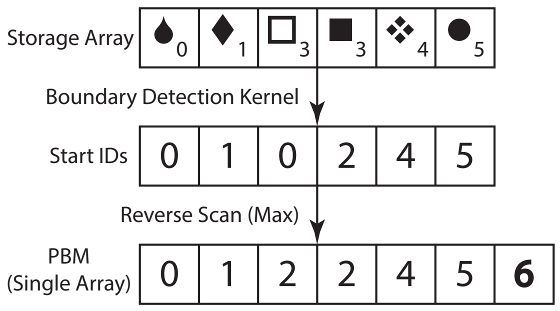
The construction of the USP data structure uses the parallel primitives of sort and scan, and a reorder kernel to move elements to their sorted indices. Both sort and scan implementations are widely available in highly optimised GPU primitive libraries such as CUB [2], Thrust [5] and CUDPP [3]. Although the USP data structure is static, requiring a full reconstruction to relocate a single element, the construction's utilisation of these common operations allows it to remain viable on the SIMT architecture of GPUs.



(a) A collection of items may be pair sorted according to their bin index into the storage array.



(b) A PBM constructed from start and end index arrays can be produced from the storage array by detecting the boundaries. In this example '/' denotes the invalid value flag.



(c) As a more costly alternative, a single array PBM can be produced with an extra scan pass. 0 must be used as the invalid value flag. The final index is manually assigned based on the total number of bins.

Figure 2.11: The generation of the USP data structure's storage array (a) and PBM, which can take the form of one (c) or two (b) arrays.

Figure 2.11a shows a collection of elements, each paired with their corresponding bin's index, which are to be stored in the USP data structure. First, the elements must be pair-sorted in ascending order of their bin indexes. This sorted array is the storage array of the USP data structure.

Next, the PBM must be generated. Figure 2.11b shows how this can be achieved by allocating start and end index arrays of length corresponding to the number of bins, initially filled with an invalid value flag (denoted in Figure 2.11b as '/'). A kernel is then launched, with one thread corresponding to each stored item. Each thread checks both the following and their own item. If a boundary is found, the start and end arrays are updated for the corresponding bins. This produces a PBM across two arrays, identifying the storage bounds of each bin. Where the invalid value flag is read, a bin contains no items. Alternatively, as shown in Figure 2.11c, a more compact, single array PBM (start array only) can be produced to reduce the memory footprint. However, this requires an additional reverse exclusive scan (max operation) to update the invalid flags (from 0) to the following valid boundary.

2.5.3 Query

To access data located within the radial neighbourhood of a position, the position's containing environment bin is identified and all neighbour data stored within the bin and it's adjacent bins (the Moore neighbourhood) are then iterated. As shown in Figure 2.10, only those with a position inside the radial neighbourhood are considered by the simulation. In two dimensions this means that neighbour data within a spatial area 2.86x larger than required (2D neighbourhood area: πR^2, 2D Moore neighbourhood area: $(3R)^2$) are accessed. In three dimensions this increases to 6.45x (3D neighbourhood volume: $\frac{4}{3}\pi R^3$, 3D Moore neighbourhood volume: $(3R)^3$). Thus, in both 2D and 3D environments the majority of memory accesses can be assumed to be redundant.

By subdividing the environment uniformly, it is possible that non uniform distributions of data may lead to many empty bins with a small number of densely populated bins. However, the benefit of uniform subdivision is that it allow neighbours to be searched using 1-2 reads per bin. This access to bins would not be as direct, likely requiring significantly more reads per bin under non-uniform subdivision, due to the need to traverse the division hierarchy.

2.5.4 Uniform Spatial Partitioning in Complex System Simulations

As briefly discussed in Chapter 1 and previous sections of this chapter, USP is actively used to provide FRNN search within many fields which simulate complex systems [103, 35, 124, 55].

This subsection explores how USP has been applied within different fields in greater detail, exploring reported properties of execution to understand the range of properties required to be reflected in a general case of FRNN search.

Pedestrian, Crowd, Flock

Since Reynolds seminal flocking paper [135], modellers of crowds of pedestrians and flocks of animals have sought to move from discrete cellular automata to actors represented in continuous space. This has seen implementations move to GPU acceleration, utilising USP to provide their FRNN search. Two recent examples of this are Charlton et al [35], and Demšar et al [42].

This field is often concerned with real-time models for visualisation, and hence the population size is limited to what can be simulated for real-time rendering. Furthermore, many applications of pedestrian modelling are concerned with sparsely populated environments, which may not facilitate high populations. Models are often simulating a single plane, and can hence be modelled in two dimensions, however, aerial flocking and buildings with multiple floors may require models to utilise three dimensional partitioning.

Across these two papers, populations are demonstrated from below 1×10^3 actors to 500×10^3. Furthermore, it is noted that the crowd model has actors which move at variable speeds with variable radii. This is representative of a diverse demographic within the modelled population.

Whilst Charlton et al mention ‘large densities’ found in crowd models, and concede that their model is unable to handle the highest densities, exact figures representing the density of actors relevant to the search radius are not provided.

Whilst exclusively high density crowd models may support short ranges of actor vision, due to the impact of being tightly packed, visually realistic modelling of environments which combine sparse areas and dense areas is likely to require a greater range of vision. This may lead to the densest areas of an environment (e.g. environmental bottlenecks) producing much higher than average densities. This could be further exacerbated in multi-level environments, whereby three dimensional USP is required.

Biological

There are various approaches to cell level biological modelling found in literature. Of particular relevance to USP are those of cell-centre modelling and lattice site modelling. Whilst cell-centre modelling makes full use of FRNN search, lattice site modelling uses discrete locations, such that direct access to the USP data structure’s multi-map is required.

Similar to crowd modelling, biological models are gradually shifting to utilise agent-based approaches, where individual cells are represented. Three examples are discussed in this section, covering medical nanorobotics [103], immune reaction [150] and epithelial skin cells [36]. Across these three examples, populations range from 20×10^3 actors to over 300×10^3. Notably, Tamrakar states actor population was limited to 20,000 by the CPU based simulator simulator (Repast) that they were assessing in parallel with the GPU based simulator FLAMEGPU. Additionally, the population within the model of epithelial skin cells increases by over 2x over 1000 iterations of the model. This is notable behaviour, not previously discussed in pedestrian models.

As before, none of these papers discuss the details related to density. However Liu does note the worst case, whereby all nano-robots are located within the same subdivision. This may suggest a model which can experience high densities for brief periods.

Due to the diversity of biological structures modelled, it seems likely that densities and distributions are highly variable between models. Models of tissue growth, are likely to represent packed cells which apply forces to their direct neighbours, whereas models representative of cells within biological fluids are likely to require a greater range in order to model highly mobile cells.

Molecular Dynamics

Molecular Dynamics (MD) forms a special case, whereby USP has been used extensively and tuned for the specific properties of the field to provide efficient FRNN search. MD involves incredibly high actor populations, over 100 million and with a consistent actor density [55]. This enables several techniques outside of the scope of the general case targeted within this thesis.

In order to facilitate such high actor populations, MD implementations of USP both perform more work per thread, and utilise distributed approaches with thousands of GPUs operating simultaneously [55]. These scales are significantly higher than found in all other fields of complex system simulation, as such they sit outside the general case of FRNN search.

One approach seen is that of replacing 1 thread per actor, with 1 thread per bin [17]. This reduces repeated memory accesses and can provide additional benefits via instruction level parallelism. However, the approach is limited to low densities as a high number of actors per bin would lead to registers spilling into higher latency GPU memory. Some higher densities can be handled by instead processing 1 bin per warp or thread block [9], however, again, this is only suitable for limited actor densities and distributions. Furthermore, comparatively low actor populations seen in other fields are unlikely to achieve full device utilisation if multiple actors are executed per thread, which would additionally harm performance.

When distributed, it becomes necessary for actors within boundaries of a GPU's simulated area to be communicated to devices handling neighbouring areas. This has been solved with techniques such as ghost particles [55], where boundary particles are shared each time the data structure is built, and chequerboard decomposition, where only actors within 1/8 of bins are executed simultaneously to avoid conflicts between neighbouring bins [17].

Smoothed Particle Hydrodynamics

Whilst Smoothed-Particle Hydrodynamics (SPH) has similarities with MD, most applications are still confined to a single GPU's memory limitations. Whilst this limitation will vary between models and hardware, the upper bound appears to be in the low millions [115, 77]. However, others, particularly using SPH to drive graphical applications, are content with using actor populations of less than a million [56].

Yang et al suggest two densities are present in SPH simulations. Whilst low densities of 30 neighbours may be utilised for visual applications, computational physics instead requires around 80 neighbours to achieve the required precision [167].

Summary

This subsection is only able to demonstrate a small sample of applications for which USP provides FRNN search. Complex system simulations are employed across a wide number of broad fields such as social science, earth sciences, physics and engineering. Many of these applications will require the representation of spatial actors, so likely also rely on FRNN search, which may be provided by USP, however, researchers with domain specific modelling knowledge are often unconcerned with the supporting algorithms such as FRNN search, due to their reliance on frameworks such as FLAMEGPU.

The properties highlighted in this subsection provide a representative sample of the general case to test optimisations against in the following chapters.

The applications of USP and FRNN search explored in this subsection have shown agent populations ranging from 1×10^4 to 5×10^6, limited by the requirements of performance and a single GPU's memory capacity.

Measures of neighbours returned per FRNN query are less available, however, Yang et al provide a range of 30-80 neighbours per query [167], covering SPH applications interested in both visuals and precision. Again, in this case the density limitation for visuals is bound by performance.

Distributions found within complex simulations vary from mostly compact arrangements of actors found in some biological and SPH models to loose non-uniform distributions found within other biological models and some pedestrian environments.

Other general properties relevant to FRNN include actors travelling at variable speeds, actors of variable radii and actor populations which change during execution. Entropy, a metric for actor speed is unlikely to impact performance, as GPU sorting methods are not incremental so perform equally for pre-sorted and un-sorted data. Actors of variable radii requires the addition of actor radii within shared neighbour data, and a search radius capable of capturing the centre point of the largest radius actors. This can be achieved by testing higher densities. As each iteration of complex system simulations are independent, so long as appropriate memory is allocated, a growing population simply defines a range of populations to be assessed.

2.5.5 State of the Art

The preceding sections of this chapter have introduced the basic data structure and algorithms related to FRNN search using the USP data structure. This section introduces innovations from the literature which have sought to augment them to improve performance.

Common GPU optimisation techniques apply to FRNN search, such as use of texture cache, faster mathematical operations, structure of arrays (coalesced memory accesses) and radix sort [27, 141]. These approaches are familiar concepts to those working with GPGPU development and are often discussed in official NVIDIA documentation and guidance for

CUDA developers.

The scope of research towards the construction stage has been rather limited due to its reliance on fundamental primitive algorithms, which are already the focus of research by other disciplines concerned with improving parallel sorts and similar [112, 78, 49].

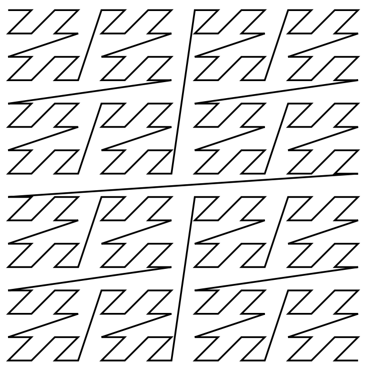


Figure 2.12: A level 4 Z-order curve (Morton code) in 2 dimensions.

Hongyu et al optimised the reconstruction of the data structure by developing a novel sorting technique that takes advantage of the knowledge that particles can only move to neighbouring bins, utilising a prefix sum of the changes to each bin's capacity [149]. They were able to improve performance in a small SPH simulation of 8192 particles within a 16^3 grid. However, overall performance was equal to that of their initial unoptimised reconstruction when applied to larger simulations. Section 4.4 shows how in larger simulations the construction time is a small fraction of that required for the FRNN search, so optimisation efforts targeting the search process are likely to have a greater overall impact.

Research regarding the USP data structure has primarily considered improvements towards the query stage. It is hypothesised that this is a result of the relatively low cost of construction in most applications, Chapter 4 explores this fully. Other data structures such as 'neighbourhood grid' and Verlet lists have been proposed and used, however (see Section 2.4), the USP data structure remains most performant for GPU hardware.

Goswami et al were able to improve the performance of GPU FRNN searches during SPH on GPUs [56]. They adjusted the indexing of the bins, which the environment is subdivided into, so that they are indexed according to a Z-order space-filling curve (also known as a Morton code). The space-filling curve maps multi-dimensional data into a single dimension whilst preserving greater spatial locality than regular linear indexing. This creates power-of-two aligned square grids of bins, with contiguous Z-indices, such that bins containing neighbour data are stored in a more spatially coherent order. This additional locality intends to ensure that more neighbourhoods consist of contiguous blocks of memory, such that surplus data in cache lines is more likely to be required in subsequent requests, reducing additional latency caused by cache evictions.

Also using space-filling curves, the AMBER GPU molecular dynamics toolkit, as described by Salomon-Ferrer et al [141], uses particle sorting within bins according to a $4 \times 4 \times 4$ Hilbert space-filling curve in order to calculate the forces between molecules during simulation. In addition to possible benefits of spatial locality, this may allow for the neighbourhood cut off to be extended via a bitmask filter, reducing the quantity of redundant accesses to neighbourhood data. As a small part of a much larger unit of research, the precise method of implementation and impact are not detailed and hence understanding this remains an open research question. Subdividing bins into a further 32 sub-bins may only be viable for densities where the sub-bin occupancy remains close to 1.

Furthermore, Rustico et al utilised an interleaved storage of neighbour data to improve the coherency of memory accesses [140], however, again this remains an open research question as the impact was not assessed in isolation.

In reproducing the techniques of Goswami et al and trialling other possible optimisations linked to memory locality and access coherence, it was discovered that performance improvements were only found when testing code which lacked compiler optimisations (debug builds). Therefore, it is likely that the architectural and compiler improvements since 2010 which have greatly improved the caching on GPUs, has rendered these techniques more costly than any savings achieved. This aligns with the observed reduction in the significance of coalesced memory accesses on newer GPU architectures, as discussed in Section 2.3.1. This raises the question, which other recent GPU hardware changes could impact the performance of FRNN search.

Howard et al have proposed using multiple threads to process each individual neighbourhood query [81]. Warp shuffle commands are then employed in order to reduce the local results from threads working the same query. They found this was able to improve performance at low device utilisation, by utilising a greater number or threads. This approach has merit, however, limits itself to cases of low actor population/occupancy.

Whilst FRNN search via the USP data structure is not utilised for approximate techniques, the earlier discussed Neighbourhood Grid technique was inspired by USP for use in SPH [85]. This approach sorts neighbour data such that a maximum of 1 item is stored per bin. The query is then performed by only surveying the Moore neighbourhood of 26 bins. Whilst this has great implications for performance, through the likely elimination of divergence between threads, 26 neighbours is below even the suggested lower bound of 30 neighbours for visual applications of SPH by [167]. Therefore, it is unlikely to be an appropriate alternative for all but the most performance constrained models. However, it does highlight the significant cost that divergence has on performance of FRNN search.

2.5.6 Summary

This chapter has introduced USP, which is the data structure most commonly used to provide FRNN search on GPUs, and it has detailed the operation of the two stages, construction and query, which are required to deliver FRNN search. Furthermore it has examined existing optimisations to the data structure and its use for FRNN search as found in the literature.

In combination, these demonstrate the shortcomings of the data structure, which whilst the most promising data structure for delivering FRNN on GPUs still presents an opportunity for improved performance for a general case of FRNN search.

The literature has identified several open research questions: How can modern GPU hardware/features be applied to improve FRNN search? How can the redundant area between the Moore neighbourhood and radial search area during FRNN queries be reduced? Is it possible to reduce divergence during queries without reducing FRNN queries to an approximate method? These are investigated further in the remainder of this thesis.

2.6 Conclusion

This chapter has explored applications of FRNN search within complex system simulations, techniques for handling spatial data and utilising GPUs to accelerate this handling of spatial data.

This has provided an understanding of how FRNN search is central to a wide range of complex systems simulations where thousands of interacting actors require awareness of their spatial surroundings. Yet it also remains the mostly costly common process between many of them. The properties of many of these applications have been documented to provide guidelines to represent realistic bounds for the general case targeted for improvement within this thesis.

There are many unique applications for spatial data, with many similarly unique data structures suited to their purpose. An exploration of classic techniques for handling spatial data, primarily via hashing and tree data structures, has highlighted the features most applicable, like perfect hashing, for the requirements of FRNN search.

Although problems like nearest neighbour and k nearest neighbours are similar to FRNN search, their difference make them more suited to tree data structures. kd-trees are highly suited for nearest neighbour search and k nearest neighbours in high dimensional spaces which are both widely used in machine learning. In contrast, k nearest neighbours, in the spatial domain (two and three dimensions) can perform well with uniform subdivision techniques like USP, however, when dealing with non-uniform distributions the uniform subdivision quickly leads to high quantities of bin accesses. As such, Octrees have been recommended for use with k nearest neighbours where skewed actor distributions are likely to occur.

In contrast, the USP data structure has been shown to be the most appropriate technique across a diverse range of fields to deliver FRNN search on GPUs. To date, state of the art optimisations have primarily attempted to optimise the query stage by improving memory locality and coherence, the impact of which has been greatly reduced with architectural improvements for modern GPUs, or through the removal of divergence by switching to approximate methods. Improvements to construction have been limited, as the algorithm relies heavily on common parallel primitive algorithms which are already highly optimised, although bespoke sorting for the incremental movement has been applied, in an attempt to achieve an equivalent of parallel insertion sort.

The review of state of the art methods has raised several open research questions related to USP and FRNN search: How can modern GPU hardware/features be applied to improve FRNN search? Is it possible to reduce divergence during queries without reducing FRNN queries to an approximate method? How can the redundant area between the Moore neighbourhood and radial search area during FRNN queries be reduced? These are investigated in Chapters 4, 5 & 6 respectively.

The following chapter introduces models capable of assessing the performance of FRNN. Some of these benchmark models are capable of representing a range of properties identified as forming the general case in Section 2.5.4. Others are examples of real-world models, to demonstrate the impact of FRNN search in practice.

Chapter 3

Benchmark Models

Benchmark models are important, as they provide a controlled environment where variables which impact performance can be modified independently. Chapter 2 showed that FRNN search has been applied across a wide range of complex system simulations, each with unique actor representations and distributions to be handled by FRNN search. In order to target the general case and understand how variations in properties such as population size, density and entropy impact performance, it is necessary to utilise benchmark models capable of controlling these properties when assessing performance.

This chapter contributes two benchmark models and discusses their effective usage. These models are essential in understanding the performance of FRNN search during complex system simulations in order to answer the main research question and demonstrate evidence of the main thesis contributions: C1, Atomic counting sort for construction of the USP data structure; C2, Strips optimisation for FRNN queries; C3; Proportional Bin Width Optimisation for FRNN search.

The Circles model provides a benchmark for FRNN search. It both allows scaling across a range of properties and moves actors through a range of distributions. Hence, it is able to provide a more general overview of the range of impacts to FRNN search performance than selecting a specific real-world model and/or scenario. This model has been used throughout when assessing the impact of changes to the query stage of FRNN search.

Similarly, the Network model provides a benchmark for assessing the performance of discrete accesses to the data structure across a range of properties. As a secondary application for the data structure, accesses to individual bins can be used for representation of networks or biological lattice sites. This model has been used primarily in Chapter 4 to highlight the greater impact improvements to the construction stage may have on these applications of the data structure.

Additionally, this chapter introduces three models from FLAMEGPU which act as benchmarks of how changes may look when applied to a full model with additional model logic unrelated to the USP data structure. These have been used throughout the thesis to further assess the impact of the most favourable optimisation configurations, as determined using the benchmark models.

The following sections describe the models summarised above in greater detail.

3.1 Continuous: Circles Model

The Circles benchmark model provides a means for benchmarking FRNN search using a simple particle model analogue which is typical of those seen in cellular biology and other multi-agent simulations. Within the model, scattered actors located in a 2D or 3D environment converge over time to form circles (Figure 3.1b) or hollow spheres respectively. An earlier version of the Circles model, was published as 'A Standardised Benchmark for Assessing the Performance of Fixed Radius Near Neighbours' [38]. The version present in this thesis has a smoothed force equation (Equation 3.4) which significantly reduces actor jitter at convergence.

The Circles model is highly configurable, operable in both two and three dimensions and allowing a range of parameters to be controlled. During execution the model moves actors from an initial uniform random distribution to a non-uniform clustered distribution. This flexibility allows the model to be used as a general benchmark for FRNN search, highlighting the impacts of optimisation across a broad range of population sizes, densities and entropies, as observed in a range of complex system simulations.

Within the model each actor represents a particle whose location is clamped between 0 and $W-1$ in each axis. Each particle's motion is driven by forces applied from other particles within their local neighbourhood, with forces applied between particles to encourage a separation of r. Examples of equivalent behaviour are shown in cell centre models [163, 44], where damped force resolution is applied, enforcing strict separation of biological cells which require close contact in tissue formation.

This model has been used in Chapters 5 and 6 to assess the impact of optimisations to FRNN search across a broad range of configurations.

3.1.1 Model Specification

The benchmark model is configured using the parameters in Table 3.1. In addition to these parameters the dimensionality of the environment (E_{dim}) must be decided, which in most cases will be 2 or 3. The value of E_{dim} is not considered a model parameter as changes to this value are likely to require implementation changes.

Parameters are managed so that they can be controlled independently. As the density parameter is increased, the number of particles per grid square and ring increases, and the environment dimensions decrease. Changes to the actor population size simply affect the environment dimensions so as to not change the density.

Parameter	**Description**
k	The unified force dampening argument. Increasing this value adds energy, increasing particle speeds.
r	The radial distance from the particle to which other particles are attracted. Twice this value is the interaction radius.
ρ	The density of actors within the environment
W	The diameter of the environment. This value is shared by each dimension therefore in a two dimensional environment it represents the width and height. Increasing this value is equivalent to increasing the scale of the problem (e.g. the number of actors) assuming ρ remains unchanged.

Table 3.1: The parameters for configuring the Circles benchmark model.

Initialisation

Each actor is solely represented by their location. The total number of actors A_{pop} is calculated using Equation 3.1[1]. Initially the particle actors are uniform randomly positioned within the environment of diameter W and E_{dim} dimensions. Figure 3.1a provides a visual example of this. A denser population would see more actors per bin and a larger population would see the environment grow.

$$A_{pop} = \lfloor W^{E_{dim}} \rho \rfloor \tag{3.1}$$

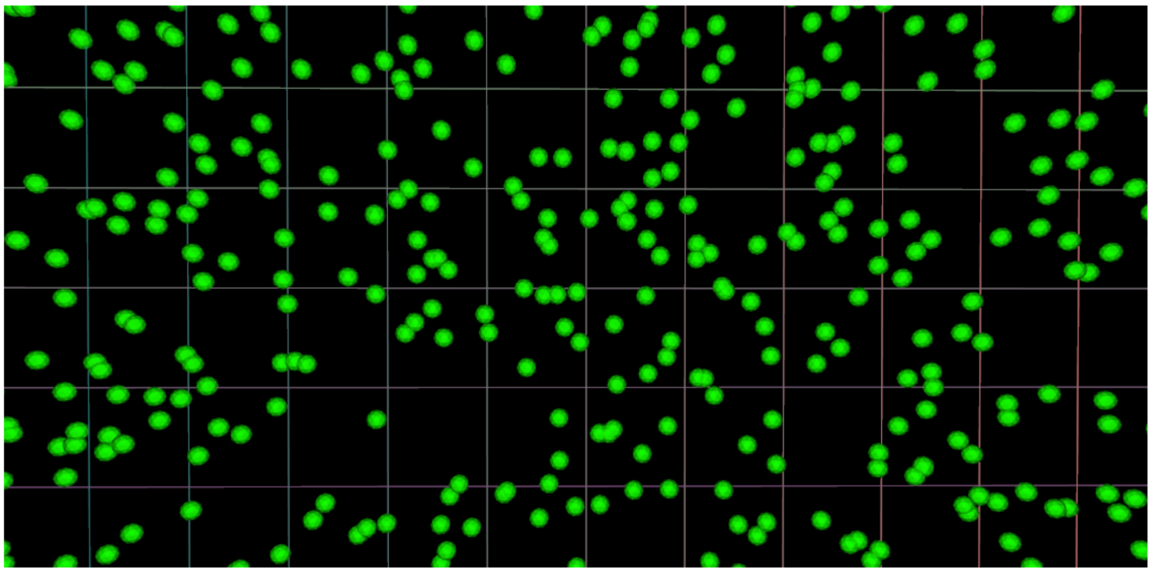
Single Iteration

For each timestep of the benchmark model, every actor's location must be updated. The position x of an actor i at the discrete timestep $t+1$ is given by Equation 3.2, where F_i denotes the force exerted on the actor i as calculated by Equation 3.3. Within Equation 3.3, F_{ij} represents the respective force applied to actor i from actor j. The values of $F_{ij}$is calculated using Equation 3.4. The unified force parameter is multiplied by the distance sine of the normalised separation of x_i and x_j, and the unit vector from x_i to x_j. After calculation, the actor's location is then clamped between 0 and $W-1$ in each axis.

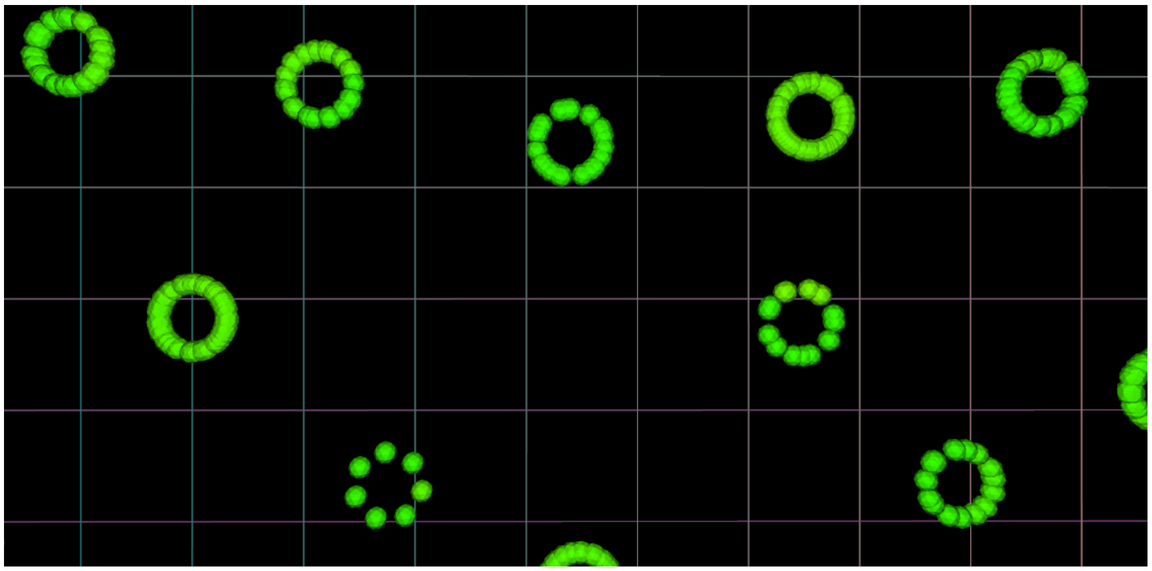
$$\overrightarrow{x_{i(t+1)}} = \overrightarrow{x_{i(t)}} + \overrightarrow{F_i} \tag{3.2}$$

$$\overrightarrow{F_i} = \sum_{i \neq j} \overrightarrow{F_{ij}} \tag{3.3}$$

[1] $\lfloor\ \rfloor$ represents the mathematical operation floor.



(a) Uniform Random Initialisation State



(b) Circles End State

Figure 3.1: An area from the visualisation of the state of the Circles model. The spheres represent point based actors. The grid lines show the subdivision of the environment into bins. (a) The start state under uniform random initialisation whilst executing in 2-dimensions with a Moore neighbourhood volume average of 37. (b) The end state whereby the model has converged to a steady state with a Moore neighbourhood volume average of 41.

$$\overrightarrow{F_{ij}} = sin(-2\pi(\frac{\|\overrightarrow{x_j x_i}\|}{r}))k\frac{\overrightarrow{x_j x_i}}{\|\overrightarrow{x_j x_i}\|} \tag{3.4}$$

Algorithm 3 provides a pseudo-code implementation of the calculation of a single particle's new location, where each actor only iterates their neighbours rather than the global actor population.

Algorithm 3 Pseudo-code for the calculation of a single particle's new location.

```
vec myOldLoc;
vec myNewLoc = myOldLoc;
foreach neighbourLoc
{
  vec toVec = neighbourLoc-myOldLoc;
  float separation = length(toVec);
  if(separation < RADIUS and separation > 0)
  {
    float k = sin((separation/RADIUS) * -2PI);
    myNewLoc += UNIFIED_FORCE * k * normalize(toVec);
  }
}
myNewLoc = clamp(myNewLoc, envMin, envMax);
```

Validation

There are several checks that can be carried out to ensure that the circles model has been implemented correctly. The initial validation technique relies on visual assessment.

During execution, if the force k is positive, particles can be expected to form clusters arranged as rings in two dimensions (Figure 3.1b) and hollow spheres in three dimensions. This has the effect of creating hotspots within the environment where bins contain many particles and dead zones where bins contain few or no particles. This structured grouping can be observed in complex systems such as pedestrian models, where a bottleneck in the environment creates a non-uniform distribution, with high and low actor densities before and after the bottleneck respectively [131, 64]. These clusters also lead to the Moore neighbourhood volumes more closely matching the radial neighbourhood volumes.

Instead, when k is negative, particles can be expected to separate, although small clusters may remain. Executed with uniform random initialisation, this sees insignificant movement such that the pattern is harder to visually classify.

More precise validation can be carried out by seeding two independent implementations with the same initial particle locations. With appropriate model parameters (such as those in Table 3.1), it is possible to then export actor positions after a single iteration from each implementation[2]. Comparing these exported positions should show a parity to several decimal places, with significant differences between the initial state and the exported states.

[2]It is recommended to export actors in the same order that they were loaded, as sorting diverged actors may provide inaccurate pairings.

The precision of this validation is likely to vary between hardware implementations. Non deterministic elements of implementations lead to floats accumulating in different orders such that the resulting floating point representations may differ between runs. Due to these floating point arithmetic limitations, it was found that a single particle crossing a boundary between two FRNN search radii can subsequently cause many other particles to differ between simulation results due to the chaotic nature of emergent models such as this.

3.1.2 Effective Usage

In order to best utilise the Circles benchmark model it is necessary to execute the model over a range of configurations to subject the target implementation to a range of properties which may impact performance. The properties which may affect the performance of FRNN search implementations are actor quantity, neighbourhood size, actor speed and location uniformity. Whilst it is not possible to directly parameterise all of these properties within the Circles benchmark, a significant number can be controlled to provide understanding of how the performance of different implementations is affected.

To modify the scale of the problem, the environment width W can be changed. This directly adjusts the actor population size, according to the formula in Equation 3.1, whilst leaving the density unaffected. Modulating the scale of the population is used to benchmark how well implementations scale with increased problem sizes. In multi-core and GPU implementations this may also allow the point of maximal hardware utilisation to be identified, where smaller population sizes do not fully utilise the available hardware.

Modifying either the density ρ or the radius r can be used to affect the number of actors found within each neighbourhood. The number of actors within a neighbourhood of radius r can be estimated using Equation 3.5. This value assumes that actors are uniformly distributed and in practice will vary slightly depending on the search origin.

High density neighbourhoods, beyond the bounds of the general case (30-80 neighbours in radius), can be used as a rough analogue for increasing the size of neighbour data. Both have the effect of accessing more message data per bin, so impacts to scaling will be related.

$$N_{size} = \rho\pi(2r)^{E_{dim}} \tag{3.5}$$

Modifying the speed of the actor's motion affects the rate at which the data structure holding the neighbourhood data must change (referred to as changing the entropy, the energy within the system). Many implementations are unaffected by changes to this value. However optimisations such as those by Sun et al [149] should see performance improvements at lower speeds, due to a reduced number of actors transitioning between cells within the environment per timestep. The speed of an actor within the Circles model is calculated using Equation 3.3. There are many parameters which impact this speed within the Circles model. Since a particle's motion is calculated as a result of the sum of vectors to neighbours it is clear that the parameters affecting neighbourhood size (ρ & r) impact particle speed in addition to the forces F_{att} & F_{rep}.

The final metric, location uniformity, refers to how uniformly distributed the actors are within the environment. When actors are distributed non-uniformly, as may be found within many natural scenarios such as pedestrian dynamics [131, 64], the size of actor neighbourhoods are likely to vary more significantly. This can be detrimental to the performance of implementations which parallelise the FRNN search such that it creates unbalanced computation, with some actors having significantly larger neighbourhoods to consider. It is not currently possible to strictly control the location uniformity within the Circles model, however, if initialised with a uniform random population, the model progresses slowly towards a steady state whereby agents are in non uniformly distributed clusters.

Visual testing showed that with a unified force dampening argument of 0.05, 200 iterations was required to consistently progress the model from uniform random initialisation to a clustered steady state. As such these two parameters have been used throughout experimentation, to ensure a range of distributions are covered.

Additionally, benchmarks should be performed without visualisation to avoid additional graphics overhead impacting performance results.

3.2 Abstract: Network Model

The network benchmark model is designed to utilise the USP data structure as a multi-map, in a manner analogous to how a simplified lattice site model (used for simulating biological immune systems) operates. Actors are assigned discrete bins and also access data from one or more discrete bins. This removes two elements required of FRNN search, the discretisation of a continuous environment and the iteration of a Moore neighbourhood of bins. The significance of this difference is that the runtime split between data structure construction and access is inverted, with each actor accessing significantly fewer bins and neighbour data.

The impact that improvements to construction time have on overall runtime are primarily dictated by the scale of the queries performed to the USP data structure. Hence, this model best demonstrates the impact of optimisations to construction of the USP data structure and has been used in the assessment of optimisations to the construction stage in Chapter 4.

Due to the SIMT architecture of GPUs, the impact of changing bins can be equally significant compared to that of accessing many items from the data structure. The SIMT architecture enforces synchronisation of warps and blocks such that divergence between threads may leave many threads within the warp or block idle. For this reason, to explore how significant the impact on the overall performance under both techniques is, a model representative of travel through a graph has been utilised.

3.2.1 Model Specification

The model represents a directed graph with uniform connectivity (each vertex has e outward edges, e inward edges). A demonstrative projection of a potential network is shown in Figure 3.2.

Actors traverse the graph, reporting their current edge via the spatial partitioning data structure. Once an actor has surpassed the length of their current edge, they survey all edges from their new vertex, counting the number of actors on each edge. Edges have a common capacity (c) and, if possible, the actor will move to the surveyed edge with the greatest remaining capacity. This process then repeats, with the actor moving through their new edge's length. To reduce the impact of actors completing edge traversal at the same time, actors begin iterating a vertices edge list from a random point[3].

The model was designed to enable a parameter sweep over the number of edges per vertex and size of the actor population to demonstrate the impact of ACS construction across a range of potential discrete applications. The parameters (explained in Table 3.2) of the network benchmark allow it to be used to assess how the performance of spatial partitioning construction affects the total runtime when factors such as actors per bin and edges per vertex are changed.

Parameter	**Description**
a	The number of actors within the environment.
v	The number of vertices within the network.
e	The number of (directed) edges per vertex. This value affects how many bins each actor accesses per timestep.
c	Edge (soft) capacity. This must be high enough to accommodate the other parameters ($c >= \lceil \frac{a}{ve} \rceil$).

Table 3.2: The parameters for configuring the network benchmark model.

The model is inefficient, as the iteration of elements within a bin to calculate size could be read directly from the PBM. However, this model exists to evaluate how modifying the number of bins accessed impacts the division of execution time. In practice, a real-world model would likely contain both more costly model logic and larger data stored per actor in the USP data structure.

Initialisation

Vertices within the network can be implicit. Edges are assigned to vertices in consecutive order, such that the first e edges begin at the first vertex, the second e edges at the second

[3]Our implementation uses a combination of GPU time and thread indexes to generate a pseudo-random index.

vertex and so forth. Therefore the network can be represented as several arrays of length equal to the number of edges.

The arrays represent each edge's destination vertex, length and capacity. The destination vertex can be assigned in any uniformly distributed pattern to essentially shuffle elements. Length is randomly assigned with respect to the actor speed. Capacity is currently a fixed value c, so may be replaced with a constant. Edge destinations are randomly assigned, and each edge must be assigned a length between 1-2.

Actors within the model are represented by current edge index, distance along edge and speed. The speed and starting edge can either be assigned with fixed or randomly distributed values.

Single Iteration

To begin each iteration, actors are stored in the spatial partitioning data structure according to their edge index.

Algorithm 4 Pseudo-code for an individual actor's behaviour during a single iteration of the network model.

```
uint myEdge;
float myEdgeProgress, mySpeed;
uint myVert = myEdge / EDGES_PER_VERT;
myEdgeProgress += mySpeed;
uint nextVert = edges[myEdge].destination;
uint nextEdge = nextVert * EDGES_PER_VERT;
uint minId = UINT_MAX, minCount = UINT_MAX;
for(i = 0;i<EDGES_PER_VERT;i++)
{
    uint edge = nextEdge + i;
    uint count = 0;
    foreach message in bin[edge]
    {
        count++;
    }
    if(count-edges[edge].capacity<minCount)
    {
        minCount = count-edges[edge].capacity;
        minId = edge;
    }
}
if(myEdgeProgress>= edges[myEdge].length)
{
    if(minCount<UINT_MAX && minCount>0)
    {
        myEdge = minId;
        myEdgeProgress = 0;
    }
}
```

Algorithm 4 provides pseudo-code for an actors behaviour within the network model. Each actor increments their distance along edge according to their speed. They then survey the potential following edges leading from their destination vertex, identifying the edge with the most remaining capacity. Following these actions, if the actor has both exceeded their edge's length, and has found a connected edge with available capacity, the actor will switch to that edge and reset their edge progress. This model could clearly be implemented in a more efficient manner, however, it serves to facilitate benchmarking of particular features.

Validation

Firstly, the generated networks should be validated. This can be performed by checking that each vertex has been assigned e edges inwards and e edges outwards. This is important, as it ensures uniform connectivity such that actors should remain roughly uniformly distributed throughout the network. It is not a problem, for purposes of the benchmark, if the random generation of the graph has managed to generate a graph with disconnected sub-graphs.

Secondly, the correct behaviour of actors should be validated. This can be performed by executing a small network (e.g. 10 v, 3 e and 90 a) for multiple iterations (e.g. 100), with stochastic elements removed.

Edge lengths and actor speeds should be initialised to a constant value e.g. 1 and 0.5, respectively. It becomes necessary to temporarily remove the stochastic elements of actor's behaviour. Actors should be initialised in order, such that the first 3 actors begin on edge 0, second 3 edge 1, etc. The pseudo-random behaviour when deciding next edge should use each agent's global thread index.

Now if an equal number of iterations are executed, actors should still be uniformly distributed about the network. This can be tested by counting or producing a histogram of actors per edge for visual confirmation.

3.2.2 Effective Usage

The network benchmark model is primarily concerned with assessing the impact of bin accesses per thread during the query stage of FRNN search. This can be affected by changing the parameter e, which modifies the quantity of edges per vertex.

The other metrics, which may affect the performance of both stages of FRNN search, are actor quantity, bin quantity and actor location uniformity. Actor and bin quantity can be modified directly by changing the total number of actors and edges (via vertex count and/or edges per vertex) respectively.

To affect the actor location uniformity, the initial actor assignment to edges should be modified according to the desired distribution function. Actor travel is designed to encourage uniform travel, such that the distribution should persist unless the edge capacity value is set too low to permit such a high density (this value can just be set to the type's max value if not desired).

The total number of bins accessed can be modified by changing the number of edges per vertex.

3.3 FLAMEGPU Models

The existing benchmark models presented allow isolation of specific parameters to observe how changing their values affect performance. They neglect to account for the impact once applied to a full model, where model logic unrelated to the FRNN search reduces the relative impact of performance improvements. Therefore, the Pedestrian (LOD), Boids & Keratinocyte models (described below), available with FLAMEGPU, which is a general framework for the implementation of MAS on GPUs, have been used to demonstrate the techniques applied to full models.

FLAMEGPU has been modified to support the optimisations present within this thesis, rather than needing to re-implement real-world models independently. Further clarity on optimisation specific implementation details are provided in Sections 4.4.2 and 6.3 where FLAMEGPU has been used. Additionally both of the case studies in Chapter 7 make use of FLAMEGPU.

3.3.1 Pedestrian

FLAMEGPU provides a pedestrian models [91], handled via an implementation of the social forces model in a 2D environment, as described by Helbing and Molnar [72]. There is no configuration for this model as it is calibrated for the scenario and environment, which is representative of an urban transport hub, which is a relatively low density of pedestrians.

The local collision avoidance of pedestrian modelling relies heavily on FRNN search, where the environment is decomposed into a grid and pedestrians survey others within their Moore neighbourhood. As such, it provides a suitable application with which to test the impact of changes to the spatial partitioning data structure on a model utilising FRNN search.

Pedestrian models can be expected to contain 1,000+ actors depending on the scale of the simulation. Similarly, pedestrian densities can vary significantly based on the nature of the environment used.

Figure 3.3 provides a visualisation of the environment, however, benchmarking is carried out without a visualisation so that model performance is not constrained by the requirement of a real-time simulation and visualisation.

3.3.2 Boids

FLAMEGPU also provides an implementation of Reynolds flocking model [136], which provides an example of emergent behaviour represented by independently operating birds

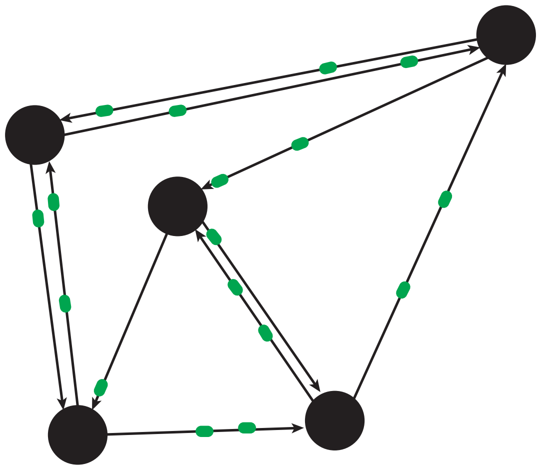


Figure 3.2: A visualisation of how a network model instantiated with 5 vertices, 2 edges in each direction per vertex and 17 actors might appear. Actual edges lengths would normally have greater variance. Vertices do not have defined spatial locations, such that it may not be possible to project generated networks to Cartesian space.

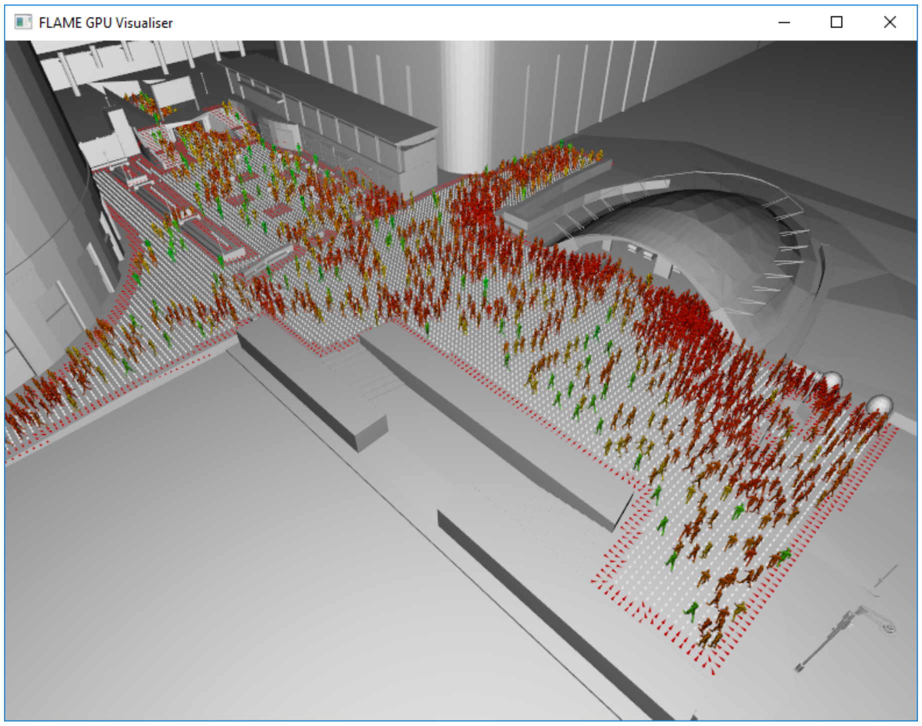


Figure 3.3: A visualisation from the pedestrian model representing an urban transport hub.

in a 3D environment. This model operates in an unstructured environment. Therefore, its population size can be scaled to hundreds of thousands, or millions of actors.

3.3.3 Keratinocyte

Keratinocyte is a cellular biological model representative of Keratinocyte cells, which are the predominant cell found on the surface layer of the skin. This model utilises two instances of the USP data structure, which both operate in a 3D environment, corresponding to inter-cellular forces and bonding behaviour respectively.

The Keratinocyte model was modified slightly to decrease the resolution of the force USP. This both allowed the model to support a higher agent population and improved performance 2x, without affecting the model's operation.[4]

As a particularly resource intensive model, this does not support large agent populations due to the memory limitations of a single GPU.

3.4 Summary

This chapter has contributed two benchmark models, 'Circles' and 'Network', for evaluating the performance of accessing the USP data structure via FRNN queries and direct bin accesses respectively. It has detailed how they can be used effectively to assess the impact that model properties have on the performance of FRNN search and the USP data structure. The Circles model predominantly supports the contributions of Chapters 5 & 6, which investigate the optimisation of FRNN search's query stage. The Network model is primarily used to support the contribution of Chapter 4 which investigates an optimisation for the construction stage of FRNN search.

This chapter has also introduced three FLAMEGPU models. These have been using in Chapters 4 & 6 to contrast between the impact of optimisations on the benchmark models, over a range of properties, and the impact on three full complex system models.

[4]The original force resolution structure had around 400 bins per actor, most of which would therefore remain unused. This was reduced to around 1 bin per actor. It is possible that greater reductions would further improve performance, however, that is outside the scope of this thesis.

Chapter 4

Optimising Construction

Section 2.5 introduced the USP data structure which is used to provide FRNN search. It highlighted the limited approaches to improving the construction of the data structure, which have resulted in an approximate construction. It also raised the open research question; How can modern GPU hardware/features be applied to improve FRNN search?

This chapter proposes that the construction of the USP data structure can be improved, by replacing the sorting algorithm. The proposed optimisation, termed 'Atomic Counting Sort', aims to take advantage of modern improvements to the performance of atomic operations on GPUs. Thus replacing traditional radix sort with Atomic Counting Sort, this has the benefit of reducing work required to construct the PBM.

This chapter provides contribution C1, via the proposed 'Atomic Counting Sort' optimisation for construction of the USP data structure during FRNN search and experiments which demonstrate its impact on performance. These experiments utilise the Network and Pedestrian models, described in Chapter 3, to investigate how a range of properties from the general case of complex system simulation properties, presented in Section 2.5, affect the impact of the 'Atomic Counting Sort' optimisation.

The remainder of this chapter is divided into sections which give the background of the optimisations development, present how Counting Sort is applied atomically, introduce how this can be used to optimise construction, present experiments to be used in assessment of the optimised construction, discuss the results obtained and draw high level conclusions from the work within this chapter.

The following chapter instead looks at how the query stage of FRNN search, proposing an optimisation to reduce divergent code during FRNN queries.

4.1 Justification

Limited research towards FRNN search has targeted the construction of the data structure. Existing approaches have attempted to optimise the sorting to take advantage of the incremental movement of data. However, unlike serial insertion and merge sorts, there is not a parallel sort which is faster when sorting pre-sorted data. Therefore these approaches have often further reduced FRNN search to an approximate method in addition to applying this change.

Sorting is central to the construction of the USP data structure's storage array. As detailed in Section 2.5.2, all items that are to be stored must be sorted according to their containing bin's index. Then the PBM is constructed by detecting the boundaries of bins with a simple kernel, producing two arrays representing the start and end indices of each bin.

Many sorting algorithms have been re-implemented to work on the SIMT architecture of GPUs, such as radix sort, merge sort [142] and quick sort [33]. These have then been further improved, reducing data movement costs with techniques such as improving data locality and cache coherence [21, 143, 147]. Efficient GPU radix sort implementations have been made available through sources such as CUDPP [3], Thrust [25, 5] and CUB [2], leading to its wide adoption. A 2015 study found CUB to provide the greatest sort performance from a selection of libraries providing GPU sort implementations [112].

Counting sort fits neatly into the construction of the USP data structure, as it creates an intermediate representation of the PBM as a by-product of the sorting algorithm. This can then be converted to a single array PBM with a primitive scan operation at less cost than the existing boundary detection kernel. Furthermore, NVIDIA GPU architectures since Maxwell[10] have greatly improved the performance of atomic operations. This chapter explores the impact of applying Atomic Counting Sort to the construction of the USP data structure.

4.2 GPU Counting Sort with Atomics

Counting sort is a non-comparative integer sorting algorithm which operates by counting the number of items of each value, effectively building a histogram and subsequent PBM [40].

Counting sort can be implemented in serial using 3 loops, with best, average and worst time complexities given by $O(n+k)$, where n represents the number of items and k is the size of the key range. Counting sort operates in a similar way to a single pass of GPU radix sort, sorting the entire data set at once, rather than each digit in turn. Radix sort by contrast has the time complexity $O(n+r^w)$, where r refers to the radix (number of buckets) used and w the word length, such that k is replaced by r^w.

Figure 4.1 provides an overview of how Counting sort operates. The algorithm first requires that each item increases the counter assigned to its relevant bin to produce the histogram of bin sizes. As each item contributes to the histogram, the increment operation returns the bin's previous total. This is stored as their index within the bin (offset index). For example, there are two 1s in the initial unordered list, so the histogram bin for 1 is set to 2, and their offsets are set to 0 and 1, respectively.

A prefix sum is then performed over the histogram of bin sizes, cumulatively adding each histogram value. This creates what we refer to as the PBM. To calculate each item's new unique ordered index, the storage offset of the bin containing the item (as retrieved from the PBM) is added to the item's offset (Offset List in Figure 4.1). Items are subsequently reordered by copying them to their new index in a second array (Ordered List in Figure 4.1). For example, the two 1s from the initial unordered list have offsets of 0 and 1, respectively. These are added to the PBM value for 1 (which is 1), to produce their ordered indices of 1 and 2.

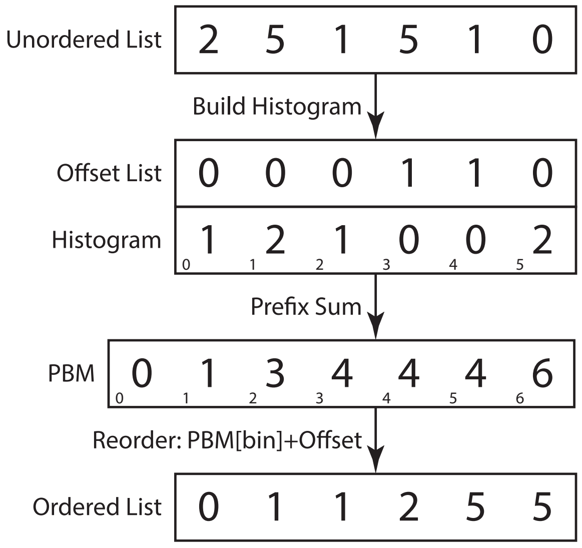


Figure 4.1: A visualisation of the stages of a sample counting sort. A histogram is constructed, as elements contribute to the histogram they store the previous value into the offset list. A prefix sum is then performed across the histogram to produce the PBM. Each element is then relocated according to the PBM value of their containing bin and their value from the offset list.

When implemented in parallel, such as on GPUs, counting sort requires a mode of conflict resolution to construct the histogram, to ensure that items sharing a bin are assigned unique indexes within the bin.

There are two main strategies for conflict resolution on GPUs, which can be classified as push and pull [166]. The push strategy requires competing threads to write to a global shared variable, which can be performed with atomic operations (such as max) to ensure the writes do not conflict. A synchronisation must be performed prior to threads then accessing the final result. Alternatively, the pull strategy requires competing threads to write their decision to a unique location within an array. Subsequently, this array can be scanned to produce a resolution that can be read back by each thread.

The key difference between these techniques is that atomic operations (used in the push conflict resolution strategy) are considered non-deterministic, where, in the case of counting sort, the order of items of equal value may differ between repeated sorts. When applied to construction of the USP data structure, however, this has negligible impact, as the items within each bin have no required order.

NVIDIA's Maxwell [10] and Pascal [119] architectures have both improved the performance of atomic operations. These changes enable atomic resolution at every level of cache hierarchy. An experiment was carried out, producing Figure 4.2, to demonstrate how the performance of ACS responds to varying contention of the data being sorted on a Pascal architecture GPU. These results demonstrate that ACS favours sorting data of low contention, where fewer items share the same value. In particular, where the range is greater than 2^{11}, the performance of ACS is similar to that of CUB. This equates to 512 or less threads competing. This value is far higher than necessary for performance within the USP data structure, for example, the densest SPH case of 80 neighbours [167] would evaluate to 19 items per bin if uniformly distributed [1]. Whilst it is possible that other applications, such as crowd modelling, may achieve greater densities, it is unlikely they would come close to surpassing 512 items per bin.

In contrast, to utilise a pull approach, a scan would need to be completed for each of the data structure's bins. Due to the presumed low number of items per bin, the overhead of handling hundreds of small scan operations would likely harm performance.

As demonstrated in Figure 4.3, a prefix sum can be applied to the intermediate histogram produced during ACS to generate a single array PBM. This replaces the boundary detection kernel with a generic parallel primitive operation available from a highly optimised library such as CUB [2] and produces the single array PBM which has half the memory footprint.

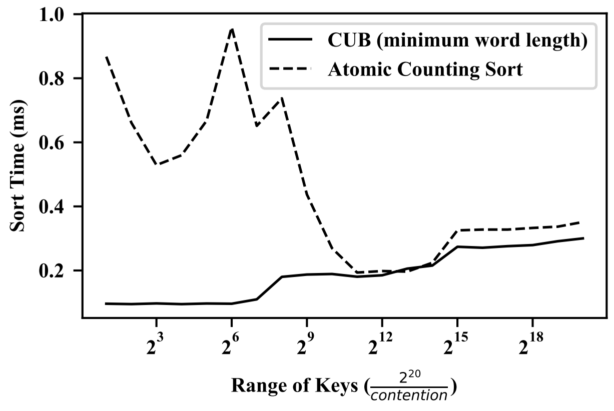


Figure 4.2: **Radix sort vs ACS - Variable Contention:** The runtime of both CUB radix sort (limited to minimum required word length) and atomic counting sort when applied to a dataset of 1048576 (2^{20}) keys. The range of key values varies from 2 to 1048576. This result was captured with a Titan X (Pascal) using CUB 1.8.0 and Cuda 9.0 on Windows 10.

4.3 Atomic Counting Sort Construction

Figure 4.4 illustrates how the PBM is constructed from the neighbour data and neighbour bin index arrays. First, an array of length equal to that of the neighbour data array is created. In this array, each element from the neighbour data array has its containing environment bin's index stored (Neighbour Bin Index Array in Figure 4.4). Equation 4.1 can be used to transform a spatial coordinate located within

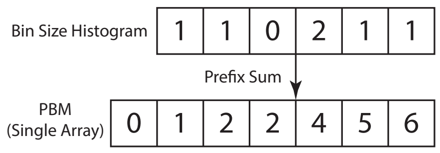


Figure 4.3: The bin size histogram, produced as a by product of atomic counting sort, may be used to produce a single array PBM.

[1] $80\frac{6.45}{27}$, where 6.45 represents the multiplier from 3D radial neighbourhood volume to 3D Moore neighbourhood volume, and 27 the number of bins within the 3D Moore neighbourhood

the environment to its corresponding environment bin's coordinate ($gridPos$). This can then be transformed to the bin's 1-dimensional index using either equation 4.2 for a 2D environment, or equation 4.3 for a 3D environment.

$$gridPos = \left\lfloor gridWidth \frac{envPos}{envWidth} \right\rfloor \tag{4.1}$$

$$gridId_{2D} = (gridPos_y * gridWidth_x) + gridPos_x \tag{4.2}$$

$$\begin{aligned} gridId_{3D} = {} & (gridPos_z * gridWidth_y * gridWidth_x) \\ & + gridId_{2D} \end{aligned} \tag{4.3}$$

Next, the neighbour bin index array is used to calculate the offset of each bin's storage (the PBM). This is achieved by first producing a histogram representing the number of messages within each bin. An exclusive sum scan is then performed over the histogram to produce the single-array PBM. For the PBM to be produced correctly, the PBM must have a length one greater than the total number of bins, so that the final value in the PBM denotes the total number of neighbour data.

Finally, the neighbour data array is sorted, so that neighbour data is stored in order of their environment bin indices. This can be achieved using a primitive pair sort, such as that found within CUB [2], and a kernel to subsequently reorder the neighbour data.

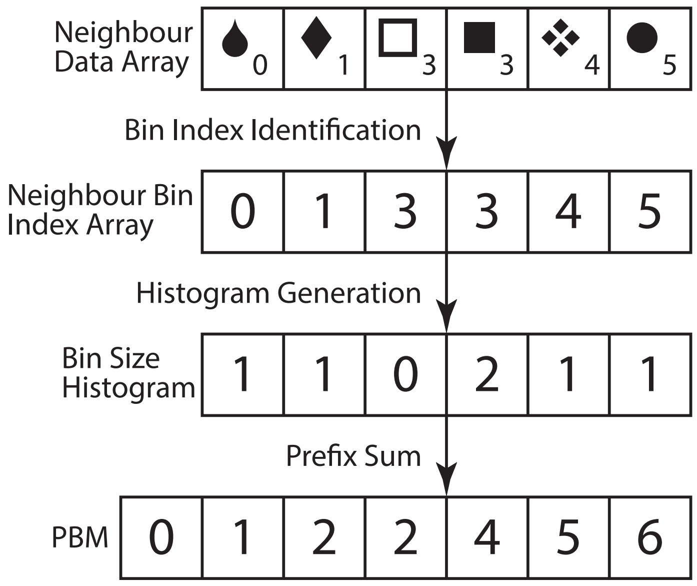


Figure 4.4: The arrays used to generate a PBM.

4.4 Experiments

The experiments presented within this section compare the ACS construction technique described in the previous section with the original construction technique described within Section 2.5.2.

Original Algorithm

- Hash Items
- Radix Sort Hashes
- Reorder Messages & Construct PBM

ACS Algorithm

- Atomic Histogram
- Scan (Construct PBM)
- Reorder Messages

Firstly, the techniques were tested with quantitative experiments of the construction algorithms in isolation. The purpose of these experiments is to evaluate them under a range of configurations, not inherently representative of real-world applications, but covering the experimental parameter space. This style of experiment provides a greater understanding of the circumstances which most favour each technique and how their performance is affected as parameters change (e.g. scaling of population size or density).

Secondly, the techniques are applied to models within FLAMEGPU representative of real-world applications. These were earlier defined in Chapter 3. The purpose of these experiments is to demonstrate that performance features identified in the previous quantitative tests are observable when applied to real configurations. Furthermore, these experiments are able to demonstrate the impact an optimisation to construction has on the overall performance of a model.

The following two sections explain the experiments of each style that have been performed.

4.4.1 Quantitative Experiments

Two levels of abstraction have been used to quantitatively assess the impact of ACS construction.

Firstly, the most abstract assessment considers how performance scales in response to the number of bins. At the code level, the only two parameters which are likely to significantly affect the performance of the algorithms are the size of the actor population and the number of bins which they are divided up between.

Secondly, building on the abstract assessment, the quantity of actors within bins is assessed. It is apparent (from Figure 4.2) that the distribution of actors among the bins is likely to affect performance of construction, especially in cases of high contention (many actors per bin) which are unfavourable to ACS.

Bin Quantity

These experiments use a standalone implementation[2] specifically for the purpose of timing the construction algorithms and their constituent stages in isolation.

A fixed size actor population is used for construction over a large range of bin quantities. In each instance the actors are uniform randomly distributed among the bins.

This produces a trend for each technique as the quantity of bins increases. Furthermore, this trend can be separated into its constituent components, allowing identification of how the individual stages of construction are affected.

Actor Density

These experiments were carried out using the same standalone implementation as the previous experiment. However, in this case, only the overall performance of each construction technique is being considered.

Both the parameters of actor population and actors per bin are scaled. To ensure a uniform density, the total number of bins is decreased to accommodate the value In each instance the actors are uniform randomly distributed among the bins.

By then comparing the performance of the two techniques, a heatmap of their relative performance across the two dimensional parameter sweep is produced. This allows identification of bands of favourable configurations for each technique, using parameters more easily mappable to specific forms of MASs.

4.4.2 Applied Experiments

The USP data structure can be accessed in two forms: individually accessing bins (like a multi-map), or performing a FRNN query which accesses multiple bins (as is the focus of this thesis). These two cases have different access patterns, which are likely to affect the relative significance of construction with respect to the overall runtime of a given model. As such, the experiments described below assess the impact to both forms of access. A more detailed explanation of these models can be found in Chapter 3.

Network Model

These experiments were carried out using the main spatial partitioning test suite attached to this thesis[3], using the Network benchmark model. This model is representative of communication about a network, allowing accesses to the data structure to be scaled, varying the impact of construction on overall runtime.

[2]`https://github.com/Robadob/sp_con_test`

[3]`https://github.com/Robadob/SP-Bench`

Feature	**Bin Count**	**Actors Per Bin**	**Neighbourhood Volume (2D)**	**Neighbourhood Volume (3D)**
Leftmost	5,000	200	629.37	837.21
Rightmost	1,000,000	1	3.15	4.19
Intersection	780,000	1.28	4.03	5.36
ACS gradient change	54,000	18.52	57.28	77.53

Table 4.1: The Bin Counts from Figure 4.5 converted to relative representations.

Pedestrian

This pedestrian model is available within the main FLAMEGPU repository[4], under the name `PedestrianLOD`, although replacing the environment requires a tool not available within the public FLAMEGPU repository. FLAMEGPU can be built with or without the ACS construction by toggling the pre-processor macro `FAST_ATOMIC_SORTING`. This model is a functional pedestrian simulation of an urban environment, validating that the optimisation persists when applied to a full model with more costly logic and FRNN search.

4.5 Results

Within these experiments, the original technique, Radix sort (provided by CUB, limited to shortest suitable radix) and a boundary detection kernel, has been compared against the proposed technique, atomic counting sort and prefix sum. These are referred to as 'Original' and 'Atomic' (and ACS) respectively throughout the results.

4.5.1 Quantitative Experiments

Bin Quantity

These experiments were performed using an Nvidia Titan-V GPU under CUDA 9.1 on Ubtunu 16.04.6.

For the quantitative experiments a total of 200 configurations, uniformly distributed throughout the tested range (e.g. bin count $1 - 10^6$), were each executed 200 times. In order to remove outliers and stochastic noise from the results, the minimum time measured from each of the 200 executions has been used.

Figure 4.5 shows the performance scaling when executed with 1 million actors. The original technique has constant time performance, on 3 tiers, increasing at 260,000 and 525,000 bins respectively. In contrast, the performance of ACS construction quickly degrades till around 54,000 bins, after which it continues to degrade at a much slower linear gradient. As such, after 780,000 bins, the runtime of ACS construction surpasses that of the original technique.

[4] `https://github.com/FLAMEGPU/FLAMEGPU`

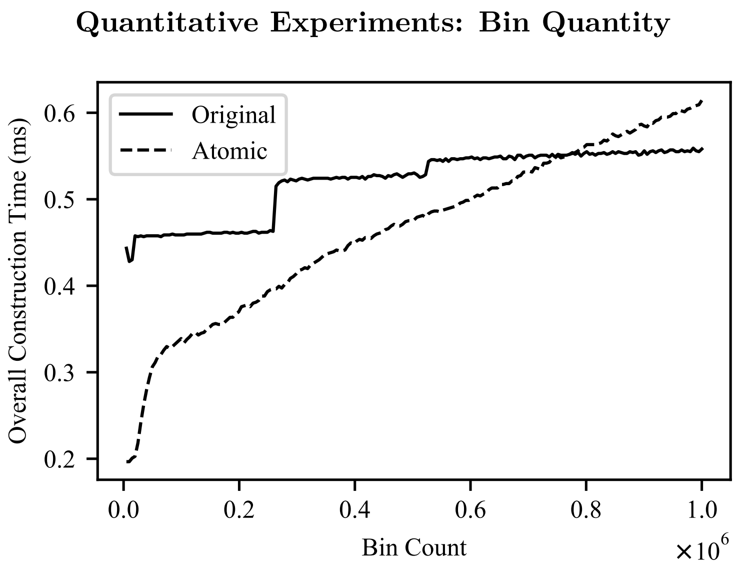


Figure 4.5: **USP Construction - Variable Contention:** The performance of USP construction with 1 million unsorted actors and how this changes as the quantity of bins rises from 5000 to 1 million.

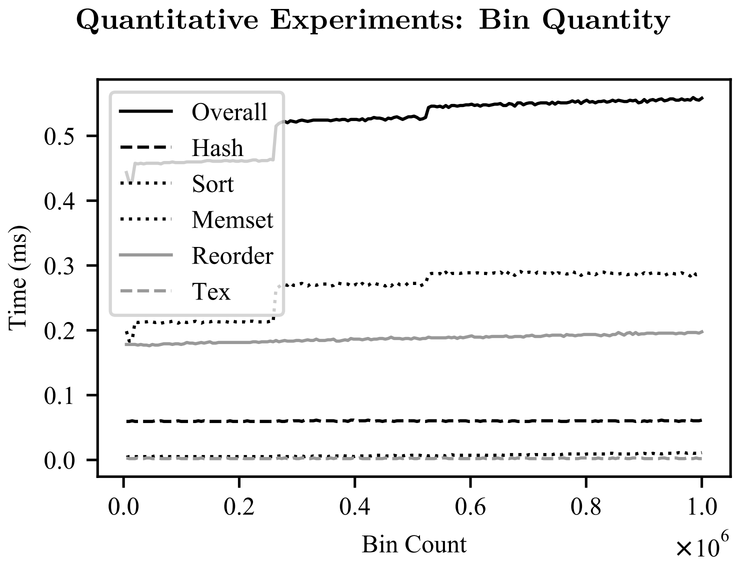


Figure 4.6: **Original Construction - Components (unsorted):** The performance of the constituent components of the original USP construction technique with 1 million actors, and how this changes as the quantity of bins rises from 5000 to 1 million.

Table 4.1 puts these numbers in context, using the equations from Section 6.1[5] to calculate the corresponding approximate actors per bin and neighbourhood volumes. This demonstrates that the intersection (in Figure 4.5) occurs at a very low density if translated into 2D or 3D environments, with less than 5 neighbours appearing within FRNN search's radial neighbourhood. Furthermore, the ACS construction gradient reduces at an approximate radial neighbourhood size of 77.53 actors. In context, literature shows that 80 results per query is an upper bound for SPH models [167], whereas crowd simulations can see this value both much higher and lower dependent on the the behaviour of individuals within the model.

Figures 4.6 and 4.7 build upon these results, by considering the performance of the constituent processes of both the original and ACS construction techniques. The legends for these figures list the components in the order of execution, with the overall execution time added to the start. The copy to texture cache operations and primitive operations memset and scan have negligible cost, however are included for completeness. The full Original and ACS construction algorithms are outlined in Sections 2.5.2 & 4.3 respectively.

From Figure 4.6 it is clear that each process, aside from the Radix sort, has an almost constant runtime. The points at which the sort runtime increases, are where the word length of radix sort increase to accommodate 2^{18} and 2^{19} bins, respectively.

In contrast, the primary contributors to the runtime of ACS construction is the reorder kernel, which performs 3 reads from global memory to calculate the mapping, before copying the messages to their new sorted position. The performance of the atomic histogram remains constant until 560,000 bins after which it begins to slow, however, due to the low density here it should not be a concern.

When executed with alternate actor populations (not shown in figures), the point of intersection moves. The trend of the original technique remains mostly constant. Its time is clearly linked to the total number of actors. In contrast, the performance of ACS construction is closely linked to atomic operations and actors per bin, which causes its performance to degrade with a steeper gradient as the actor population increases. 100,000 actors causes ACS construction to remain at around 1/3 of the runtime of the original technique throughout. On the other hand, 5 million actors places the intersection point at 200,000 bins. Digging into the underlying processes of ACS construction shows that the reorder kernel again dominates the runtime, with the atomic histogram's performance remaining constant until around 590,000 bins, very close to the earlier breakdown seen in Figure 4.7.

These experiments have highlighted a threshold of around 575,000 bins, after which performance of the atomic histogram begins to degrade. This value is likely a consequence of device specific properties combined with atomic contention.

Additionally these experiments have demonstrated that the most significant component

[5]The area of the Moore neighbourhood is 2.86x and 6.45x larger than the area of the radial neighbourhood in 2D and 3D respectively.

Quantitative Experiments: Bin Quantity

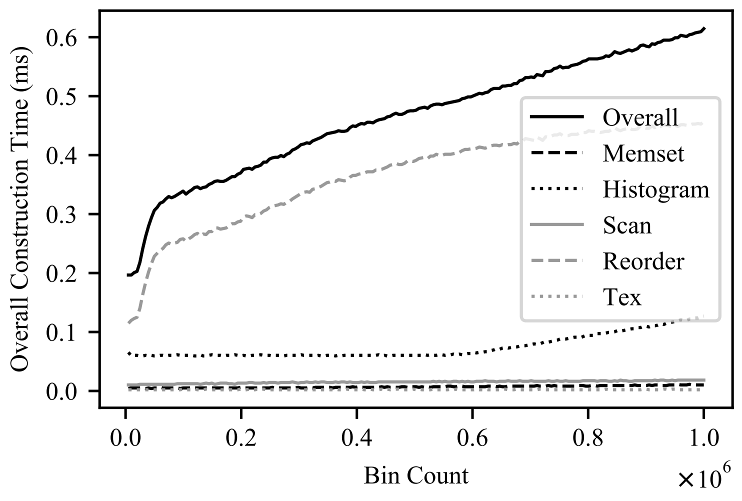


Figure 4.7: **ACS Construction - Components (unsorted):** The performance of the constituent components of the atomic counting sort USP construction technique with 1 million actors, and how this changes as the quantity of bins rises from 5000 to 1 million.

Quantitative Experiments: Bin Quantity

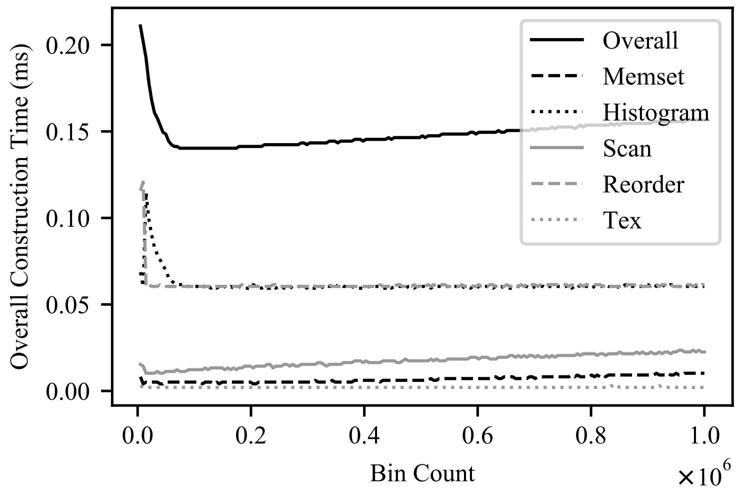


Figure 4.8: **ACS Construction - Components (sorted):** The performance of ACS construction with 1 million sorted actors, and how this changes as the quantity of bins rises from 5000 to 1 million.

of ACS construction is the reorder kernel, due to its scattered memory accesses which quickly bottleneck the GPUs caches, rather than the atomic histogram. In practice, when applied to a model, actor movement is incremental, such that the level of scatter should be heavily reduced. After the first iteration of a model, actors should be sorted. From here they often only move incrementally, so the number of actors out of order should be significantly less than the uniform random allocation of this experiment. Figure 4.8 demonstrates this by reproducing the experiment behind Figure 4.7 with sorted actors. Therefore the earlier Figure 4.7 represents the worst-case, whereas Figure 4.8 with sorted actors represents the best-case. In this best-case it is visible that the runtime of ACS construction is less than 50% of the original technique throughout. The runtime of Reorder is now equal to Histogram throughout. However, as noted in Section 4.2, ACS has no guarantee on the sort order of items of the same value, therefore it is likely that data movement is still occurring, albeit at a more local and cache friendly scale. It can thus be assumed that if the memory footprint of bins increases, e.g. that the number of actors within each bin increases, this may become more significant.

Actor Density

These experiments were performed using an Nvidia Titan-V GPU under CUDA 9.1 on Ubtunu 16.04.6. A total of 288 configurations, uniformly distributed between the minimum and maximum actor population sizes and actors per bin tested were each executed 200 times. In order to remove outliers, the minimum time measured from each of the 200 executions has been recorded.

In order to allow testing of all configurations, as the environment must be a single coherent rectangle, bins were organised as a single strip. As actors are static, this only impacts the hashing arithmetic, hence leaving the measured performance unchanged.

As with the previous experiment results are presented in Figures 4.9 and 4.10 for sorted and unsorted actors, respectively, to demonstrate the effective best and worst cases.

Initially, it is visible from both these figures that ACS construction outperforms the original technique in all of the tested cases, potentially only falling behind for the most extreme combination of high actor population and low density with an unsorted population. This is in agreement with results from the previous experiment, whereby Figure 4.6 demonstrated the same switch when density became very sparse. Although, notably in this case, these extreme combinations are the only instances whereby the original implementation's radix sort requires an increased word length of 2^{18} (hence harming the performance of the original technique, otherwise linked only to actor population).

Looking closer, Figure 4.9 has a blue central band, whereby the most performant configurations are found. Around two thirds of the configurations tested fall within this blue band, representative of ACS construction completing in 35-55% of the original techniques runtime. This shows that when handling unsorted actor populations, as the scale of the actor population increases, so does the required density for the most significant perform-

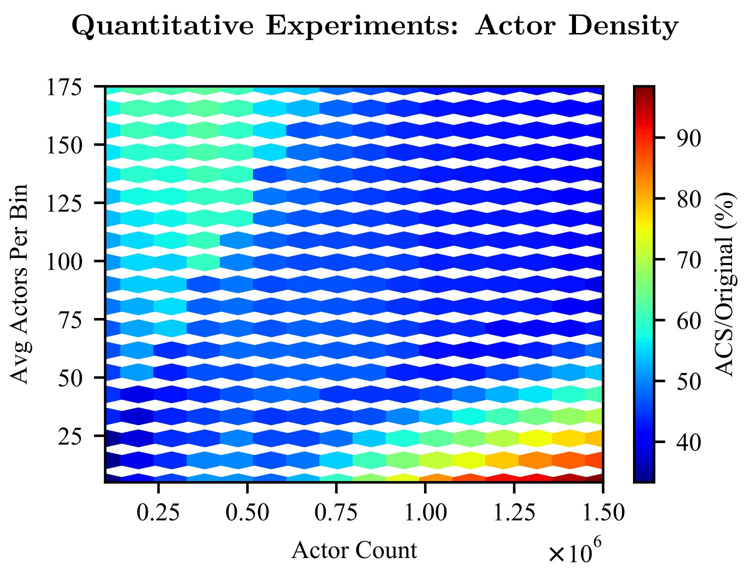


Figure 4.9: **ACS Construction Speedup (unsorted):** A 2D parameter sweep across a range of population sizes and densities, demonstrating the configurations whereby ACS construction performs best for unsorted actor populations, relative to the original technique.

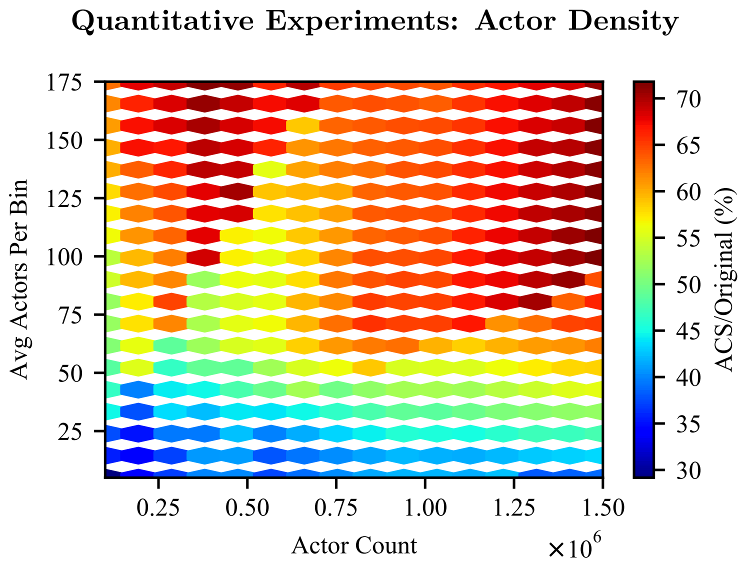


Figure 4.10: **ACS Construction Speedup (sorted):** A 2D parameter sweep across a range of population sizes and densities, demonstrating the configurations whereby ACS construction performs best for sorted actor populations, relative to the original technique.

ance improvements.

The sorted actor populations represented in Figure 4.10 again show ACS construction outperforming the Original construction in all cases and by a wider margin than with unsorted data. In particular, it displays a similar distinct pattern in the same position as Figure 4.9's blue band. To better understand this band, Figure 4.11 provides a cross section of Figure 4.10, for configurations with 99.4 actors per bin. Here it is clear that the performance of both techniques is almost linear, however, minor shifts in gradient lead to the band seen in Figure 4.10. Given the similar placement of the band in Figure 4.9, it is likely a sign that the performance of the original technique scales better after the first edge of the band (e.g. (0.25, 70) to (0.7, 160)). As seen earlier in Figure 4.8, the sorted actor population in Figure 4.10 benefits both techniques by reducing the cost of the Reorder, however, ACS appears to benefit more.

These experiments have further highlighted the significant impact that sorted data has on the relative performance of ACS construction. Furthermore, they have highlighted an inconsistency between relative performance on sorted and unsorted actors. For sorted populations the best case appears to be the lowest density throughout. In contrast, for unsorted populations the worst case is the combination of high population and low density. The following applied experiments handle the movement of actors and as such can provide a better understanding of how significant the rate of change is.

4.5.2 Applied Experiments

The faster that actors move between bins of the environment, the further the actors will move from a sorted state each step. The quantitative results have shown that this is an important performance factor for construction of the USP data structure. Additionally, the higher the dimensionality of the environment, the higher the potential for an actor to move. This is due to the need to reduce the dimensionality of the environment from two or three dimensions into a single dimension for storage.

By performing the following experiments, representative of actual MAS, an understanding can be gained of where these models place the rate of change between the two previously identified best and worst case scenarios.

Network Model

These experiments using the Network model (introduced in Section 3.2), were performed using an Nvidia Titan-X (Pascal) GPU under CUDA 8.0 on Windows 10. A total of 320 configurations uniformly distributed between the minimum and maximum actor population sizes and edges per vertex were each executed 100 times. In order to reduce outliers the average step time has been recorded.

A parameter sweep was performed across both edges per vertex, a proxy for the number of bins accessed by each thread, and actor population size. Two configurations were used,

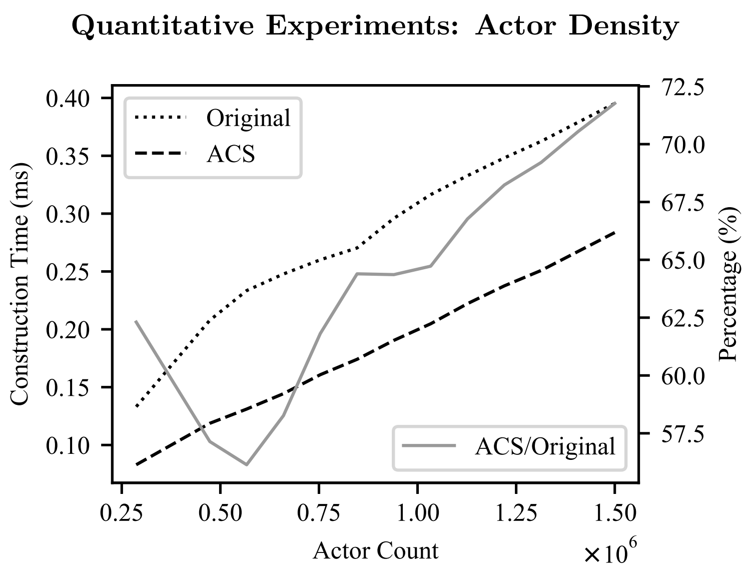


Figure 4.11: **ACS Construction Speedup (sorted) - Cross Section:** A cross section of Figure 4.10 with an average of 99.4 actors per bin.

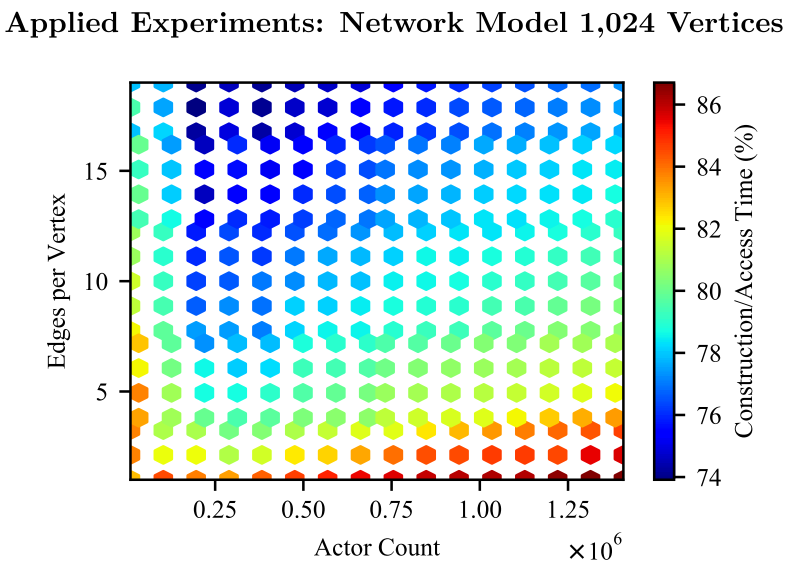


Figure 4.12: **ACS Network Model - Construction/Query Time (1,024 vertices):** A 2D parameter sweep of the network model under ACS construction when initialised with 1,024 vertices, demonstrating how the relative cost of construction changes according to the actor count and quantity of bins accessed per step.

one with 1,024 vertices (Figures 4.12 and 4.13) and one with 16,384 vertices (Figure 4.14) in order to cover a range of densities and bin counts representative of the general case. The number of vertices multiplied by the number of edges per vertex provides the total number of bins within the USP structure. By only modifying the edges per vertex (bin's accessed count) and maintaining a static vertex count, the total messages accessed should remain constant (calculated as actor count divided by vertex count), allowing the focus of results to be on the impact of the number of bins accessed.

Figure 4.12 demonstrates the significance of construction (within the network model). The results show that the relative cost of construction decreases fastest (as the edges per vertex increases) at around 250,000 actors. This suggests that construction scales more favourably than accessing the data structure at lower actor populations. As the actor population is increased, this balance shifts more in favour of accessing the data structure, as the relative cost of construction declines at a slower pace. However, the network model's logic is a simplified version of a real application, for the purposes of scaling the number of bins accessed, so the cost of accesses may be higher in practice.

In Figure 4.13, a fairly smooth gradient starts from the top left corner (few actors, high bin accesses) and sweeps across to the bottom right corner (high actors, low bin accesses). The peak improvement observed is around 225% speedup throughout the lowest actor population tested. The original construction technique has outperformed ACS construction in the cases of highest contention (high actors, low bin accesses, larger bins). However, overall, ACS construction was faster in 77% of cases tested (245/320).

When the total number of vertices is increased to 16,384, as shown in Figure 4.14, it appears as though the same pattern as in Figure 4.13 persists, with the smallest actor count providing the optimal case to ACS construction. The increased vertex count and subsequent bin count, however, leads to ACS construction's improved performance persisting through to the highest tested actor population size (1.5 million), where the ACS construction model still executes with 150% speedup. This can be seen as a consequence of reduced overall contention (more vertices leads to more edges, less actors per edge). All the tested configurations in Figure 4.14 show that ACS construction improved over the Original technique, with only the configurations of highest contention (high actors, few edges per vertex) close to parity.

This experiment has primarily considered the impact that increasing the number of bins accessed has on the construction's contribution to the overall runtime. By maintaining a near constant total number of messages accessed, it has allowed assessment of bins accessed independent of additional influences to overall runtime.

The core difference demonstrated between Figures 4.13 and 4.14 is that with a higher total number of vertices, and therefore lower number of messages per bin, the improvement of ACS construction over the original technique is greater. This is in agreement with earlier experiments, where it was shown that the lowest density configurations were most favourable.

The impact of dynamic actors within the model (rate of movement between bins) has

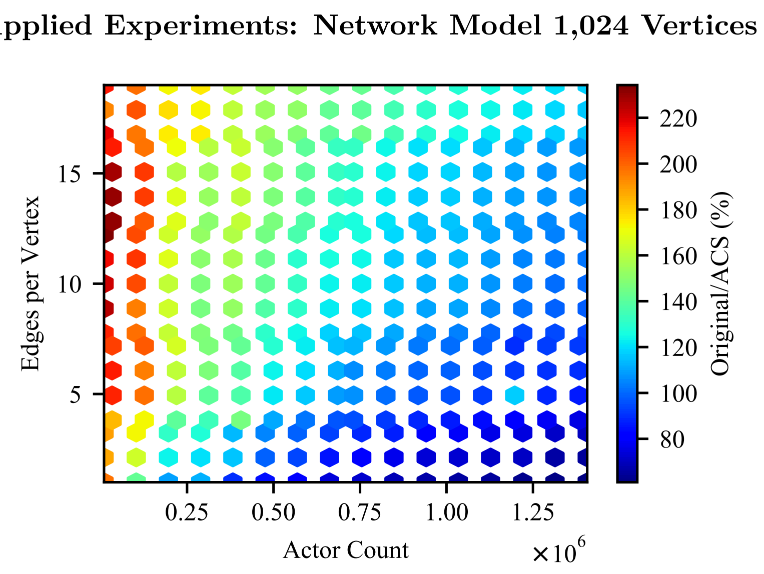


Figure 4.13: **Network Model - ACS Speedup (1,024 vertices):** A 2D parameter sweep of the network model when initialised with 1,024 vertices, demonstrating where ACS improves over the original construction technique, in terms of overall runtime.

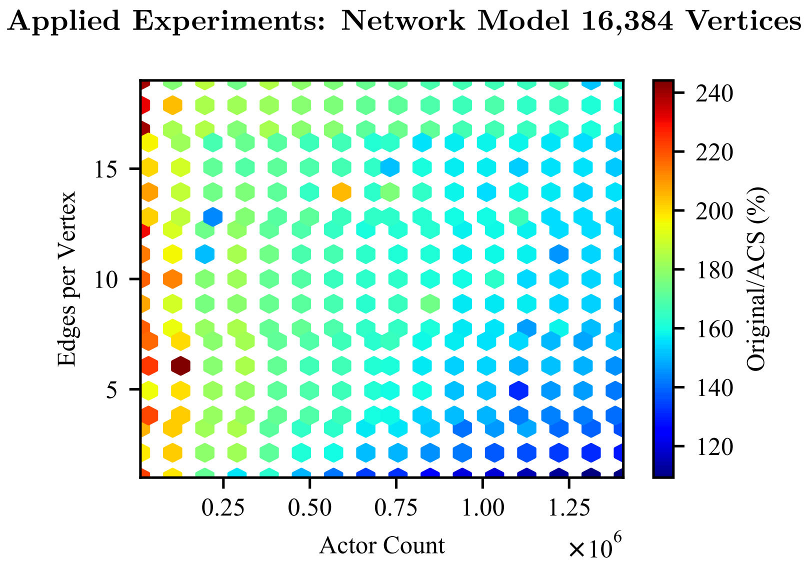


Figure 4.14: **Network Model - ACS Speedup (16,384 vertices):** A 2D parameter sweep of the network model when initialised with 16,384 vertices, demonstrating where ACS improves over the original construction technique.

not harmed the relative performance of ACS constriction over the original technique. This suggests that the rate of change is low enough to prevent significant loss of order between each step of the simulation.

Pedestrian

This experiment was performed using an Nvidia Titan-X (Maxwell) GPU under CUDA 8.0 on Windows 10.

When filled, the model operates with a peak of approximately 1950 pedestrian actors, with a variety of densities ranging up to 30 pedestrians per neighbourhood search. When comparing the time to execute 10,000 iterations of the model, the original technique performed in 7,555ms, whereas ACS performed in 7,343ms. Therefore, ACS provided a modest 2.8% improvement of the total runtime.

Due to the expensive model logic and larger neighbourhoods (FRNN search operates by searching a Moore neighbourhood of bins), the impact of construction on the overall runtime is less significant than in the network case.

4.6 Conclusion

The work taken to apply ACS to the construction of the USP data structure has demonstrated that the performance of construction can be improved as significantly as 3x, dependant on several important properties, providing Contribution C1.

Firstly, the order of the actor population before construction has a significant impact on the performance of both techniques, due to a reduction in widely scattered memory accesses. ACS construction, however, benefits more significantly from ordered actors, as the original technique has greater memory traffic, due to the border detection operation which is performed in the same kernel as message reordering. The impact of this is that ACS construction, whilst almost always more performant than the original technique, is likely to provide a greater improvement where actors are moving between bins at a slower rate.

Secondly, through isolation of experimental parameters, the optimal cases for ACS construction have been identified. Low density environments, with few actors per bin provide the optimal case for ACS construction over the original technique. Although it favours lower actor populations (achieving 3x improvement with 100,000 actors), even when tested as high as 1.5 million actors 2.5x improvement was still achieved.

Notably, the maximum density tested (175 actors per bin) is significantly higher than is likely to be found in real complex system simulations. For a 2D model this would equate to an average radial neighbourhood of 550 neighbours, and 3D equates to 732 neighbours. Therefore, the most favourable range of 1-45 actors per bin, which provides 2x improvement or greater throughout, still provides support for densities as significant as 188 neighbours per radial neighbourhood. This is more than enough to satisfy the requirements of SPH, which has a suggested upper bound of 80 neighbours [167]. Neighbourhoods as populated as 188 actors may only be achieved in the most diverse pedestrian models, which pair sparsely populated open areas alongside high density crowds, due to their unsuitability for the uniform discretisation provided by USP.

Furthermore, the experiments have shown the impact of the ACS construction optimisation on the overall performance of simulations. Construction under the simple network model was around 74-90% of runtime in all tested scenarios. However, when applied to a full pedestrian model, with significantly more expensive computation, the impact of ACS construction was a modest 1.03x (two decimal places) speedup to the total runtime. This is attributable to the high cost of the query stage under FRNN search, where a greater number of bin accesses occur, reducing the relative cost of construction.

As identified at the start of this chapter, improvements to construction will have a greater overall impact to models such as biological lattice site, which instead use USP as a multi-map. However, as absolute improvements to the construction time are present in all configurations within the general case that were tested, a minimum of modest improvements should be present to the overall runtime of any model that the optimisation is applied to.

In addition to the results presented in this chapter, FLAMEGPU was amended during the research (2018 commit `5146a95`) to ensure that when using Radix sort the minimum suitable word length is used. This change was necessary to ensure fair assessment between construction techniques and was fed back into the main release of FLAMEGPU. The stepping visible in Figure 4.6 (Page 73) demonstrates the impact of increasing the word length from 17 to 19.

The next chapter instead focuses on how the query stage of FRNN can be optimised by reducing divergent code, proposing an optimisation to the query stage of FRNN search.

Chapter 5

Optimising Queries: Reducing Divergence

Section 2.5 addressed the USP data structure and that the two stages of ‘construction’ and ‘query’ are required to provide FRNN search functionality. In particular it highlighted areas of investigation for optimisation, including the impact of divergence on the performance of the query stage. This raises the open research question; Is it possible to reduce divergence during queries without reducing FRNN queries to an approximate method?

In this chapter it is proposed that the query stage of FRNN search can be improved by reducing the number of bin changes when iterating a Moore neighbourhood. The proposed optimisation, termed ‘Strips’, aims to reduce the quantity of divergent code. Divergence between threads is costly under the SIMT architecture of GPUs, and hence this optimisation has the potential to improve the performance of FRNN queries.

This chapter provides contribution C2, via the proposed ‘Strips’ optimisation for the query stage of FRNN search and experiments demonstrating its impact on performance. These experiments use the Circles benchmark model, described in Section 3.1, across a wide range of properties applicable to the general case, identified in Section 2.5, in order to assess the applicability of the ‘Strips’ optimisation.

The remainder of this chapter is divided into sections which analyse the FRNN search’s query algorithm, present the optimisation of the query algorithm, called Strips optimisation, present experiments to be used for assessment of the optimised query algorithm, discuss the results gained from the experiments, and draw conclusions from the work within this chapter as a whole.

The following chapter builds on this work, by proposing a further optimisation targeting redundant memory accesses during the query stage. These two optimisations are compatible, and can be applied simultaneously.

5.1 Justification

FRNN search operates by decomposing a continuous environment, containing spatially located data, into a regular grid. The query stage returns all items within a fixed radius of the search origin. The search origin is mapped to its bin within the regular grid to identify the Moore neighbourhood of bins which may contain items within range. A distance test between the search origin and location of each item in these bins is performed to decide the results of the query. A general query algorithm for FRNN search is outlined in Algorithm 5.

Algorithm 5 Pseudo-code showing the FRNN search's query algorithm for a two dimensional environment, before optimisation.

```
vector2i absoluteBin = getBinPosition(origin)
loop -1<=relativeBin.x<=1:
  loop -1<=relativeBin.y<=1:
    selectedBin = absoluteBin + relativeBin
    if selectedBin is valid:
      hash = binToHash(selectedBin)
      loop i in range(PBM[hash],PBM[hash + 1]):
        handle(neighbourData[i])
```

The algorithm simply consists of several nested loops to iterate bins of the desired Moore neighbourhood, containing a final loop to iterate items within the current bin. With an understanding of the USP data structure's memory layout from Section 2.5.1, it is clear that several of the bins accessed can have their data stored contiguously in memory (if messages are sorted). It is proposed that contiguously stored bins could be accessed as a single strip. This may reduce divergent code where neighbouring threads change bins at different times and remove memory accesses for the shared boundaries. This chapter will explore the possibility of this approach using a minimal case implementation.

FRNN Query Within MAS Framework (FLAMEGPU)

FLAMEGPU, as described in section 3.3, has been used for experiments with functional MAS within this thesis. It has an alternative implementation for handling bin changes during an FRNN search's query, compared to Algorithm 5 from the previous section, which may also benefit from the proposed Strips optimisation.

The reason FLAMEGPU uses an alternate implementation is to abstract away the complexity of the query from the user of the framework, allowing them to iterate all items within a Moore neighbourhood via a single loop. Within FLAMEGPU the items accessed via FRNN queries are messages for spatial communication between agents (actors).

Under the FLAMEGPU technique for FRNN queries, the user calls an init function (`get_first_NAME_message`), which returns a structure containing both the first message and search status information. To access subsequent messages the user must call the next message function (`get_next_NAME_message`), passing the last returned structure. This then retrieves the subsequent message and updates the search status information. Once the returned structure holds no message, the search is complete.

Algorithm 6 The algorithm used by FLAMEGPU to calculate the next bin during an FRNN search's query of a 2D environment.

```
boolean nextBin(ivec2* relative_cell)
{
        if (relative_cell->x < 1)
        {
                relative_cell->x++;
                return true;
        }
        relative_cell->x = -1;
        if (relative_cell->y < 1)
        {
                relative_cell->y++;
                return true;
        }
        relative_cell->y = -1;
        return false;
}
```

Within the next message function, first it is checked to see whether all messages within the current bin have been returned. If not, the next is returned, otherwise the algorithm shown in Algorithm 6(FLAMEGPU's next bin operation) is performed to identify the next bin. It should be clear from this algorithm that FLAMEGPU also performs these bin accesses individually and may benefit from handling neighbouring bins as a single strip, as this would reduce the number of times the code in Algorithm 6 must be called.

5.2 Strips Optimisation

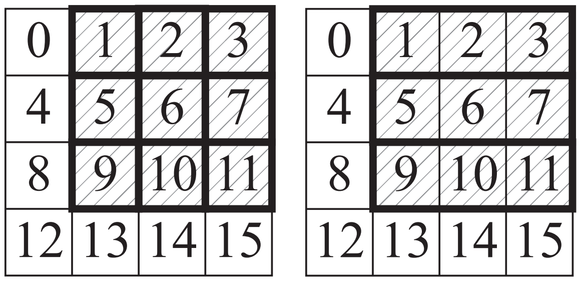


Figure 5.1: Expanding on the Moore neighbourhood from Figure 2.10 (on page 42). The bold borders denote the individual accesses to bins in the original technique (left) and with the Strips optimisation (right).

It is anticipated that changing bins adds latency to the query process. As such it is proposed that the number of bin changes can be reduced by treating bins within a query's Moore neighbourhood, which are stored contiguously in memory, as a single strip. In two dimensions, the Moore neighbourhood of a query's origin's containing bin consists of a 3×3 block of bins. These nine bins exist as three strips of three bins, where each strip's storage exists contiguously in memory. In three dimensions, the 27-bin Moore neighbourhood is formed of nine strips of three bins. Within a strip, only the first bin's start index and the last bin's end index need to be loaded from the PBM to identify the range within the neighbour data array that contains the neighbour data from each of the strip's bins. This optimisation only affects how bins are accessed, leaving how the data is structured in memory, as described in Section 2.5.1, unchanged.

In the example shown in Figure 5.1, each row of the Moore neighbourhood would become a strip, i.e. bins 1-3, 5-7 and 9-11 would be treated as strips.

The SIMT architecture of GPUs does not execute divergent threads in each synchronous thread group (warp) in parallel. This leads to one or more threads within a warp becoming idle whenever branching occurs until the implicit synchronisation point whereby the divergence ends. Therefore it is assumed that when threads within the same warp are accessing different bins, which will most often have differing sizes, all threads within the warp must wait for the thread accessing the largest bin to complete before continuing to the next bin. An example of the impact of this implicit synchronisation at bin changes is shown in Figure 5.2. By merging contiguous bins the Strips optimisation reduces the number of these implicit synchronisation points.

Additionally, the use of Strips can only reduce divergence between threads of a warp accessing differently sized bins. As demonstrated in Figure 5.2, this should both reduce the quantity and duration of idle threads, it is not possible for the divergence to increase as a consequence of this optimisation.

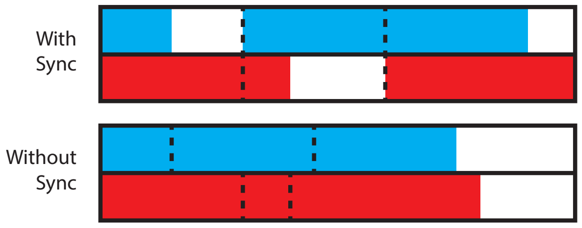


Figure 5.2: A visual example of how implicit synchronisation at bin changes may impact runtime. Checked lines denote the bin changes, synchronisation of bin changes leads to a lot of unusable idle thread time.

The proposed Strips optimisation essentially removes two bin changes from every central strip (of 3 bins), whereby the middle bin is not at the edge of the environment, and one from every boundary strip. This occurs three times per neighbourhood in two dimensions and nine times in three dimensions. Therefore the optimisation provides a near constant time speed up through removal of code, related to the problem dimensionality and number of simultaneous threads. Additionally, the removal of the bin switches, as discussed above, is likely to reduce divergence, whereby threads within the same warp are operating on differently sized bins and hence would change bins at different times. Algorithm 7 shows pseudo-code for the original FRNN search's query technique after the Strips optimisation has been applied, in contrast with Algorithm 5, by removing one of the loops and introducing an additional bounds check.

The Strips optimisation is not compatible with techniques which utilise non-linear indexing of bins, such as space filling curves [56]. However as discussed in Section 2.5 techniques, such as space filling curves, which improve memory locality and coherence are less applicable for modern GPU architectures due to improved caching.

Algorithm 7 Pseudo-code showing the Strips optimisation applied to the FRNN search's query algorithm for a two dimensional environment.

```
vector2i absoluteBin = getBinPosition(origin)
loop -1<=relativeStrip<=1:
  startBin = absoluteBin + vector2i(-1, relativeStrip)
  endBin = absoluteBin + vector2i(1, relativeStrip)
  startBin = clampToValid(startBin)
  endBin = clampToValid(endBin)
  startHash = binToHash(startBin)
  endHash = binToHash(endBin)
  loop i in range(PBM[startHash],PBM[endHash + 1]):
    handle(neighbourData[i])
```

Application of the Strips Optimisation Within FLAMEGPU

FLAMEGPU still iterates bins and messages during a FRNN search's query in the same order as the general case. With the algorithm for changing bin being significantly different (for loop vs if statements), the relative constant cost of bin changes may be slightly different. Hence, experimental results under the Strips optimisation are likely to differ between FLAMEGPU and the general case of the minimal implementation.

5.3 Experiments

There are two key areas where the Strips optimisation can be expected to have an impact, through removal of code and a reduction in divergence of threads sharing warps.

Firstly, the removal of operations should provide a near constant speedup. In 2D, 9 bin changes is reduced to 3, and, in 3D, 27 are reduced to 9. The respective removal of 6 and 18 occurrences of the bin change algorithm (such as that shown in the previous section) should find similarly scaled reductions in runtime between 2D and 3D.

Secondly, there is the impact towards divergence, although this is more difficult to assess directly. By reducing the bin changes, it is possible that there may be an observably greater improvement for models with more computationally intensive handling of items from the FRNN search. However, the handling of divergence is decided by the compiler and GPU firmware. Whilst proportions of active threads can be observed directly using profiling tools, the reason for the particular levels of divergence, especially in such complicated algorithms, is influenced by many runtime properties such as the quantity and distribution of actors.

The Circles model [38] (explained in Section 3.1), with uniform random initialisation, is used to evaluate the impact of the Strips optimisations on the performance of FRNN search. The Circles model allows parameters that can affect performance scaling, such as population size and population density, to be controlled and hence provide understanding of how the impact of the Strips optimisation is affected by runtime properties. The experiments have been evaluated in both two and three dimensions, as the three dimensional case has a greater reduction in bin changes.

FRNN search is composed of two stages, construction and query. As the Strips optimisation does not impact construction, these experiments only investigate impacts to the query stage. In this case, the query stage refers to the neighbourhood search performed by each actors in parallel, including any model specific logic that occurs per item accessed during the FRNN query.

The following experiments were performed;

- Experiment 1: Strips, Scaling Population Size - This experiment aims to evaluate the Strips optimisation with respect to speedup of the query operation through a range of population sizes.
- Experiment 2: Strips, Scaling Density - This experiment aims to evaluate the Strips optimisation with respect to speedup of the query operation through a range of neighbourhood sizes (densities).
- Experiment 3: Strips, Warp Divergence - This experiment aims to evaluate the Strips optimisation with respect to it's impact to divergence.

Experiments 1 and 2 assess the impact of the Strips optimisation on performance scaling through population size and density, in two and three dimensions to identify the properties in which the optimisation is favourable.

In order to assess the impact to divergence, Experiment 3 observes an instance of the Circles model using the NVIDIA Visual Profiler before and after the Strips optimisation has been applied. This allows the amount of divergence within warps to be measured, termed Warp Execution Efficiency (WEE). The Circles model contains an appropriate level of mathematical operations that any impact on divergence should be visible if present.

A bespoke minimal implementation[1] has been used to carry out these experiments. The use of a minimal implementation isolates the FRNN search from additional overheads found within applied frameworks (such as FLAMEGPU) and provides greater control over timing, allowing greater accuracy in the collected result. This bespoke implementation follows the general case presented in Section 2.5. The targeting of the general case ensures optimisations are more widely applicable, whilst still remaining relevant when applied to real models.

FLAMEGPU has not been used to carry out these experiments, however testing has demonstrated similar results to those produced by the minimal implementation. In the following chapter both the minimal implementation and FLAMEGPU are used by the experiments.

[1]`https://github.com/Robadob/sp-2018-08`

5.4 Results

The following experiments were performed using an Nvidia Titan-X (Pascal) GPU in TCC mode under CUDA 9.1 on Windows 10.

The two cases, before and after application of the Strips optimisation, are referred to as ‘Original’ and ‘Strips’ respectively.

5.4.1 Experiment 1: Strips, Scaling Population Size

To assess the impact of the Strips optimisation on the performance of the query operation, the Circles model was executed before and after the Strips optimisation was applied. In these experiments, the density is configured such that an average neighbourhood query should return 70 results. This density was chosen as it is an expected average of densities observed in SPH [167] and MAS. The actor population has been scaled from 100,000 to 5 million, in order to assess the impact through a wide range of viable population sizes and to ensure maximal device utilisation. A total of 50 configurations, uniformly distributed between the minimum and maximum actor population sizes were each executed 200 times. In order to reduce outliers, the average time of the 200 executions has been recorded.

In both the 2D (Figure 5.3a) and 3D (Figure 5.3b) results, execution after application of the Strips optimisation performed the fastest throughout. In 2D the Strips optimised version performs 1.2x faster, whereas in 3D the proportional improvement is a slightly lower 1.17-1.18x throughout. These results show that the impact of applying the Strips optimisation is a performance improvement which scales linearly with actor population.

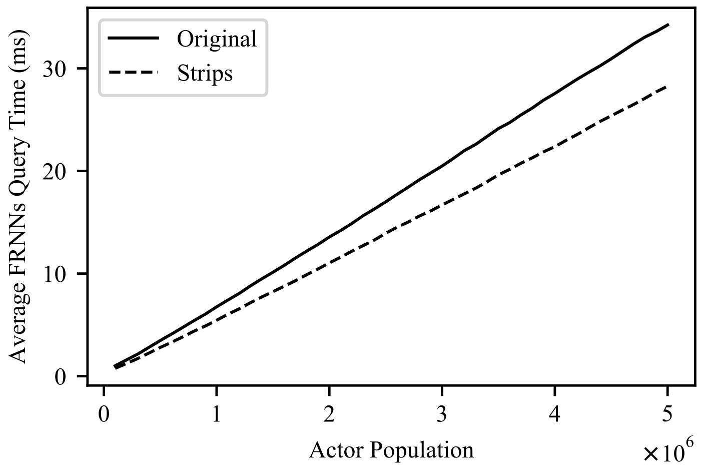
5.4.2 Experiment 2: Strips, Scaling Density

This experiment explores the impact of scaling density, which provides more context to the previous experiment’s results allowing more detailed analysis.

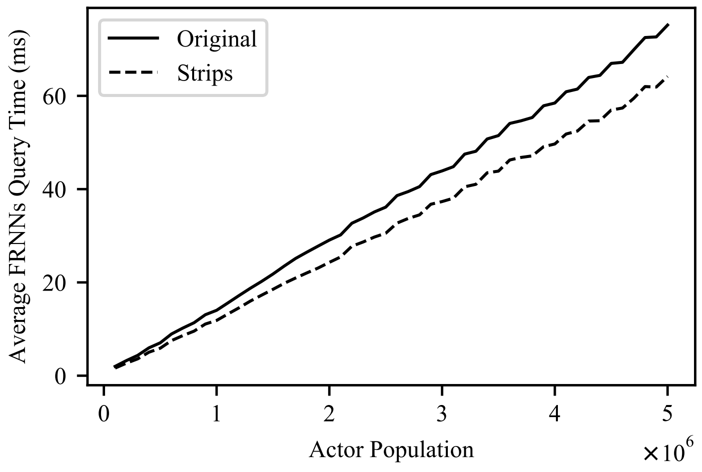
To further assess the impact of the Strips optimisation on the performance of the query operation, the Circles model was executed before and after the Strips optimisation was applied. In this experiment, the impact of changes to density is evaluated with a fixed actor population of 1 million in order to ensure full device utilisation. A total of 50 configurations, uniformly distributed between the minimum and maximum radial neighbourhood volumes, were each executed 200 times. The radial neighbourhood volume refers to the average number of items returned by each FRNN search query. In order to reduce outliers, the average time of the 200 executions has been recorded.

With a fixed actor population of 1 million in 2D (Figure 5.4a), query time increases linearly with radial neighbourhood size. Throughout, the queries under the Strips optimisation perform a stable 1-1.2ms faster than that without the optimisation. A similar pattern is visible in 3D (Figure 5.4b). Here the Strips optimisation performs a stable 2.3-3.5ms faster than that without the optimisation. This roughly three-fold increase is in line with the reduction in bin changes, from 9 to 3 in 2D and 27 to 9 in 3D, reductions of 6

Experiment 1: Strips, Scaling Population Size



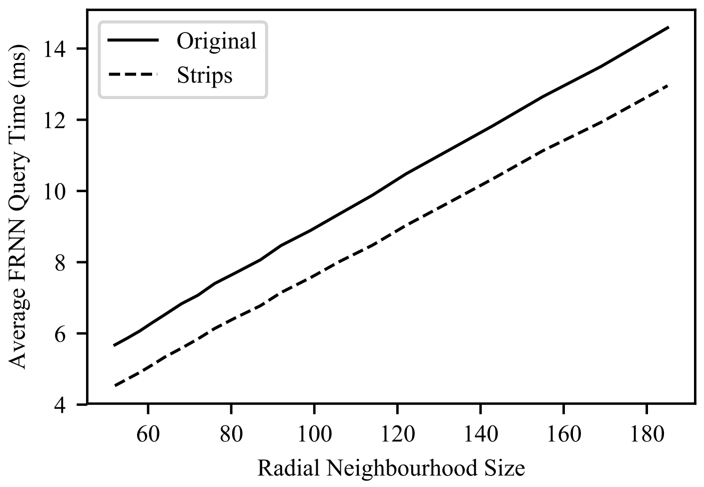
(a) 2D



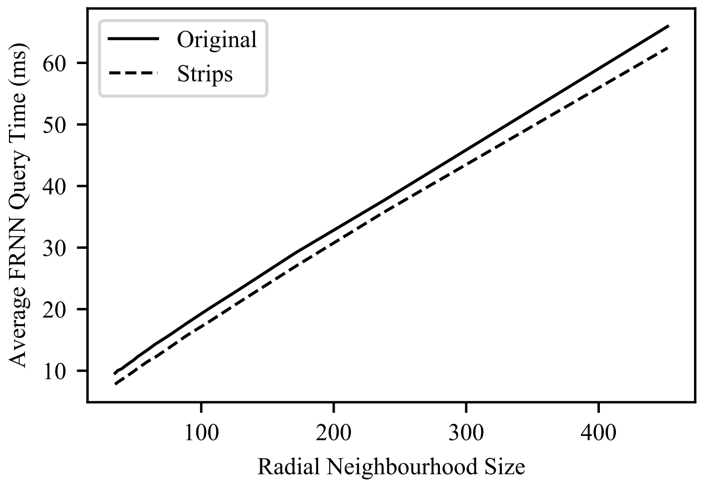
(b) 3D

Figure 5.3: **Strips, Scaling Population:** The performance of the query operation before and after the Strips optimisation was applied during the Circles benchmark model, with an average radial neighbourhood population of 70 actors in (a) 2D and (b) 3D.

Experiment 2: Strips, Scaling Density



(a) 2D



(b) 3D

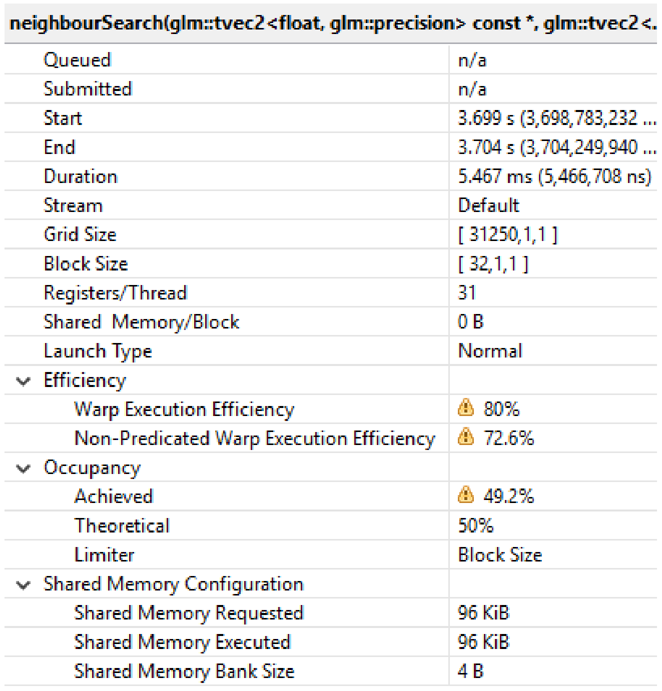
Figure 5.4: **Strips, Scaling Density:** The performance of the query operation before and after the Strips optimisation was applied during the Circles benchmark model, with a population of 1 million actors in (a) 2D and (b) 3D. Radial neighbour size refers to the average number of items found by actors during FRNN search.

and 18 bin changes, respectively. Due to the stable speedup, the proportional speedup decreases significantly as the radial neighbourhood size increases.

In combination with the previous population scaling experiment, it becomes clear that the constant time improvements with respect to population density should also scale proportionally with respect to population size. Low density, low population configurations are shown to have the greater proportional improvement, and high density, high population configurations have a significantly smaller proportional improvement.

5.4.3 Experiment 3: Strips, Warp Divergence

A sample configuration of the Circles model was executed via Nvidia's visual profiler for 200 iterations, enabling observation of performance properties of individual kernel invocations. This allows insight to be gained into how application of the Strips optimisation impacts divergence. Figure 5.5 provides an example some of the information provided about a single kernel invocation. In particular the measurement of divergence is concerned with 'Warp Execution Efficiency' (WEE) which is documented as 'Ratio of the average active threads per warp to the maximum number of threads per warp supported on a multiprocessor.' The optimal value of WEE is 100% which represents no divergence and lower values are harmful to performance as divergence leads to idle threads. The 'Non-Predicated' variant only concerns conditional instructions, therefore it only represents a subset of the total.


neighbourSearch(glm::tvec2<float, glm::precision> const *, glm::tvec2<.

Queued	n/a
Submitted	n/a
Start	3.699 s (3,698,783,232 ...
End	3.704 s (3,704,249,940 ...
Duration	5.467 ms (5,466,708 ns)
Stream	Default
Grid Size	[31250,1,1]
Block Size	[32,1,1]
Registers/Thread	31
Shared Memory/Block	0 B
Launch Type	Normal
Efficiency	
Warp Execution Efficiency	⚠ 80%
Non-Predicated Warp Execution Efficiency	⚠ 72.6%
Occupancy	
Achieved	⚠ 49.2%
Theoretical	50%
Limiter	Block Size
Shared Memory Configuration	
Shared Memory Requested	96 KiB
Shared Memory Executed	96 KiB
Shared Memory Bank Size	4 B

Figure 5.5: Kernel invocation information presented by the Visual Profiler. The WEE metric is listed as Warp Execution Efficiency.

Table 5.1 presents the measured WEE before and after application of the Strips optimisation, for the first and last kernel invocation of the FRNN search's query operation during the Circles model. The first and last invocation have been tested, as they represent uniform random and clustered distributions respectively. A more uniform distribution is likely to have more equally sized bins and thus accounts for an observable higher WEE.

The results in Table 5.1 show that the Strips optimisation has reduced divergence during

Invo. #	**Actor Distribution**	**Original**	**Strips**	**Difference**
0	Uniform Random	49.7%	52.1%	2.4%
199	Clustered	68.9%	80.0%	11.1%

Table 5.1: WEE during execution of the Circles model. Configuration: 2D, 1 million actors, average radial neighbourhood size of 70.

the query operation. This improvement scales the WEE with an absolute improvement of 2.4% (59.7%-52.1%) where actors are most uniformly distributed and divergence is highest, to 11.1% (68.9%-80.0%) where actors are most clustered.

The improvement when actors were in a clustered distribution could also be considered a 1.16x improvement to WEE. When comparing the runtime of the same kernel invocations, 6.58ms and 5.47ms, respectively, a greater improvement of 1.2x is seen. This potentially accounts for both the removal of code and the impact of reduced divergence, which is evidence to suggest the decreased divergence is greater contributor to the performance improvements seen than the removal of code.

The implications of improving WEE are that models which spend more time within the query stage, on processing returned items, would see a greater absolute improvement, as a result of more time being spent within the highly divergent query stage. More time spent within the query stage may be due to higher densities, or MAS with more complex logic for processing returned items (e.g. certain biological cellular models with high numbers of properties). Contrary to this implication, the results from the previous experiment which explored a scaling density only showed a very small absolute increase between the lowest and highest densities. In part, some of the reduction in WEE can be attributed to the reduction of the number of bin change operations, however, this further confirms the complex relationship between thread divergence and performance.

The emergent clusters within the Circles model (earlier shown within Figure 3.1b) have a diameter slightly less than the FRNN search radius, however, they are not centrally located within bins so occupy multiple bins. This causes actors within a cluster to have several different Moore neighbourhoods which they access. Similarly, one bin may contain only a few actors from a cluster, whereas another may contain a high number. Furthermore, due to the 2D mapping of bins, to a 1D index, bin offsets in the Y-axis do not form strips. When extended to 3D, this would likely see even lower WEE, due to greater potential for divergence.

By reducing the individual numbers of bins accessed, the Strips optimisation has allowed bin volumes to merge, such that divergent cases overlap (earlier shown in Figure 5.1). In many cases this will have no impact, due to the initial divergence arithmetic, however, as the profiling data has shown in some cases (shown to be as high as 1/3) divergence has been avoided by the Strips optimisation.

5.5 Conclusions

This chapter has demonstrated the Strips optimisation to reduce operations and divergence of FRNN query, Strips, which removes a redundant operation from the FRNN search's query algorithm. Experiments were carried out using the Circles model, due to its versatility for controlling a range of actor population parameters which influence the performance of FRNN queries.

The direct impact of the Strips optimisation is the removal of redundant code. As demonstrated with the experiments, this provides a near constant speed-up, relative to the actor density and environment's dimensionality.

A secondary impact of the Strips optimisation is the improvement of WEE, such that threads spend less time inactive due to divergence. This is achieved by merging potentially unevenly sized bins, such that the combined total offset, leading to divergence, may be reduced. In the worst case this has no effect on divergence of threads within a warp, whereas in the best case this can cancel all divergence within a warp.

The direct impact of divergence on performance is not easily assessed in isolation, due to thousands of warps executing simultaneously, each with different potentials for divergence. Furthermore, the impact of the optimisation on divergence changes as actors move between bins. However, using profiling tools, it has been demonstrated that WEE can be improved from 68.9% to 80.0%. This could potentially provide an improvement as significant as 1.16x to performance. Observed values show an overall improvement of 1.2x, which additionally accounts for effects such as the removal of instructions between the Original and Strips optimised kernels. The improvement to WEE was greatest with a clustered actor population. These distributions are more likely to be seen in MAS representing structured environments (e.g. macro-biological actors) rather than the highly compacted actor populations found in molecular models and similar.

The experiments in this chapter were carried out using a Pascal architecture GPU and other architectures may differ. In particular, it is not possible to assess the impact with respect to the earlier architectures, such as Fermi and Kepler, which benefit from memory coherence optimisations such as space filling curves [56]. Goswami's application of space filling curves does not isolate the impact, and hence a parallel cannot be drawn.

In summary, the Strips optimisation has been shown to provide a consistent improvement to performance of the FRNN search's query operation. The absolute speedup of this improvement scales linearly with population and remains static with changes to population density. Performance results suggest improvements of 1.18x and 1.2x in 2D and 3D, respectively. Actual improvements are likely to vary as a consequence of the computational expense of the model logic performed, per actor, within the query algorithm and the distribution of actors among the data structure's bins. As such, this chapter has provided contribution C2.

The next chapter applies a further optimisation in conjunction with the Strips optimisation to address the level of redundant data accessed during the FRNN query stage.

This sees the peak improvement to the query operation increase to 1.32x and expands the experiments to consider the performance impact of the optimisations on full models.

Chapter 6

Optimising Queries: Surplus Memory Accesses

Chapter 5 proposed the Strips optimisation for the query stage of FRNN search, building on knowledge of the USP data structure and FRNN search algorithm from Section 2.5, to improve performance through the reduction of divergent threads. Section 2.5 also raised another question relating to FRNN queries; How can the redundant area between the Moore neighbourhood and radial search area during FRNN queries be reduced?

This chapter proposes that the query stage of FRNN search can be further improved by reducing the number of redundant memory accesses. It is posited that by changing the 'Proportional Bin Width', relative to the query radius, the trade-off between redundant items accessed and total bin accesses can be optimised. By optimising this trade-off and reducing the high level of redundant items accessed, reductions in memory traffic may lead to improved performance.

This chapter provides contribution C3, via the proposed 'Proportional Bin Width' optimisation for the query stage of FRNN search, which is compatible with the earlier proposed 'Strips' optimisation, and evaluation of the optimal 'Proportional Bin Width' candidate. Additionally, experiments demonstrating the efficacy of the optimisation are contributed. These experiments utilise the Circles benchmark and a range of full complex simulation models, described in Chapter 3, to both assess the optimisation with regard to a range of properties applicable to the general case, identified in Section 2.5, and demonstrate the impact when applied to full models.

The remainder of this chapter is partitioned into sections which summarise the significance of redundant memory accesses during the FRNN search's query algorithm, present the Proportional Bin Width and combined Strips and Proportional Bin Width optimisations, evaluate these optimisations against benchmark and real-world applied models, discuss the results obtained from the experiments, and conclude the work from this chapter as a whole in the wider context of complex system simulations.

The following chapter then combines the work from this and the previous two chapters, providing a case study which applies all three optimisations simultaneously to two complex system simulations with greatly differing properties in order to confirm the understanding of their suitability.

6.1 Proportional Bin Width Optimisation

Standard implementations of USP subdivide the environment into bins with a width equal to that of the radial search radius. This decision ensures that all radial neighbours will be guaranteed to be found in the same or surrounding bins of the query's origin. This is referred to as the Moore neighbourhood. The implication of this decision is that many surplus items outside of the radial neighbourhood must be accessed. In 2D the Moore neighbourhood accessed has 2.86x the spatial area of that of the radial neighbourhood. In 3D this becomes 6.45x the volume. It is proposed within this chapter than an alternate ratio between the radial search radius and bin width ('Proportional Bin Width'), can reduce the surplus area accessed during FRNN queries, which may result in improved FRNN search performance.

By adjusting the width of the bins relative to the neighbourhood radius, the number of bins that must be accessed (to ensure the entire radial neighbourhood is covered) and the volume of individual bins change. Thus the total (and surplus) area of the environment accessed also changes. Figure 6.1 provides an example of how alternate Proportional Bin Widths change these values. When the width of bins is R, the total grid area is greater than when the width of bins is $\frac{1}{2}R$, equivalent to $3R^2$ and $2.5R^2$ respectively. These areas are spatial, but provide an estimate for the proportion of redundant items accessed in the average case or under uniformly distributed items.

As detailed in the previous chapter, additional bin accesses have a negative impact on performance. Therefore, in order to optimise redundant items accessed during the query operation, the impact of increased bin accesses per query must also be balanced.

Hoetzlein [77] considered a similar approach in his work, however, he did not suggest an optimal proportion, only considering the trade-off between bin volume and bins per query. The following calculations give a wider consideration to the theoretical impact of adjusting the Proportional Bin Widths.

The min (a_g) and max (a_G) grid areas of access under any Proportional Bin Width in 2 dimensions can be calculated using Equations 6.1 & 6.2, where R is the chosen search radius and W the absolute bin width. In Table 6.1, a_g increases when $R = 0.7$, as 0.7 is not a factor of the search diameter, whereas $R = 1$ and $R = 0.5$ are both able to achieve the minimum grid area that would fit the search radius.

$$a_g = \left\lceil \frac{2R}{W} \right\rceil^2 W^2 \tag{6.1}$$

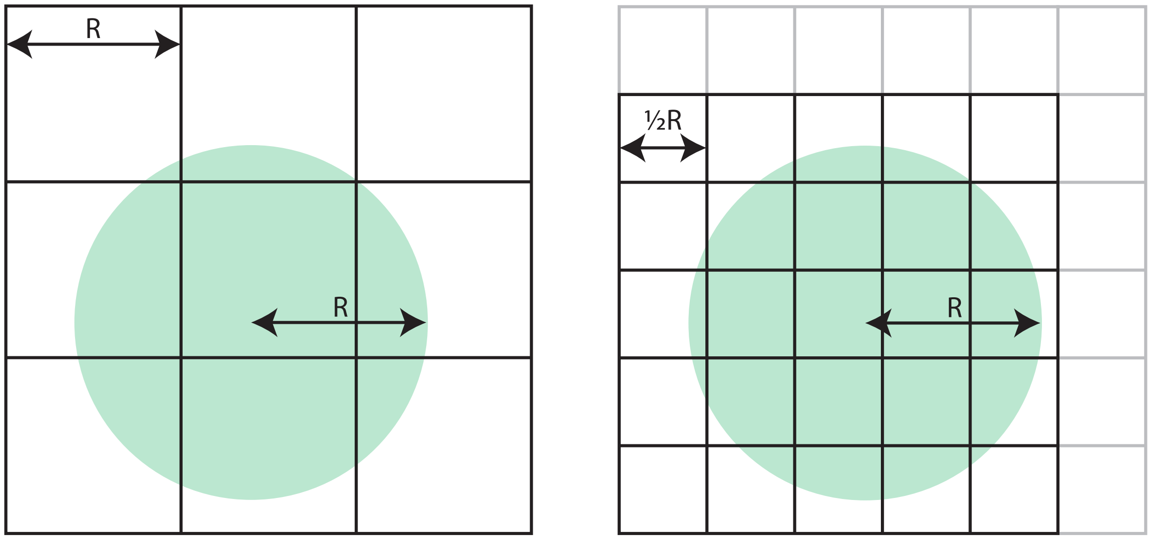


Figure 6.1: Visual representation of how different bin widths (grid cells) require different search areas to cover the whole radial neighbourhood (green circles). Left, $BinWidth = R$ requires 9 bins with a total area of $3R^2$. Right, $BinWidth = \frac{R}{2}$ requires 25 bins with a total area of 2.5^2.

$$a_G = \left(\left\lceil \frac{2R}{W} \right\rceil + 1\right)^2 W^2 \tag{6.2}$$

By additionally calculating the proportion within the min grid area (p) and max grid area (p'), in a 1 dimensional implementation, Equation 6.4 can be used to calculate the average grid area access (a_a) in two dimensions.

$$\begin{aligned} p &= \frac{2R - \left\lfloor \frac{2R}{W} \right\rfloor W}{W} \\ p' &= 1 - p_m \end{aligned} \tag{6.3}$$

$$a_a = p^2 a_g + 2pp'\sqrt{a_g}\sqrt{a_G} + p'^2 a_G \tag{6.4}$$

Dividing by the radial neighbourhood area (a_r), Equation 6.5 then provides the surplus multiplier, which denotes the search grid's area relative to the radial neighbourhood area. This represents the level of redundant memory access where data is uniformly distributed.

$$a_r = \pi R^2 \tag{6.5}$$

Table 6.1 gives the values resulting from the above Equations 6.1-6.5 for three differing Proportional Bin Widths. These equations can be extended to 3 dimensions or more, as required, as they are constructed from the 1-dimensional mathematics surrounding the radial area ($2R$) divided by the bin width (W).

Proportional Bin Width	**1**	**0.7**	**0.5**
Min Grid Area (a_g)	4	4.41	4
Max Grid Area (a_G)	9	7.94	6.25
Average Grid Area (a_a)	9	4.84	6.25
Bin Count Max ($(\lceil\frac{2R}{W}\rceil + 1)^2$)	9	16	25
Strip Count Max ($\lceil\frac{2R}{W}\rceil + 1$)	3	4	5
Max Surplus Mult. ($\frac{a_G}{a_r}$)	2.86	2.50	1.99
Average Surplus Mult. ($\frac{a_a}{a_r}$)	2.86	1.54	1.99

Table 6.1: This table provides an example of the variability by changing the bin width as a proportion of the search radius of 1 (with a corresponding radial search area of 3.14) in 2D. These values can be calculated using Equations 6.1 to 6.4.

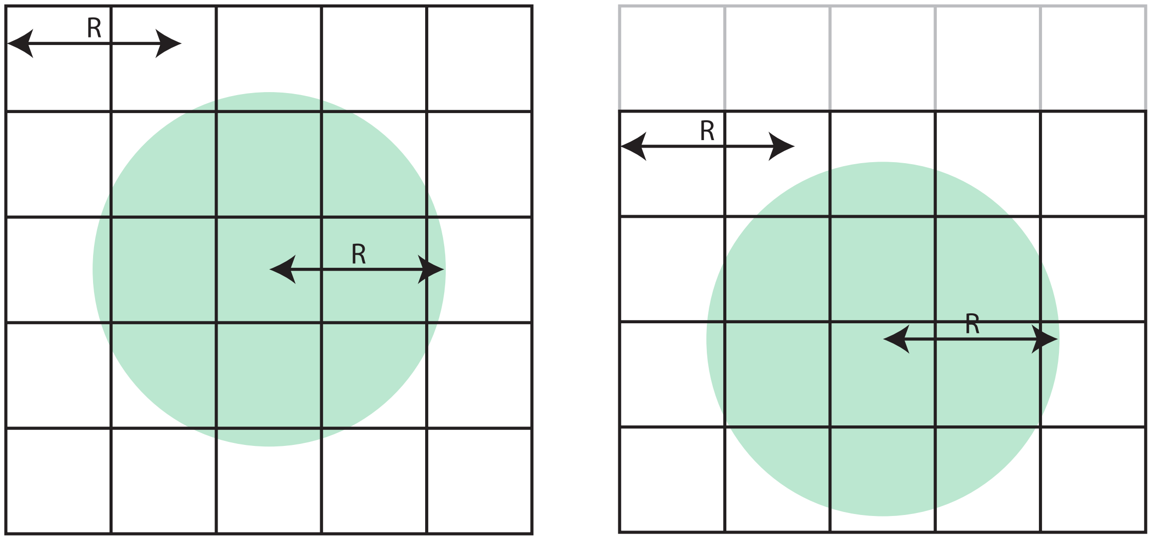


Figure 6.2: Visual representation of how under some bin widths the query origin affects the size of the required search area to cover the whole radial neighbourhood. R denotes the neighbourhood radius.

Some bin widths lead to different bin counts dependent on the query origin's position within its own bin – see Figure 6.2, where the query origin can lead to either a square or rectangular search grid. Due to the SIMT architecture of the GPU, all 32 threads of each warp will step through the maximum number of bins any individual thread within the warp requires, although this may lead to some threads remaining idle through later bins. Similarly, it is expected that threads processing surplus items (which do not require handling by the model), will incur a runtime cost similar to that of the desired items which are likely to incur additional compute.

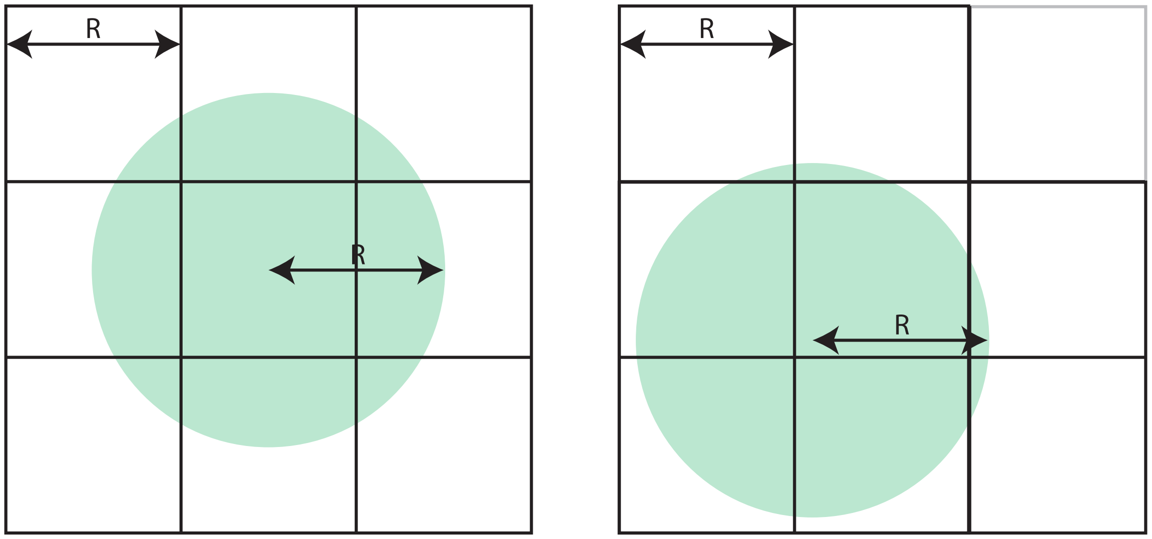


Figure 6.3: Visual representation of how, under some bin widths and search origins, corner bins do not always need to be accessed. R denotes the search radius.

Some combinations of Proportional Bin Width and query origin do permit for corners of Moore neighbourhoods to be skipped, as they do not intersect with the radial neighbourhood (see Figure 6.3). However, the expensive mathematical operations (e.g. trigonometrical functions) to test for bins which may be skipped and frequency of checks required, combined with the low incidence of bins which may be skipped, suggests that in most cases the additional compute necessary to detect and skip the few redundant corner bins would incur a prohibitive runtime cost. This cost would worsen as the Proportional Bin Width is decreased and bins become smaller, outweighing any performance savings in all but the highest densities.

A further consideration with respect to increasing the number of bins is that this leads to a larger partition boundary matrix. This both requires more memory for storage and increases construction time. These impacts are considered further in Sections 6.3 and 6.4.

Algorithm 5 (Repeated from page 85) Pseudo-code showing the FRNN search's query algorithm for a two dimensional environment, before optimisation.

```
vector2i absoluteBin = getBinPosition(origin)
loop -1<=relativeBin.x<=1:
  loop -1<=relativeBin.y<=1:
    selectedBin = absoluteBin + relativeBin
    if selectedBin is valid:
      hash = binToHash(selectedBin)
      loop i in range(PBM[hash],PBM[hash + 1]):
        handle(neighbourData[i])
```

Algorithm 8 Pseudo-code showing the Proportional Bin Width optimisation applied to the query operation for a two dimensional environment.

```
vector2i binMin = getBinPosition(origin - radius)
vector2i binMax = getBinPosition(origin + radius)
loop binMin.x<=selectedBin.x<=binMax.x:
  loop binMin.y<=selectedBin.y<=binMax.y:
    hash = binToHash(selectedBin)
    loop i in range(PBM[hash],PBM[hash + 1]):
      handle(neighbourData[i])
```

Algorithm 8 demonstrates how the Proportional Bin Width optimisation may be applied by altering the selection of bins iterated, in contrast to that of Algorithm 5 (Original FRNN query). It can be seen that in the 2D example the nested loop changes from iterating a 3×3 Moore neighbourhood to an $N \times M$ Moore neighbourhood. The exact values of N and M are dependent on the values returned by `getBinPosition()`, which clamps the continuous space coordinates to within the environment bounds and then maps them to their containing bin's discrete coordinates.

6.1.1 Combined Technique

The removal of W^2 from Equations 6.1 & 6.2, to calculate min (a_g) and max (a_G) area, can be used to calculate the number of bins. Similarly, taking the square root of these values provides the number of Strips, following the optimisation from Section 5.2.

Algorithm 6 (Repeated from page 88) Pseudo-code showing the Strips optimisation applied to the FRNN search's query algorithm for a two dimensional environment.

```
vector2i absoluteBin = getBinPosition(origin)
loop -1<=relativeStrip<=1:
  startBin = absoluteBin + vector2i(-1, relativeStrip)
  endBin = absoluteBin + vector2i(1, relativeStrip)
  startBin = clampToValid(startBin)
  endBin = clampToValid(endBin)
  startHash = binToHash(startBin)
  endHash = binToHash(endBin)
  loop i in range(PBM[startHash],PBM[endHash + 1]):
    handle(neighbourData[i])
```

Algorithm 9 demonstrates how Algorithms 7 (Strips) and 8 (Proportional Bin Width) can be combined to utilise the benefits of each optimisation simultaneously during the query operation. The nested loop over `selectedBin` from Algorithm 8 (Proportional Bin Width) has been replaced by the strip loop and validation from Algorithm 7 (Strips).

Algorithm 9 Pseudo-code showing the combined optimisation applied to the query operation for a two dimensional environment.

```
vector2i binMin = getBinPosition(origin - radius)
vector2i binMax = getBinPosition(origin + radius)
loop binMin.y<=strip<=binMax.y:
  startBin = vector2i(binMin.x, strip)
  endBin = vector2i(binMax.x, strip)
  startHash = binToHash(startBin)
  endHash = binToHash(endBin)
  loop i in range(PBM[startHash],PBM[endHash + 1]):
    handle(neighbourData[i])
```

6.2 Implementation

To evaluate the impact of the proposed Proportional Bin Width optimisation, it was first applied to the Circles benchmark using the minimal bespoke implementation. This was used in the previous chapter, to assess the Strips techniques. This provides more controlled experiments which removes any potential to be skewed by FLAMEGPU's abstraction layer. As the Strips optimisation is independent of the implementation of Proportional Bin Width, and is likely to further benefit the optimisation, the tested optimisation combines the Strips and Proportional Bin Width optimisations.

After isolated experimentation the impact of the optimisation is evaluated within the context of FLAMEGPU in order to demonstrate how the optimisations apply to existing multi-agent simulations. The remainder of this section details the implementation of the minimal bespoke version (available online[1]), due to the additional considerations required to implement the Proportional Bin Width technique.

6.2.1 Construction Stage

The Proportional Bin Width and combined optimisation both affect construction. The impact is limited to affecting the scale of the environment's subdivision, subsequently increasing (or decreasing) the memory footprint of the Partition Boundary Matrix (PBM).

Figure 2.10 (on page 42) highlights how the PBM consists of a single unsigned integer array, with a length one greater than the number of environment bins ($N_{bins} + 1$). Each entry into the array, except for the final entry, identifies the first index of the corresponding bin's items in the neighbour data array. The final element of the array denotes the total number of items. The layout of the PBM allows the bounds of a bin's data in the neighbour data array to be identified by accessing both the index of the desired bin and the subsequent index within the PBM.

The construction is implemented according to the earlier algorithm detailed in Section 4.3 on page 68. The only changes necessary for construction under the Combined

[1] `https://github.com/Robadob/sp-2018-08`

technique are applied to the utility methods, used for mapping continuous environment coordinates to discrete bin coordinates and indices (e.g. Equations 4.2 (2D) or 4.3 (3D)).

6.2.2 FRNN Query Stage

To access the items within a radial search area during a query stage, the bins which cover that area must be identified and accessed. Whilst the neighbour data from a single bin can be identified with two accesses to the PBM, to perform a query a contiguous block of bins must be accessed. This contiguous block of bins must include all bins which may intersect the search area. The (maximum) number of bins accessed can be calculated using Equation 6.6. When $PBW = 1.0$, a $3 \times 3(\times 3)$ block of bins must be accessed, and when $PBW = 0.5$, a $5 \times 5(\times 5)$ block of bins must be accessed.

$$bins = (1 + 2\left\lceil \frac{1}{PBW} \right\rceil)^{Dims} \tag{6.6}$$

Algorithm

The FRNN search's query algorithm can be described using the pseudo-code for the combined Strips and Proportional Bin Width in Algorithm 9 on page 103, which combines the individual optimisations from Algorithms 7 and 8 (Strips and Proportional Bin Width respectively). Each GPU thread operates independently, concerned with its own unique FRNN query. In this two dimensional example we loop over `relativeStrip` which represents the Y axis, the rows of the block of bins. In three dimensions this would be a nested loop over the Y and Z axes. The bounds of the strip can then be calculated by taking the x coordinate from `binMin` and `binMax`, respectively, and using `strip` as the y coordinate. This provides the start and end coordinates of the current strip of bins to be accessed. These values must be clamped within a valid range and transformed into 1-dimensional bin indices (using Equations 4.2 or 4.3). This can be applied during the initial calls to `getBinPosition()` rather than per strip. After this, the calculated bin indices (`startHash` and `endHash`) can be used to access the PBM to identify the range of elements within the storage array (`neighbourData`) that must be accessed.

This algorithm accesses all items within potentially intersecting bins. The locations of items must additionally be compared to only handle those which lie within the radial area of the search's origin.

To ensure maximal performance of FRNN search queries, it is necessary to limit the kernel's block size to 64 threads. This maximises hardware utilisation, whilst reducing the number of threads remaining idle, where item distributions are not uniform (hence leading to an unbalanced workload). The optimisation is visible in the NVIDIA-distributed CUDA particles example and has implications similar to those regarding divergence from the previous chapter, however, details of this technique have not been found in general literature pertaining to FRNN search.

FLAMEGPU's algorithm for handling bin iteration is implemented in a different way. As explained in the previous chapter, the query state is tracked and updated each time an item is retrieved. This implementation abstracts the behaviour from users, allowing them to access FRNN queries within their model logic using a single while loop. Despite these differences, the actual order of item access remains consistent.

6.3 Experiments

The experiments have been designed to demonstrate four important characteristics of this research:

- Experiment 1: Proportional Bin Width - This experiment aims to evaluate the Proportional Bin Width optimisation (Section 6.1) in combination with the Strips optimisation with respect to the trade-off between the query operation's performance and proportional bin width.
- Experiment 2: Construction - This experiment aims to evaluate the impact of the presented optimisations on the cost of USP construction relative to performing queries.
- Experiment 3: Population Scaling - This experiment aims to evaluate the impact of the presented optimisations on the linear scaling of query operations with respect to increasing population size.
- Experiment 4: Model Testing - This experiment aims to evaluate the optimisations presented with respect to existing multi-agent simulations within FLAMEGPU.

Within these experiments, construction refers to the construction of the USP data structure, which involves sorting items and generating an index to the start of each bin's storage (PBM). The query stage refers to the neighbourhood search performed by each actor in parallel, which includes any model specific logic that occurs per neighbour accessed.

The minimal bespoke implementation[2] follows the generalised case observed in many frameworks and examples, e.g. FLAMEGPU [8] and the NVIDIA CUDA particles sample [58][3]. The targeting of the general case ensures optimisations are more widely applicable, whilst still remaining relevant when applied to real models. Additionally, the use of a bespoke implementation isolates the FRNN search from additional overheads found within applied frameworks and provides greater control over timing.

The minimal bespoke implementation uses the Circles model as described in section 3.1 on page 53 to evaluate the impact of the optimisations presented in Sections 5.2 and 6.1 on the performance of FRNN search. The Circles model allows parameters that can affect

[2]`https://github.com/Robadob/sp-2018-08`

[3]The CUDA particles sample code is available on installation of the CUDA toolkit.

performance scaling, such as population size and population density, to be controlled. The presented optimisations have been evaluated in both two and three dimensions.

Additionally, FLAMEGPU has been modified[4] so that the optimisations presented within this chapter can be applied to the Pedestrian, Boids and Keratinocyte models in experiment 4. These models were introduced earlier in Section 3.3.

6.4 Results

Benchmarks of a USP implementation, before and after the application of the optimisations presented in Sections 5.2 and 6.1, were collected using the Circles model presented in Section 3.1. Data in this section was collected using an NVIDIA Titan-X (Pascal) in TCC mode with CUDA 9.1 on Windows 10.

In the following results, the two cases, before and after application of the Strips optimisation, are referred to as 'Original' and 'Strips', respectively. Trailing numbers such as 0.50 and 1.00 refer to the Proportional Bin Width, such that 'Original 1.00' refers to the Original technique before changes.

Similarly 'Radial Neighbourhood Size' has been used as a proxy for density, as it refers to the average number of items returned by an FRNN query.

The following sections correspond to the experiments outlined in Section 6.3.

6.4.1 Experiment 1: Proportional Bin Width

To assess the impact of the combined Strips and Proportional Bin Width optimisation on the performance of the query operation, the Circles model was executed with a range of Proportional Bin Widths (1.0, 0.7, $\frac{2}{3}$, 0.5, 0.4). The proportional bin width 1.0 is equivalent to the plain Strips optimisation from the previous chapter. The others were selected based on an extended version of Table 6.1, as their properties had the lowest combined maximum strip count and surplus multipliers.

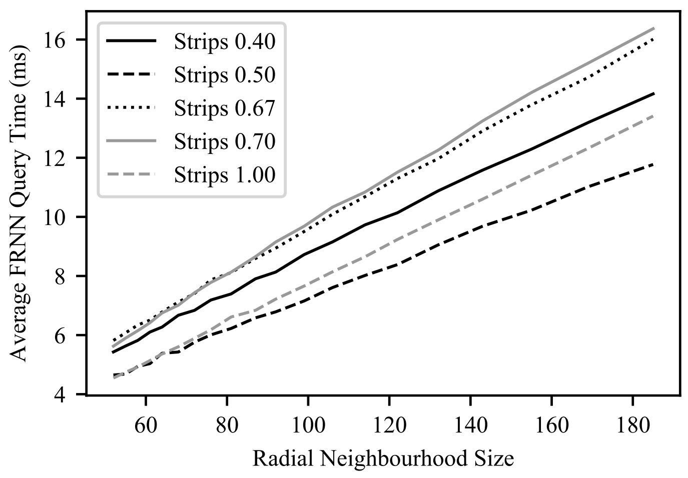
In 2D (Figure 6.4a), the most successful Proportional Bin Width found is 0.5. By reducing the surplus to 1.99x (equation and details available in Section 6.1), and only increasing the number of strips by one, it is able to improve on the performance of the original Strips implementation's query operation. This improvement provides a roughly stable 15% speed up over the original Strips implementation throughout all densities tested.

The remaining three Proportional Bin Widths tested, 0.7, $\frac{2}{3}$ and 0.4, have worse performance than the original Strips implementation (Proportional Bin Width of 1.0).

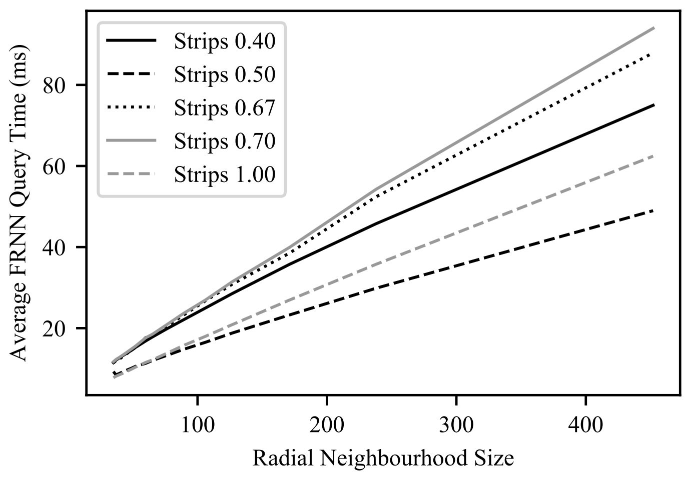
As expected, whilst a Proportional Bin Width of 0.7 should have an average surplus of 1.54x (relative to the Original's 2.87x), its performance can be attributed to its maximum surplus of 2.50x coupled with the additional strip, due to the divergence caused by alternate query origins having varying numbers of bins. The Proportional Bin Width of $\frac{2}{3}$ has an

[4]`https://github.com/Robadob/FLAMEGPU` (Changes are present in the branch titled 'PBW')

Experiment 1: Proportional Bin Width



(a) 2D



(b) 3D

Figure 6.4: **PBW, Strips, Scaling Density:** The performance of the query operation during execution of the Circles model with 1 million actors, with the Strips optimisation applied in combination with a selected range of Proportional Bin Widths, in (a) 2D and (b) 3D.

average surplus of 2.26x with no additional bin changes. Its lack of performance suggests that floating point precision limitations may have removed the primary benefit of $\frac{2}{3}$ being a factor of 2 (thus adding redundant bin changes). The Proportional Bin Width of 0.4 also harmed performance, however, the runtime was much closer to that of the original Strips implementation. 0.4 is calculated to have average and maximum surpluses of 1.54x and 1.83x, respectively. However this comes at the cost of 5 strips rather than 3, so that the cost of additional bin changes and impact to divergence harms the runtime cost.

In 3D (Figure 6.4b), as seen in 2D, the Proportional Bin Width 0.5 produces the best result, achieving a 27% speed up over the Original technique at the highest density.

Figure 6.5 takes the most promising Proportional bin width from Figure 6.4 and applies it to the Original and Strips techniques. When compared with the Original optimised technique, the combined Strips optimised technique with a Proportional Bin Width of 0.5 allows the query operation to execute 27% faster at the highest density in 2D. This increases to 34% in 3D. Similarly, the graph demonstrates how the combination of Strips and a Proportional Bin Width of 0.5 exceeds the improvement of both optimisations in isolation.

In conclusion, the Proportional Bin Width of 0.5 was among the most promising based on calculations from the earlier Section 6.1, reducing the surplus message access by 30% with 2 additional bin changes under the Strips optimisation in 2D. This produced a total improvement of 1.27x and 1.34x over the Original unoptimised technique in 2D and 3D, respectively.

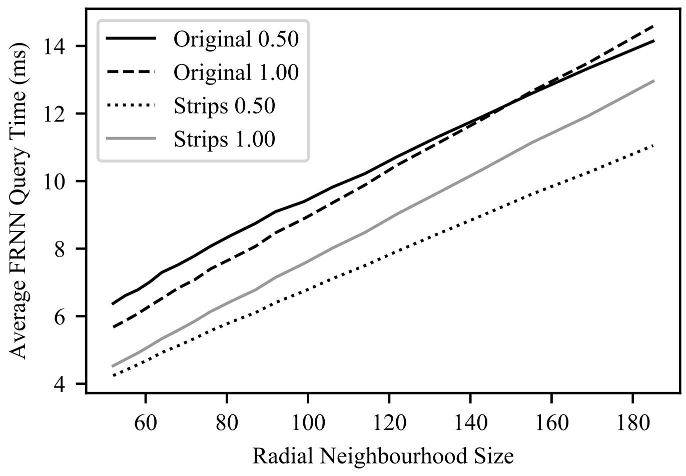
The outlier to these calculations was the Proportional Bin Width of $\frac{2}{3}$. This can be attributed to its tight fit of the Proportional Bin Width to the minimum Moore neighbourhood ($3\frac{2/3}{R} == 2R$) and floating point imprecision leading to an extended Moore neighbourhood. This greatly increases its redundant area and adds an additional bin change. Furthermore, due to the understanding of divergence gained from the previous chapter, it is clear this only need occur to a small proportion of threads to affect the performance of many others due to the impact of divergence.

6.4.2 Experiment 2: Construction

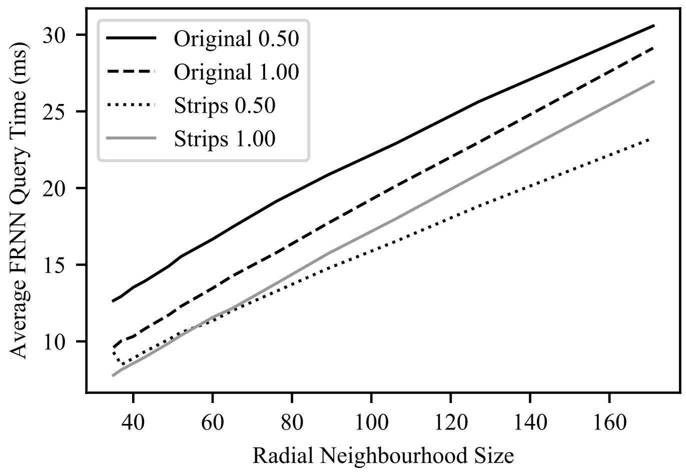
To assess the impact of the Proportional Bin Width to construction, this experiment instead explores the ratio of construction time to query time for the previous experiment (Figure 6.5).

Figure 6.6 demonstrates the small impact changes to construction time have versus query operation runtime. Construction accounts for 3% or less of the combined construction and query runtimes, reducing further as density and dimensionality increase. Regardless, comparing times between Figures 6.5 & 6.6, shows that proportion of construction time is reduced slightly by the Proportional Bin Width of 0.5, despite the earlier speedup to the query operations. This can be attributed to the increased number of bins (4x in 2D, 8x in 3D) which reduces contention. This was shown earlier in Figure 4.2 to benefit the

Experiment 1: Proportional Bin Width



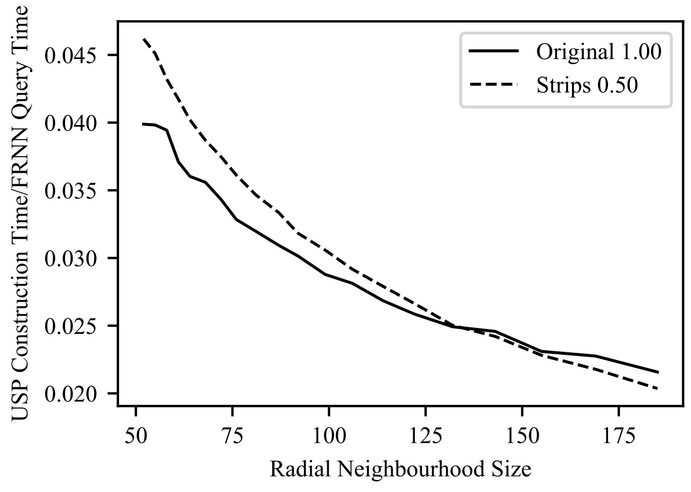
(a) 2D



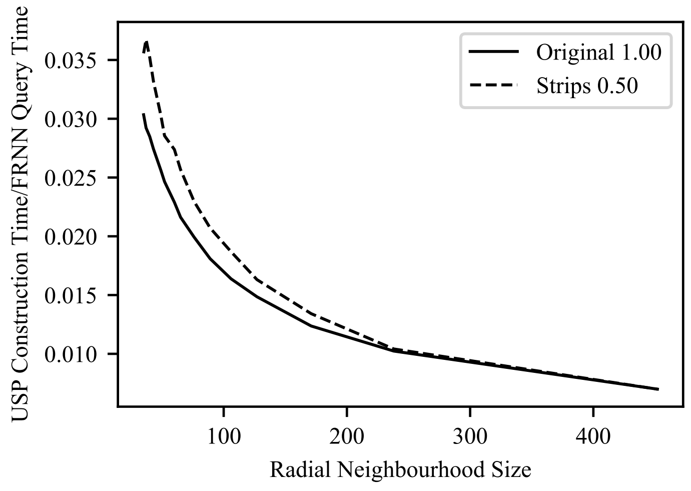
(b) 3D

Figure 6.5: **Original, PBW, Strips, Scaling Density:** A comparison of the performance of the query operation for the Original and Strips Optimised techniques with and without best Proportional Bin Width from the previous experiment applied, during execution of the Circles model with 1 million actors across a selected range of Proportional Bin Widths in (a) 2D and (b) 3D.

Experiment 2: Construction



(a) 2D



(b) 3D

Figure 6.6: **Best PBW, Construction/Query Time:** A comparison of the construction time and query stages' execution time for the Original unoptimised technique and the combined Strips and 0.5 Proportional Bin Width optimised implementations during execution of the Circles model with 1 million actors across a selected range of Proportional Bin Widths in (a) 2D and (b) 3D.

underlying atomic counting sort algorithm.

In conclusion, after application of the Strips and Proportional Bin Width 0.5, construction remains an insignificant contributor to the total runtime impact of FRNN search.

6.4.3 Experiment 3: Population Scaling

To assess the impact of the Proportional Bin Width with regards to population scaling, this experiment investigates how performance for the Original unoptimised technique and the combined Strips optimised technique with a Proportional Bin Width of 0.50 (‘Strips 0.50’) scale with population size.

Figure 6.7 demonstrates how both techniques scale linearly as population size increases ($O(n)$). As discussed in section 6.1, the optimisations applied affect the number of bins accessed and the scale of each bin relative to the search radius. These two variables are constant with respect to a scaling population.

The jagged performance seen in Figure 6.7b is a consequence of the environment dimensions, locked to a multiple of bin width, (and consequently population density) not scaling as smoothly as the population size.

3,000,000 actors is adequate to fully utilise the computational capacity of the GPU used, such that the linear performance trend can be expected to continue until the GPU's memory capacity becomes a limiting factor [140]. In 2D the optimised version performs 1.23-1.33x faster. In 3D it is 1.18-1.22x faster.

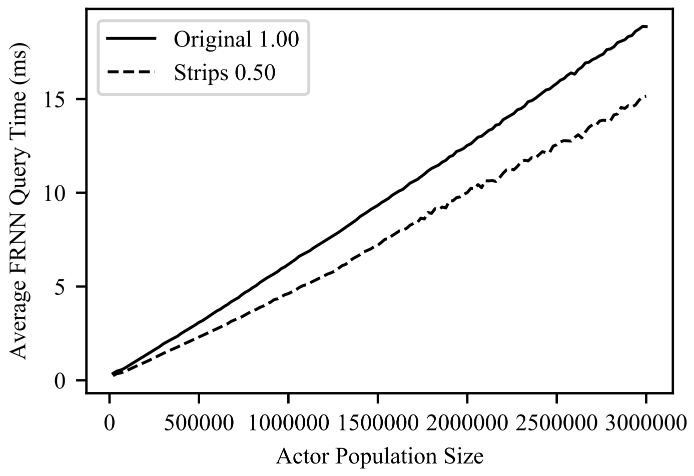
6.4.4 Experiment 4: Model Testing

Three example models found within FLAMEGPU, introduced in section 3.3, have been utilised to demonstrate the benefit of the optimisation in practice. This application to differing models aims to highlight how differences between their application of FRNN search (e.g. actor density, 2D or 3D environment) affect the optimisation.

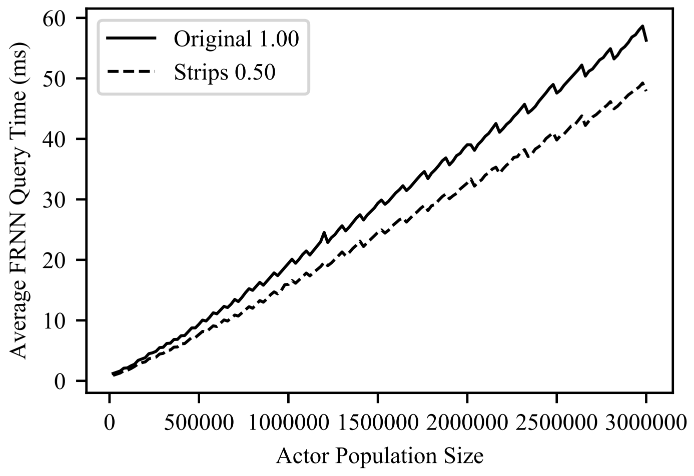
The results in Table 6.2 show that the combined optimisation with a Proportional Bin Width of 0.5 has allowed the query operation within the Boids model to operate 1.11x faster than the original's runtime. When combined with construction, which sees a slight increase, the overall runtime executes 1.15x faster than prior to optimisation. The Keratinocyte model sees a greater speedup of 1.32x for the query operations's runtime, reduced to 1.21x of the overall runtime. As the Keratinocyte model utilises two different densities of FRNN search, this greater speedup for the Keratinocyte model can be attributed to the high density one exceeding the densities found in other models, as was shown in earlier experiments to be more favourable for the Proportional Bin Width optimisation. The Keratinocyte model does, however, have several additional layers unaffected by FRNN search, reducing the proportional impact to the overall runtime.

Similarly, the only 2D model tested, Pedestrians, executed its query 1.03x faster than that of the original. This resulted in an overall speedup of 1.03x (two decimal places)

Experiment 3: Population Scaling



(a) 2D



(b) 3D

Figure 6.7: **Best PBW, Scaling Population:** Comparisons of the performance of the original and the combined Strips and 0.5 Proportional Bin Width optimised implementations during execution of the Circles model across a range of actor population sizes in (a) 2D with radial neighbourhood volumes of ~60 and (b) 3D with radial neighbourhood volumes of ~100.

		Query Time (s)			Construction Time (s)			Total Execution Time (s)		
Model	**Actors**	**Original**	**Optimised**	**Speedup**	**Orig.**	**Opt.**	**Speedup**	**Orig.**	**Opt.**	**Speedup**
Boids	262144	746.52	646.56	1.15x	5.10	7.02	0.73x	752.04	654.00	1.15x
Pedestrian	16384	1.01	0.98	1.03x	0.65	0.66	0.98x	2.25	2.18	1.03x
Keratinocyte	25000-41789	5.80	4.38	1.32x	0.94	1.87	0.50x	8.14	6.70	1.21x

Table 6.2: The runtime of 3 models within FLAMEGPU before and after the combined optimisation has been applied with a Proportional Bin Width of 0.5, when executed for 10,000 iterations.

when compared with the original. The Pedestrians model has the lowest population size of the three tested and in combination with a low density of actors there will be relatively lower levels of memory traffic. This limits the necessity and impact of the optimisation as memory access bottlenecks are not as restricted.

6.5 Conclusions

This chapter has proposed an optimisation called Proportional Bin Width, which reduces the number of redundant items accessed during the FRNN search's query algorithm. This optimisation has been demonstrated in combination with the Strips optimisation to highlight the most promising bin width to search radius ratio and the subsequent improved performance.

The theoretical analysis and experimental testing carried out has found the best ratio between search radius and environment subdivision bin width to be 0.5, which reduces redundant results from the FRNN query by 47% over the Original technique. Combined with the Strips optimisation, the number of bins accessed in 2D increases from 3 to 5, however, the optimisation still consistently outperforms the individual optimisations and the original technique in all but the lowest densities tested.

The direct impact of this Proportional Bin Width is that quantity of redundant area accessed reduces by 47% in 2D (from 1.86x to 0.99x) and 50% in 3D (from 5.44x to 2.73x). In combination with the Strips optimisation this has provided a peak speedup to the query operation under the Circles model of 1.27x in 2D and 1.34x in 3D.

A further impact of this is that the number of bins required to represent an environment of equal size increased by 8x, affecting both the memory required and the time to construct the data structure's PBM. However, experiments have confirmed that despite these changes the run-time impact of construction remains negligible in the majority of cases.

When assessing the impact of the combined optimisation against applied models, a more complex pattern of results was found. MAS utilise FRNN search in a wide variety of ways, with differing fidelity, actor distributions and dimensionality on top of each model's unique logic. These variables all impact the performance of FRNN search, often as a secondary consequence of the underlying GPU architecture. The results of the applied model testing demonstrated this with a peak improvement to the FRNN query of 1.32x faster. However, due to additional model logic, independent of the optimisation, the impact to overall runtime in this best case is reduced to a 1.21x speed up.

Due to the trade-off between surplus item reads and total bins accessed, it is imperative that the Strips optimisation is used alongside the Proportional Bin Width optimisation. Application of the Strips optimisation reduces the total bin access to $\sqrt{n}$ in 2D, and $\sqrt[3]{n}^2$ in 3D. This has a significant impact under the Proportional Bin Width optimisation. For example, in the case of a 0.5 Proportional Bin Width, the 2D Moore neighbourhood search grid expands from $3x3$ to $5x5$.

The theoretical analysis carried out highlighted the trade-off of two performance impacting properties for different values of Proportional Bin Width, redundant area accessed during query and total bins accessed, and allowed the scope of Proportional Bin Widths to be reduced. However, the complex relationship between this trade-off and performance, due to the secondary impacts to divergence and cache utilisation, meant that experimental testing was required to further reduce the list of feasible options to identify the most viable.

In summary, the combined Strips and Proportional Bin Width optimisations have been shown to provide a performance improvement which increases with density and the computational cost of handling each item surveyed. As such the performance improvement seen is likely to vary significantly between models. However, across the models tested, improvements as high as 1.22x were observed to the overall runtime of a model's performance. As such, this chapter has provided contribution C3.

Chapter 7

Case Studies & Discussion

The previous chapters of this thesis have introduced FRNN search as used by complex system simulations and proposed three optimisations for improving the performance of the construction and query stages. This chapter combines these optimisations to demonstrate their impact within the context of a number of large scale complex system simulation case studies. The case studies are examples of transport and crowd simulation which are required to have high performance.

This chapter provides contribution C4, via the specification and demonstration of properties which classify the potential for a complex system simulation to benefit from the optimisations proposed within this thesis. These specifications are applied to two case studies as a demonstration, allowing an understanding of the wider implications, including how best to utilise newly available execution time, and the justification for applying these optimisations to other viable applications.

7.1 Overview

The previous chapters of this thesis have identified 3 novel optimisations which can be applied to FRNN search:

- Atomic counting sort for construction of the USP data structure: this uses a novel sorting and construction algorithm to generate part of the data structure as a by-product of the sort.
- Strips, for the query operation: this removes redundant operations, providing benefits via both code removal and improved WEE.
- Proportional Bin Width, for the query operation: this adjusts the resolution of the environment's subdivision, which has the effect of reducing the volume of redundant messages accessed when mapping the radial search area to a set of bins.

These optimisations, in combination, have been demonstrated to provide speedups as significant as 3x and 1.34x to the construction and query operations of FRNN search, respectively. However, the results in Chapter 6 showed that once applied in practice to a full simulation model the impact of this speedup to the simulation's overall runtime is reduced. This occurs as a consequence of underlying, model specific logic, both independent of and dependent on the FRNN search algorithm. Additionally these results demonstrated that the specific actor population's distribution and properties found within a given model have a significant impact on how well the optimisations perform.

The following section outlines model design criteria which can be applied to decide whether the optimisations presented can be expected to deliver an improvement and potentially the significance of this improvement.

The remainder of the chapter then presents a case study that explores two sample optimisation candidates from the field of pedestrian modelling. Despite both representing forms of transport and pedestrian modelling which rely on FRNN search, they have vastly different properties of relevance to optimisation suitability. For each case study, the optimisation suitability criteria are discussed, followed by a demonstration of the outcome of applying the optimisations alongside analysis.

The chapter is then concluded by high level recommendations regarding the optimisations and how they can be applied to real-world applications, based on what has been discussed in the chapter.

By performing the case study, the combined impact of the optimisations presented within this thesis can be viewed when applied to a real-world models. The overall value of the case study being an understanding of the wider implications, including how best to utilise newly available execution time, and justification for applying these optimisations to other viable applications.

7.2 Optimisation Suitability Criteria

The results from Chapters 4, 5 & 6 have shown that instances of high population, high density and high entropy create the conditions for the greatest bottlenecks to the performance of FRNN search, and consequently present the most desirable areas for optimisation.

The optimisations presented within this thesis primarily address performance bottlenecks, such that they are unlikely to show improvements to applications which do not utilise the GPU highly enough to reach the targeted bottlenecks.

Population Size

The primary GPU used throughout this thesis has been the Titan X (Pascal). This high end model (circa 2016) has 28 SMs containing a total of 3584 fp32 units. Therefore a minimum of 1792-3584 threads are required to fully saturate the device's compute capacity. In practice, workloads are rarely purely compute dependent, so this number can be expected to be several times higher.

It is for this reason results within this thesis have limited minimum populations to 10,000. These properties, however, differ between devices, such that smaller populations are more appropriate for lesser hardware. Similarly, the current trajectory of GPU development is likely to see the minimum appropriate population size increase.

The most recent GPU models to be released, Titan V and Titan RTX, have 5120 and 4608 fp32 units, respectively, with the latter also featuring 576 new tensor cores. At the time of writing, tensor cores are not yet available outside of ray tracing and deep learning applications.

However, lower populations may still benefit from the optimisations due to the reductions in memory access and operations. Additionally, in cases of reduced GPU performance/availability such as using lower end GPUs or partitioned GPUs as present in some recent HPC systems, the available capacity and hence requirements to maximise utilisation can be much lower.

Density

Calculating the minimum memory access volumes required to fully saturate the hardware cannot be performed as cleanly as identifying compute bottlenecks. This is both due to the different properties of the many caches on GPUs and how the order, or lack thereof, within memory accesses can affect performance. However, the relevant device properties such as Memory bus width and cache sizes have continued to increase through ongoing developments in GPU hardware, such that recommendations will vary to some degree between both GPU architecture and model.

Results from Chapter 5 & 6 have shown performance improvements for cases of 30 neighbours and higher. As density decreases below this point, the impact of bin changes becomes greater, relative to message accesses with low quantities of messages per bin. Therefore there is a case for using the Strips optimisation without the proportional bin width in instances of lowest density, due to the increase in bin accesses caused by the proportional bin width optimisation.

Entropy

In this context, entropy is used to describe the rate of change, or more precisely the degree to which actors become out of order during a single time step. This was mostly highlighted by the ACS optimisation to construction of the data structure, whereby the cost of reordering messages was greatest for randomly ordered messages.

Hence, the ACS construction optimisation is most suitable for populations with incremental movement, such that order is largely preserved. The vast majority of MAS involve incremental movement, so this is unlikely to be a significant consideration.

One potential edge case would be any models whereby most actors are moving fast enough to switch bin each timestep. However, as shown in the results of unsorted message construction (Chapter 4), this would only negatively affect application of the ACS optimisation to environments with around 700,000 bins, or potentially even higher as messages will still be loosely ordered.

Distribution

Due to the difficulty in classifying the differences in distributions of agents between models, and within different applications of models, there has not been a significant focus on distribution within this thesis. It has, however, been understood the SIMT architecture of GPUs favours uniformity, such that divergence between threads of the same warp requires instruction of divergent branches to be executed sequentially. Distribution does not map cleanly to divergence, however, in the context of FRNN unequal bin volumes has a noticeable impact on performance. Most significantly, a strictly uniform distribution can see runtimes as much as 2x faster. Though strictly uniform distributions are not likely to utilise FRNN search, as more appropriate techniques exist for discretely ordered actors, so it is unlikely that distribution will significantly impact use of the presented optimisations.

Furthermore, we have seen that limiting block size to 64 threads, as shown by Green[58], can improve performance by reducing the impact of divergent threads. In the case of FRNN search, this is primarily a consequence of bin membership not cleanly aligning with thread-block membership, such that all threads in a block must wait for the thread with the largest neighbourhood to finish its search.

Summary

The above optimisation suitability criteria – Population Size, Density, Entropy & Distribution – provide a clear structure by which to evaluate whether a candidate MAS for optimisation is likely to benefit from the work within this thesis. MAS with large agent populations, high density neighbourhoods and non-uniform distributions, all approach bottlenecks of the GPU hardware, such that the presented optimisations can aid performance. MAS which do not fit any of these criteria are unlikely to have performance that is constrained, and hence are unlikely to justify use of these targeted optimisations.

7.3 Case Studies

The following two case studies examine two simulators, which contain microscopic pedestrian simulations utilising FRNN search. Despite model similarities, their alignment with the optimisation suitability criteria proposed in this chapter differs significantly. They are first presented with regard to the optimisation suitability criteria, after which the results of applying the optimisations are demonstrated.

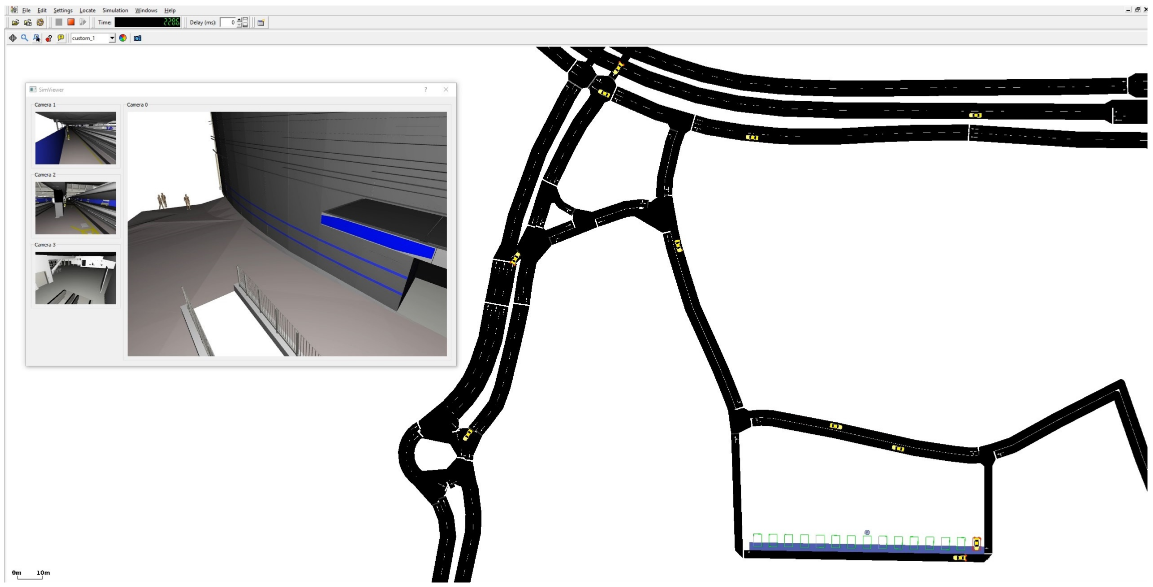


Figure 7.1: The visualisation interfaces of the road and pedestrian models within the multi-modal transport simulation. When vehicles enter the car park (bottom-right), pedestrians are able to move between the vehicle and the pedestrian simulation (top-left).

The applied optimisations do not impact accuracy. Due to the stochastic nature of the underlying parallel algorithms (e.g. atomic operations), compounded over thousands of iterations, no two runs are likely to produce identical results at a per agent level. Nonetheless, as a whole, outcomes will remain statistically equivalent.

To evaluate the impact of the optimisations, their normal operating conditions have been used. Each of the models was first executed for an internal time period of 10 mins. This is to adequately 'warm up' the models, allowing actors to fully distribute throughout the system and approach a steady state. After the models had been warmed up, 10,000 timesteps were executed and timed for both the original and optimised implementations, inclusive of finer grained timing of the elements directly impacted by FRNN search. Both models were executed on Windows 10 x64, using an Nvidia Titan-X (Pascal) GPU.

Case Study 1: Multi-modal Transport Simulation

This simulation is the result of an industrial project between Siemens Rail Automation and The University of Sheffield. The goal of this industrial project was to develop a prototype of a traveller focused multi-modal transport model, combining pedestrian, road and rail simulations.

The project sought to appropriately model known 'pinch points', such as an exit from stations onto a pavement and subsequently a road network, and passenger movement between platform and train. By using high performance, microscopic simulation techniques the combined model has the potential to support a wide range of applications in the fields of transport infrastructure and management.

The project consists of three independent simulation models, road, rail and pedestrian (Figure 7.1 shows the road and pedestrian interfaces), which communicate over the High Level Architecture (HLA) protocol to send and receive updates when actors reach transition points. For example, when a train nears a station, the rail model sends continuous location updates, which allows the train to move into the pedestrian model in synchronisation with the rail model and eventually hand over control of the train to the station. Similarly, when passengers have finished boarding, the pedestrian model sends a notification returning control back to the rail model, allowing the train to depart the station.

The pedestrian model within the multi-modal transport model is the target of the optimisations in this thesis. A central station with four platforms and two entrances across two floors is modelled via a modified FLAMEGPU. As the pedestrian simulation operates using FLAMEGPU, application of the optimisations can be completed similarly to earlier tests within this thesis, whereby FLAMEGPU was modified to support them.

The example scenario used for this case study combines 15 trains travelling between 9 stations arranged in a crucifix formation (Figure 7.2), with a pedestrian simulation of the central interchange station. Similarly, passengers can arrive and depart the central station via the model of the road network, representative of the central roads of a city with a population of 300,000 people.

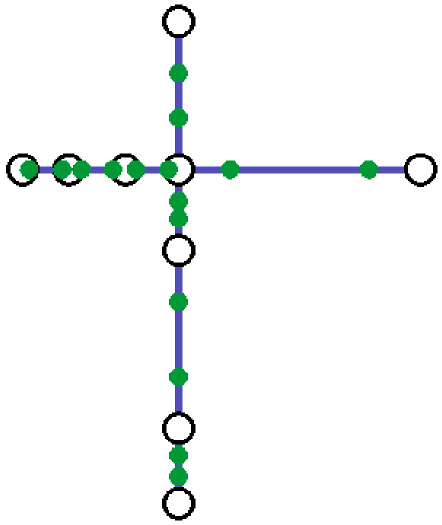


Figure 7.2: The crucifix arrangement of the rail network, with trains represented as green dots.

The multi-modal transport simulation has been implemented to execute the simulation in real-time to facilitate a live visualisation. The Rail and Road models execute with a fidelity of 1 step per second. The pedestrian model requires a higher fidelity for both the model and a smooth visualisation, so it operates at 30 steps per second. Therefore each step of the pedestrian simulation must be completed in less than 33ms.

The multi-modal transport model in regular operation has activity of around 300 concurrent pedestrians within the modelled train station. At peak demand this value rises to slightly over 1000. As such, the volume of actors within this model is not sufficiently bottlenecked to benefit from the optimisations presented in this thesis, despite small areas of density among a much larger environment.

	Time		
	Construction	**Query**	**Iteration**
Original (ms)	2.66	3.04	11.28
Optimised (ms)	3.16	5.53	13.95
Change (ms)	+0.5	+2.49	+2.67
Speedup	0.84x	0.55x	0.81x

Table 7.1: Results from applying the thesis's optimisation to the Multi-modal Transport Simulation. The model's low device utilisation leads to the cost of the optimisations outweighing improvements.

Optimisation Impact

Population size was measured at the start and end of the timing run. The initial and final population sizes were 244 and 282 respectively, as the population fluctuates as passengers enter and leave the train station. This population is well below the figure required for appropriate device utilisation.

Table 7.1 shows the results of applying the optimisation. As expected, the small population makes this model an unsuitable candidate for the optimisations. Runtimes increased for all directly affected functions, without any bottlenecks to address. The additional code instead merely increases the runtime cost. The construction cost is close to that of the FRNN search, which is indicative of a very low density model unsuitable for the optimisations presented in this thesis.

Additionally, the runtimes of other, unaffected components of the model also changed; for example the global navigation function's runtime decreased from $73.4ms$ to $45.0ms$. This is suggestive that at such low levels of device utilisation, minor changes can have a large impact on relative performance, as the model was shown to execute correctly after the optimisations were applied. This may be due to changes, such as how the proportional bin width optimisation increases the grid resolution and has a larger impact on how caches are utilised relative to the same change in a bottlenecked simulation.

A consequence of the low device utilisation, however, is that the pedestrian population could likely be increased 100-fold or greater without significantly impacting the overall performance of the model, dependent on how memory bound it is shown to be.

When passenger influx is increased to achieve a population averaging 1000 (peak demand), average iteration time increases to 20.8ms and 20.4ms for the Original and Optimised version, respectively. In this case, the Optimised version provides a slight improvement (1.02x). This is evidence that device utilisation is the limiting factor and that higher populations or less performant GPU hardware can be expected to increase the efficacy of the optimisations for this particular case study. However, this model is incapable of extending to greater population sizes, the density of passengers within the confined environment leads to their behaviour breaking down, rendering any further results invalid.

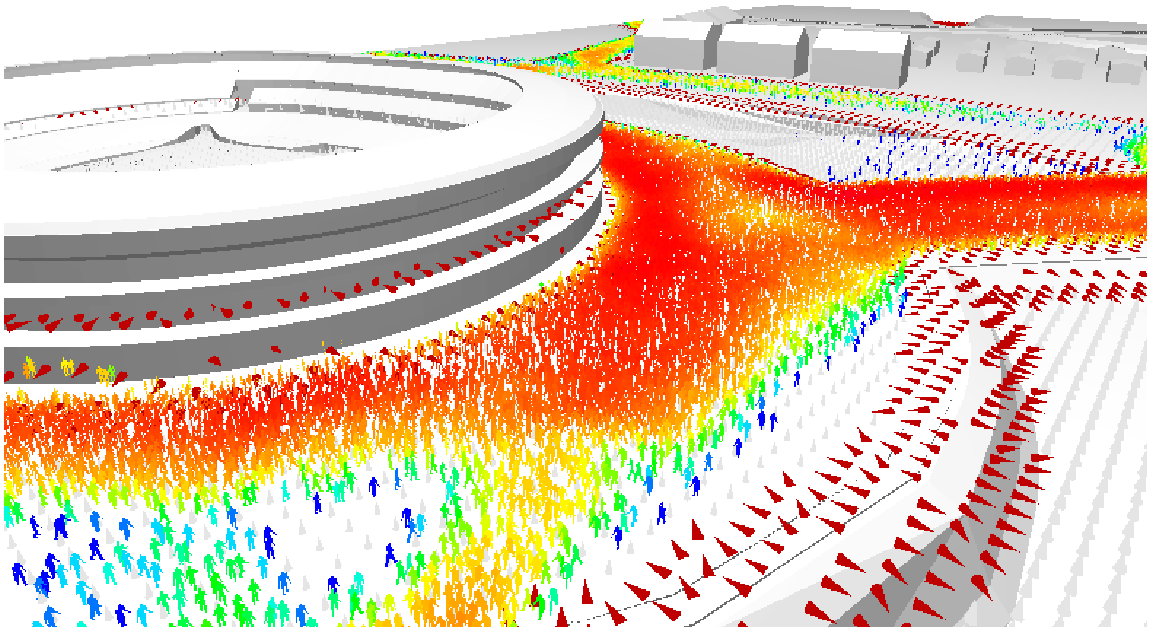


Figure 7.3: A visualisation of the FRUIN LOS modelled by the large event crowd simulation.

Case Study 2: Large Event Crowd Simulation

This project was developed to model the impact of crowd flow around the stadium of a large sporting event, and to visualise the Fruin LOS [52] measure to identify zones which could benefit from alteration (Visualisation shown in Figure 7.3).

Fruin LOS is a qualitative measure used to assess congestion from crowd density, assigning 6 coloured grades from the best case 'LOS A' (Blue) <=0.08 pedestrians per m^2, to 'LOS F' (Red) >=1.66 pedestrians per m^2. Different density scales exist for cases such as queuing and stairs. The implication is that LOS can be mapped to an assumed flow rate, which can be improved by removing bottlenecks and increasing capacity.

The project consists of an independent pedestrian simulation and visualisation of the urban environment surrounding the event venue. The model was developed using FLAMEGPU. As with the multi-modal transport simulation, the techniques from earlier in the thesis can be utilised to apply the optimisations to the model.

Whilst the large event crowd simulation can be executed in real-time with a demonstrative visualisation (Figure 7.3), it can also be executed in a headless batch mode to facilitate testing of FRUIN LOS in response to changes to the environment.

In regular operation the large event crowd model simulates as many as 100,000 pedestrians. The model is intended for highlighting impacts to FRUIN LOS, hence peak densities are often reached. As such, the volume and density of actors within this model satisfy two

	Average Time		
	Construction	**Query**	**Full Simulation Step**
Original (ms)	0.49	1.30	4.78
Optimised (ms)	0.37	1.01	4.31
Change (ms)	-0.12	-0.29	-0.47
Speedup	1.32x	1.28x	1.11x

Table 7.2: Results from applying the optimisations from this thesis to the Large Event Crowd Simulation.

of the identified properties of an appropriate optimisation candidate.

Optimisation Impact

Population size and neighbourhood size were measured at the start and end of the timing run. The initial population size was 62210 ± 30, which rose to the final population of 86739 ± 42. By reversing the FRUIN LOS calculation, the average neighbourhood size was calculated. Initially neighbourhoods consisted of 16.4 neighbours with a standard deviation of 9.0, by the end this had risen to 21.4 neighbours with a standard deviation of 11.4. The original and optimised simulations had the same neighbourhood properties, to 1 decimal place, at the points tested.

Table 7.2 shows the results of applying the optimisations. Most visible is an overall speedup to each step of $1.11x$. This is comprised of a 1.32x speedup to construction, and $1.28x$ speed up to the FRNN search, showing that the ACS optimisation to construction was able to more than overcome any additional overhead introduced by the proportional bin width optimisation.

The runtime of the other components of the model remained fairly stable, as would be expected, for example the stage which combined movement forces executed in $0.17ms$ for both models and the navigation structure export executed in $0.20ms$ and $0.19ms$, respectively.

These results confirm that the Large Event Crowd Simulation is an appropriate candidate for the optimisations presented within this thesis, due to its agent population and neighbourhood density. Results within this thesis have assessed populations and densities to higher levels that might be seen in biological or physics model, therefore it can be assumed that models with larger populations and densities would likely see the relative improvement increase. Additionally, models which have larger spatial items returned by FRNN queries are likely to respond in a similar way to increased density, due to them both increasing memory accesses and cache utilisation.

In total, the FRNN search functionality takes up 45% of the model's runtime within the Original unoptimised version. The next most expensive operation is the global navigation which costs around 34% of the total runtime, whilst simple, this model-specific behaviour has high levels of branching leading to its cost. However, this clarifies why the impact to overall model runtime (1.11*x*) is lower than that of the improvement to the individual FRNN search components (1.28*x*).

As this model is intended for batch execution, the impact of the 1.11x improvement to run time enables a greater throughput of simulations for any given period of time. As the model is already lightweight, the requirement for its intended visual framerate of 25FPS (40ms per frame) is already achievable prior to optimisation. Even VR requiring a higher frame rate of 90FPS (allowing a shorter 11ms per frame) is within its scope of operation.

7.4 Conclusion

This chapter has outlined optimisation suitability criteria and demonstrated their efficacy by presenting two contrasting applications containing pedestrian simulations, discussed their applicability to our three optimisations (reviewed in Section 7.1) and demonstrated how their performances each respond after the optimisations have been applied. Therefore providing contribution C4.

The Large Event Crowd Simulation case study demonstrates a positive result which shows how the optimisations are able to improve performance by 1.11x. Results throughout the thesis are suggestive that this could be higher or lower, dependent on the exact specifics of the candidate application, its FRNN search and how significant FRNN search is among the performance critical components.

By delivering a 1.11x speedup to the Large Event Crowd Simulation, a greater throughput of pedestrian simulations can be completed. This could be used to allow for greater environment or model complexity, or simply facilitate faster evolution within a genetic algorithm which uses the FRUIN LOS as a measure of health in optimising the design of the simulated environment.

Additionally, the Multi-modal Transport Simulation case has shown how the advanced optimisations presented within this thesis address performance bottlenecks, and as such are unsuitable for application to models which underutilise the GPU in use. It is important to demonstrate the harmful effects of premature optimisation both in that it can have an unintended effect on performance and often additionally reduce the readability and maintainability of algorithms.

The use of two simulations from the same background of pedestrian modelling has shown how even within a small field various factors within the application of a model can impact the suitability and need of optimisation to FRNN search. Within pedestrian modelling the environment has a huge impact on the distribution of actors, and similarly the context of the simulation will affect the population's overall size and density. These properties have all been shown, through results and the optimisation suitability criteria, to affect how beneficial the optimisations are to a candidate application.

FRNN search is used across a wide range of MAS domains. Visual models may have actor populations in the low hundreds, whereas models using cellular and particle scale actors will often run into the millions. Areas of influence (neighbourhood density) also vary significantly. Uniformly distributed actors may allow for a lower effective density for all actors to survey their nearest neighbours. However, models mixing sparse and densely populated areas, such as transport models mixing urban and rural areas, may suffer from a need to capture actors over a wider distance to allow all actors to have visibility of their closest neighbours. Similarly, entropy can also impact the applicability of a candidate application, although most applications relying on FRNN represent actors with incremental movement relative to the scale of their environment.

Whilst it is not possible to predict the exact impact of applying the optimisations from this thesis, due to the many dependent properties, the provided optimisation suitability criteria highlight the 4 factors (Population Size, Density, Entropy, Distribution) which act towards creating bottlenecks in the handling of FRNN search. At a high level these are suggestive of the absolute significance of FRNN search.

Chapter 8

Conclusions

The opening chapter of this thesis asks the question: What techniques can be used for improving the performance of FRNN searches during complex system simulations on GPUs? After carrying out the work documented within this thesis, this can now be answered.

It was first demonstrated in Chapter 2 that a common technique is used among complex system simulations to provide FRNN search, and that a general case for the operation of FRNN search within complex system simulations could be defined. across a range of fields utilising FRNN search within complex system simulations, a number of properties remain within a common range. The experiments carried out have demonstrated, for this general case [167, 35] that the performance of FRNN search can be improved with a number of techniques. Furthermore, these techniques have been applied to real world models within FLAMEGPU to demonstrate the scope of impact when assessed with regards to the overall runtime of a model.

The investigation carried out into the construction of the USP data structure showed that existing researchers had addressed the sorting of the elements. These had limited success, only providing improvements in very small populations [149], or reducing the query operation to an approximate method [85].

An understanding of the recent developments in GPU hardware directed explorations towards atomic counting sort. By utilising this sorting algorithm, a more performant construction of the USP data structure can be delivered. The experiments carried out showed in all instances present within the general case that atomic counting sort was able to speed up construction of the data structure, with peak improvements of 2.5x, hence providing contribution C1.

The most effective technique discovered for optimising FRNN queries within the existing literature is that of restricting the block size. This technique, demonstrated by Green's seminal introduction to particle simulation on GPUs [58], reduces the impact of block level divergence within the FRNN query algorithm.

The knowledge of the high rates of divergence within the FRNN query operation led to the Strips optimisation, providing contribution C2. This optimisation combines accesses to bins which share boundaries in memory. This has the effect of reducing the number of times the bin being accessed has to change. This removal of operations results in a 35% reduction in warp divergence, leading to a peak speedup of 1.2x to the FRNN query. Experiments suggested that the improvement was closely linked to the number of bin changes removed rather than size of bins or population size. This demonstrates the significant cost of warp divergence to GPU algorithms.

Many earlier techniques attempt to improve the coherence and spatial locality of memory accesses [56, 141, 140]. However, this work carried out on the earliest GPGPU capable GPU architectures, such as Fermi and Kepler, has been superseded in modern architectures, such as Pascal and Volta, by improvements to compiler optimisations and caching strategy. This benefits all GPGPU developers, reducing attention that need be paid to memory access coherence when reading data.

To provide contribution C3 another novel technique to improve the performance of FRNN queries was identified, reducing redundant memory accesses. This technique considered the cost of surplus message reads due to the difference between the radial search area and that of the Moore neighbourhood search grid. It was found that changing to a proportional bin width of 0.5 then requires a 5x5 Moore neighbourhood search grid. In combination with the earlier Strips optimisation, reducing the bin accesses from 25 to 5, a peak speedup of 1.32x was obtained.

These three optimisations are novel and have been demonstrated to provide benefit throughout the properties of the general case of FRNN search that was identified. Therefore, they can benefit a wide range of complex system simulations by providing improved performance. Faster execution of simulations facilitates larger and more complex models. Furthermore, in some cases, faster performance may allow models to be utilised in new ways, e.g. operating at speeds appropriate for real-time visualisation.

The impact of optimisations to full complex system models was demonstrated using FLAMEGPU, providing contribution C4. These have both demonstrated the high cost of FRNN search relative to model specific logic and the impact this has on reducing the proportional speedup to total runtime. These complex system models have also provided further evidence of the properties of FRNN search that were identified as the general case.

Additionally, the research has surfaced and condensed knowledge relating to other techniques for handling spatial data. Chapter 2's review of literature provides a condensed overview of approaches used in the handling of spatial data. In particular it highlights Octrees as the data structure of preference of the similar k-nearest neighbours algorithm, and the performance benefits present if instead performing approximate methods of FRNN search. With regards to the non-approximate version of FRNN search, the focus of this thesis, the many names used to refer to the technique of FRNN search were identified. Furthermore, the shortcomings of earlier approaches when applied to modern GPU hardware were highlighted.

The nature of the proposed optimisations, in particular how they target divergence and redundant memory accesses, in combination with increasing demand for model scale and complexity suggests they will remain relevant to future GPU architectures. The only limiting factor for applicability is that of high device utilisation, which is unlikely to disappear due to the signs that Moore's law is beginning to stall. Furthermore, there is often a significant lag time (years) between the release of new GPU architectures and their wide availability via HPC systems due to the cost of upgrading infrastructure.

In summary, three optimisations benefiting FRNN search have been proposed and demonstrated: Atomic Counting Sort for construction of the USP data structure, and Strips and Proportional Bin Width for FRNN queries. These optimisations respectively take advantage of modern improvements to GPU hardware, reduce divergent code and limit redundant memory accesses and address contributions C1, C2 & C3. These have been demonstrated to improve the performance of FRNN search under both benchmark and full complex system models, which addresses contribution C4. In combination these optimisations are able to improve the performance of construction by as much as 2x, and FRNN queries by as much as 1.32x. Therefore, the answer to the question asked at the start of this thesis is that performance of FRNN search can be improved through acknowledging advances in GPU technology and addressing divergence and redundancy.

8.1 Future Work

This thesis has delivered results using the recent Pascal and Volta GPU architectures. It has exposed differences in these modern GPU architectures, not present in earlier Kepler GPUs. Therefore, there remains an open question of how future innovations to GPU hardware, particularly in the domain of task-specific architectures, will impact the performance of FRNN search and complex system simulations. Currently task specific architecture targets neutral networks (matrix multiplication) and ray tracing (octree traversal). CUDA availability of this ray tracing specific hardware may have large implications for the performance of FRNN search, as octrees have already been demonstrated as highly performant for the similar task of k nearest neighbours. Furthermore, future architectures are likely to be specialised as yet unknown tasks, which could be of benefit.

Similarly, results have been confined to FRNN on a single GPU. At this early stage, most complex system simulations are yet to approach the scale necessary to require more than a single GPU. However, it is likely that in time models will grow in scale, e.g. biological simulations where the number of cells in bodily organs range in the billions. Performing an algorithm across multiple GPUs or nodes introduces higher latencies for communication. This is likely particularly harmful for FRNN queries and agent motion across the partitions into which a model has been divided between GPUs. MD has already made this leap, producing specialised techniques for performing FRNN search across multiple GPUs. However, MD entails a restricted set of properties which may affect performance, hence it is also an open question as to whether MD techniques for multi-GPU FRNN search are applicable to other domains of complex system simulation, or these wider properties necessitate their own approaches.

Two approaches not considered within this thesis are those of restricting the radius or Moore neighbourhood, for example using a limited field of view (e.g. pedestrians cannot see behind) or a hexagonal decomposition of the environment. Early analysis of these approaches suggested they were not as straightforward as initially thought. A 180° field of view from many search origin/directions may only skip a single bin, and hence the additional computation and divergence is likely to outweigh any improvements. Similarly, a hexagonal decomposition may lead to a surplus access 4.79x the search radius in 2D, which is greater than the original square grids (2.86x). It is possible that an alternative, regular environmental decomposition, or even storage format, exists to facilitate more precise bin access. However, it remains an open question as to whether one does and at what cost (e.g. additional memory) or whether the inverse can be mathematically proven.

An inverse approach to that taken by this thesis, would be to consider whether other, more performant techniques also merit use. Whilst FRNN appears to dominate most complex system simulations (and supporting frameworks), choice over the applicability of instead using k nearest neighbours or approximate techniques is highly specific to individual models. The best outcome for modellers of complex systems would be for general frameworks for complex system modelling (such as FLAMEGPU) to facilitate multiple options for spatial searches through a common interface. This would allow modellers to easily evaluate each technique and consider the impacts to the performance and results precision to decide the appropriate choice for their model.

Glossary

atomic An operation whereby a processor can guarantee that read, modify and write actions will occur in serial without interference from concurrent threads. These operations are usually limited to simple types and instructions such as add, min, max & exchange. 31, 35

coherence The act of being in a logical order. A spatially coherent data-structure is therefore a data-structure whereby elements are stored in an order similar to how they naturally appear. 31, 32

Hamming distance While comparing two binary strings of equal length, Hamming distance is the number of bit positions in which the two bits are different. 14

hyperplane A plane which which is one dimension less than the space in which it exists. If a space is 3-dimensional, then it's hyperplanes are 2-dimensional planes. 16

load-factor A hashing data-structure statistic calculated as, the number of entries divided by the maximum number of entries (without resizing). 12–14

memory coalescing When multiple memory accesses are combined in a way such that an entire single transaction is used. For example with a 128 byte bus, it would be possible to read 32 neighbouring 4-byte integers in a single transaction. This is achieved in GPGPU programming by having neighbouring threads access neighbouring locations in memory simultaneously. 27

memory fence An operation which is used as a barrier to enforce a degree if order among concurrent threads. The usage ensures all memory accesses before the memory fence have been executed prior to those after the fence being started. 35

Moore neighbourhood Within a uniform 2D grid, the Moore neighbourhood of a cell is the cell itself and the 8 adjacent cells. This can be extended to higher dimensonalities, for example in 3D 27 cells form the Moore neighbourhood. This is known as a range 1 Moore neighbourhood, the process can be applied recursively to define higher range Moore neighbourhoods.. 44, 49, 84–87, 97, 108, 114, 127, 129

on-chip Refers to memory and caches stored on the same chip as the processor, these offer the highest throughput and lowest latencies however are often of limited size. 23, 34, 35

perfect hash A hash function capable of mapping a set of distinct elements into a set of distinct hashes with no collisions. Performed with known data, these are able to reach a maximal load factor. 13, 31

race condition A condition whereby the output of a concurrent block of code is dependant on the sequence in which the concurrent threads execute. 31

triangular numbers Numbers of the sequence: 1,3,6,10. They are calculated by summing the integers between (inclusive) 1 and the desired index (1 based index), this can be more simply formulated as $T_n = \frac{n(n+1)}{2}$. 13

Bibliography

[1] Nvidia's next generation cuda™ compute architecture: Kepler™ gk110. http://www.nvidia.co.uk/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, 2012.

[2] Cub documentation. `https://nvlabs.github.io/cub/index.html`, 11 2015.

[3] Cuda data parallel primitives library. `http://cudpp.github.io/`, 11 2015.

[4] Nvidia nvlink: High speed interconnect. `http://www.nvidia.com/object/nvlink.html`, 11 2015.

[5] Thrust. `https://thrust.github.io/`, 11 2015.

[6] Message passing interface forum. `http://www.mpi-forum.org/`, 1 2016.

[7] Openacc: Directives for accelerators. `http://www.openacc.org/`, 1 2016.

[8] Flamegpu: Flexible large scale agent modelling environment for the gpu. `http://www.flamegpu.com/`, 1 2018.

[9] Fluids v.3: A large scale open source fluid simulator. `http://fluids3.com/`, 1 2018.

[10] Nvidia maxwell architecture. `https://developer.nvidia.com/maxwell-compute-architecture`, 1 2018.

[11] Top500 supercomputer sites. `http://https://www.top500.org//`, 1 2019.

[12] M Adelson-Velskii and Evgenii Mikhailovich Landis. An algorithm for the organization of information. Technical report, DTIC Document, 1963.

[13] Highways Agency. North east and yorkshire & humber regional modelling library. `http://assets.highways.gov.uk/specialist-information/guidance-and-best-practice-regional-planning/NEYH%20Model%20Database%20June%202013.pdf`, 1 2016.

[14] Sadaf R Alam, Pratul K Agarwal, Melissa C Smith, Jeffrey S Vetter, and David Caliga. Using fpga devices to accelerate biomolecular simulations. *Computer*, 40(3):66–73, 2007.

[15] Siamak Alimirzazadeh, Ebrahim Jahanbakhsh, Audrey Maertens, Sebastián Leguizamón, and François Avellan. Gpu-accelerated 3-d finite volume particle method. *Computers & Fluids*, 171:79–93, 2018.

[16] David P Anderson. Boinc: A system for public-resource computing and storage. In *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pages 4–10. IEEE, 2004.

[17] Joshua A Anderson, Eric Jankowski, Thomas L Grubb, Michael Engel, and Sharon C Glotzer. Massively parallel monte carlo for many-particle simulations on gpus. *Journal of Computational Physics*, 254:27–38, 2013.

[18] Nathan Andrysco and Xavier Tricoche. Matrix trees. In *Computer Graphics Forum*, volume 29, pages 963–972. Wiley Online Library, 2010.

[19] Nathan Andrysco and Xavier Tricoche. Implicit and dynamic trees for high performance rendering. In *Proceedings of Graphics Interface 2011*, pages 143–150. Canadian Human-Computer Communications Society, 2011.

[20] Lars Arge, Mark De Berg, Herman J Haverkort, and Ke Yi. The priority r-tree: A practically efficient and worst-case optimal r-tree. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 347–358. ACM, 2004.

[21] Shibdas Bandyopadhyay and Sartaj Sahni. Grs—gpu radix sort for multifield records. In *High Performance Computing (HiPC), 2010 International Conference on*, pages 1–10. IEEE, 2010.

[22] Jaime Barceló and Jordi Casas. Dynamic network simulation with aimsun. In *Simulation approaches in transportation analysis*, pages 57–98. Springer, 2005.

[23] Julien Basch and Leonidas J Guibas. *Kinetic data structures.* Citeseer, 1999.

[24] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. *The R*-tree: an efficient and robust access method for points and rectangles*, volume 19. ACM, 1990.

[25] Nathan Bell and Jared Hoberock. Thrust: A productivity-oriented library for cuda. In *GPU computing gems Jade edition*, pages 359–371. Elsevier, 2011.

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

[27] Ansel L Blumers, Yu-Hang Tang, Zhen Li, Xuejin Li, and George E Karniadakis. Gpu-accelerated red blood cells simulations with transport dissipative particle dynamics. *Computer physics communications*, 217:171–179, 2017.

[28] GE Bradley. A proposed mathematical model for computer prediction of crowd movements and their associated risks. *Engineering for crowd safety. Amsterdam: Elsevier*, pages 303–11, 1993.

[29] Lukas Breitwieser, Roman Bauer, Alberto Di Meglio, Leonard Johard, Marcus Kaiser, Marco Manca, Manuel Mazzara, Fons Rademakers, and Max Talanov. The biodynamo project: Creating a platform for large-scale reproducible biological simulations. *arXiv preprint arXiv:1608.04967*, 2016.

[30] Martin Burtscher and Keshav Pingali. An efficient cuda implementation of the tree-based barnes hut n-body algorithm. *GPU computing Gems Emerald edition*, page 75, 2011.

[31] Yuan Cao, Heng Qi, Wenrui Zhou, Jien Kato, Keqiu Li, Xiulong Liu, and Jie Gui. Binary hashing for approximate nearest neighbor search on big data: A survey. *IEEE Access*, 6:2039–2054, 2017.

[32] Kristof Carlier, Stella Fiorenzo-Catalano, Charles Lindveld, and Piet Bovy. A supernetwork approach towards multimodal travel modeling. In *82nd Annual Meeting of the Transportation Research Board*, 2003.

[33] Daniel Cederman and Philippas Tsigas. Gpu-quicksort: A practical quicksort algorithm for graphics processors. *Journal of Experimental Algorithmics (JEA)*, 14:4, 2009.

[34] Pedro Celis. *Robin Hood Hashing*. PhD thesis, University of Waterloo, 1986.

[35] John Charlton, Luis Rene Montana Gonzalez, Steve Maddock, and Paul Richmond. Fast simulation of crowd collision avoidance. In *Computer Graphics International Conference*, pages 266–277. Springer, 2019.

[36] Mozhgan K Chimeh and Paul Richmond. Simulating heterogeneous behaviours in complex systems on gpus. *Simulation Modelling Practice and Theory*, 83:3–17, 2018.

[37] Robert Chisholm, Steve Maddock, and Paul Richmond. Improved gpu near neighbours performance for multi-agent simulations. *Journal of Parallel and Distributed Computing*, 137:53 – 64, 2020.

[38] Robert Chisholm, Paul Richmond, and Steve Maddock. A standardised benchmark for assessing the performance of fixed radius near neighbours. In *European Conference on Parallel Processing*, pages 311–321. Springer, 2016.

[39] Gennaro Cordasco, Rosario De Chiara, Ada Mancuso, Dario Mazzeo, Vittorio Scarano, and Carmine Spagnuolo. A framework for distributing agent-based simulations. In *European Conference on Parallel Processing*, pages 460–470. Springer, 2011.

[40] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Counting sort. In *Introduction to Algorithms*, chapter 8.2, pages 168–170. MIT Press, 2 edition, 2001.

[41] Marcelo de Gomensoro Malheiros and Marcelo Walter. Spatial sorting: an efficient strategy for approximate nearest neighbor searching. *Computers & Graphics*, 57:112–126, 2016.

[42] Jure Demšar, Will Blewitt, and Iztok Lebar Bajec. A hybrid model for simulating grazing herds in real time. *Computer Animation and Virtual Worlds*, page e1914, 2019.

[43] Andreas Dietrich, Ingo Wald, Carsten Benthin, and Philipp Slusallek. The openrt application programming interface–towards a common api for interactive ray tracing. In *Proceedings of the 2003 OpenSG Symposium*, pages 23–31. Citeseer, 2003.

[44] Ilya Dmitrenok, Viktor Drobnyy, Leonard Johard, and Manuel Mazzara. Evaluation of spatial trees for the simulation of biological tissue. In *2017 31st International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, pages 276–282. IEEE, 2017.

[45] José M Domínguez, Alejandro JC Crespo, Daniel Valdez-Balderas, Benedict D Rogers, and Moncho Gómez-Gesteira. New multi-gpu implementation for smoothed particle hydrodynamics on heterogeneous clusters. *Computer Physics Communications*, 184(8):1848–1860, 2013.

[46] Roshan M D'Souza, Mikola Lysenko, and Keyvan Rahmani. Sugarscape on steroids: simulating over a million agents at interactive rates. In *Proceedings of Agent conference. 2007 Chicago, IL*, 2007.

[47] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. A comprehensive performance comparison of cuda and opencl. *Parallel Processing (ICPP), 2011 IEEE International Conference on*, pages 216–225, 2011.

[48] Kasper Fauerby. Crowds in hitman: Absolution. In *Game Developers Conference*. IO Interactive, 2012. `http://www.gdcvault.com/play/1015315/Crowds-in-Hitman`.

[49] Neetu Faujdar and Satya Prakash Ghrera. A practical approach of gpu bubble sort with cuda hardware. In *2017 7th International Conference on Cloud Computing, Data Science & Engineering-Confluence*, pages 7–12. IEEE, 2017.

[50] Martin Fellendorf. Vissim: A microscopic simulation tool to evaluate actuated signal control including bus priority. In *64th Institute of Transportation Engineers Annual Meeting*, pages 1–9, 1994.

[51] Raphael A. Finkel and Jon Louis Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4(1):1–9, 1974.

[52] John J. Fruin. Pedestrian planning and design. *Metropolitan Association of Urban Designers and Environmental Planners*, 1971.

[53] Henry Fuchs, Zvi M Kedem, and Bruce F Naylor. On visible surface generation by a priori tree structures. In *ACM Siggraph Computer Graphics*, volume 14, pages 124–133. ACM, 1980.

[54] Robert A Gingold and Joseph J Monaghan. Smoothed particle hydrodynamics: theory and application to non-spherical stars. *Monthly notices of the royal astronomical society*, 181(3):375–389, 1977.

[55] Jens Glaser, Trung Dac Nguyen, Joshua A Anderson, Pak Lui, Filippo Spiga, Jaime A Millan, David C Morse, and Sharon C Glotzer. Strong scaling of general-purpose molecular dynamics simulations on gpus. *Computer Physics Communications*, 192:97–107, 2015.

[56] Prashant Goswami, Philipp Schlegel, Barbara Solenthaler, and Renato Pajarola. Interactive sph simulation and rendering on the gpu. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 55–64. Eurographics Association, 2010.

[57] Scott Grauer-Gray, William Killian, Robert Searles, and John Cavazos. Accelerating financial applications on the gpu. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, pages 127–136. ACM, 2013.

[58] Simon Green. Particle simulation using cuda. *NVIDIA whitepaper*, 6:121–128, 2010.

[59] Leo J Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science*, pages 8–21. IEEE, 1978.

[60] Leonidas J Guibas. Kinetic data structures–a state of the art report. 1998.

[61] RY Guo and Hai-Jun Huang. A mobile lattice gas model for simulating pedestrian evacuation. *Physica A: Statistical Mechanics and its Applications*, 387(2):580–586, 2008.

[62] Stephen Gustafson, Hemagiri Arumugam, Paul Kanyuk, and Michael Lorenzen. Mure: fast agent based crowd simulation for vfx and animation. In *ACM SIGGRAPH 2016 Talks*, page 56. ACM, 2016.

[63] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. *SIGMOD Rec.*, 14(2):47–57, June 1984.

[64] Stephen J Guy, Sean Curtis, Ming C Lin, and Dinesh Manocha. Least-effort trajectories lead to emergent crowd behaviors. *Physical review E*, 85(1):016110, 2012.

[65] M Hall and LG WILLUMSEN. Saturn-a simulation-assignment model for the evaluation of traffic management schemes. *Traffic Engineering & Control*, 21(4), 1980.

[66] Takahiro Harada, Seiichi Koshizuka, and Yoichiro Kawaguchi. Smoothed particle hydrodynamics on gpus. In *Computer Graphics International*, pages 63–70. SBC Petropolis, 2007.

[67] M. Harris. Cuda pro tip: Occupancy api simplifies launch configuration. `http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-occupancy-api-simplifies-launch-configuration/`, 11 2015.

[68] Mark Harris, Shubhabrata Sengupta, and John D Owens. Parallel prefix sum (scan) with cuda. *GPU gems*, 3(39):851–876, 2007.

[69] Ben Harwood and Tom Drummond. Fanng: Fast approximate nearest neighbour graphs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5713–5722, 2016.

[70] Yoshinori Hashimoto and Shunya Noda. Pricing of mining asic and its implication to the high volatility of cryptocurrency prices. *Available at SSRN*, 2019.

[71] Dirk Helbing, Illés Farkas, and Tamas Vicsek. Simulating dynamical features of escape panic. *Nature*, 407(6803):487–490, 2000.

[72] Dirk Helbing and Peter Molnár. Social force model for pedestrian dynamics. *Physical review E*, 51(5), May 1995.

[73] LF Henderson. The statistics of crowd fluids. *Nature*, 229:381–383, 1971.

[74] Martin C Herbordt, Tom VanCourt, Yongfeng Gu, Bharat Sukhwani, Al Conti, Josh Model, and Doug DiSabello. Achieving high performance with fpga-based computing. *Computer*, 40(3):50–57, 2007.

[75] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch hashing. In *Distributed Computing*, pages 350–364. Springer, 2008.

[76] John Hershberger. Smooth kinetic maintenance of clusters. *Computational Geometry*, 31(1-2):3–30, 2005.

[77] RC Hoetzlein. Fast fixed-radius nearest neighbors: Interactive million–particle fluids. In *GPU Technology Conference*, 2014.

[78] Kaixi Hou, Weifeng Liu, Hao Wang, and Wu-chun Feng. Fast segmented sort on gpus. In *Proceedings of the International Conference on Supercomputing*, page 12. ACM, 2017.

[79] Wenfeng Hou, Daiwei Li, Chao Xu, Haiqing Zhang, and Tianrui Li. An advanced k nearest neighbor classification algorithm based on kd-tree. In *2018 IEEE International Conference of Safety Produce Informatization (IICSPI)*, pages 902–905. IEEE, 2018.

[80] Michael P Howard, Joshua A Anderson, Arash Nikoubashman, Sharon C Glotzer, and Athanassios Z Panagiotopoulos. Efficient neighbor list calculation for molecular simulation of colloidal systems using graphics processing units. *Computer Physics Communications*, 203:45–52, 2016.

[81] Michael P Howard, Athanassios Z Panagiotopoulos, and Arash Nikoubashman. Efficient mesoscale hydrodynamics: Multiparticle collision dynamics with massively parallel gpu acceleration. *Computer Physics Communications*, 230:10–20, 2018.

[82] Hong-Jun Jang, Byoungwook Kim, and Soon-Young Jung. k-nearest reliable neighbor search in crowdsourced lbss. *International Journal of Communication Systems*, page e4097, 2019.

[83] Liheng Jian, Weidong Yi, and Ying Liu. Fast on-chip quad-trees on gpu. In *Information Science and Control Engineering 2012 (ICISCE 2012), IET International Conference on*, pages 1–5. IET, 2012.

[84] Jaemin Jo, Jinwook Seo, and Jean-Daniel Fekete. A progressive kd tree for approximate k-nearest neighbors. In *2017 IEEE Workshop on Data Systems for Interactive Analysis (DSIA)*, pages 1–5. IEEE, 2017.

[85] Mark Joselli, José Ricardo da S Junior, Esteban W Clua, Anselmo Montenegro, Marcos Lage, and Paulo Pagliosa. Neighborhood grid: A novel data structure for fluids animation with gpu computing. *Journal of Parallel and Distributed Computing*, 75:20–28, 2015.

[86] Mark Joselli, Erick Baptista Passos, Marcelo Zamith, Esteban Clua, Anselmo Montenegro, and Bruno Feijó. A neighborhood grid data structure for massive 3d crowd simulation on gpu. In *2009 VIII Brazilian Symposium on Games and Digital Entertainment*, pages 121–131. IEEE, 2009.

[87] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12. IEEE, 2017.

[88] Ibrahim Kamel and Christos Faloutsos. Parallel r-trees. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, SIGMOD '92, pages 195–204, New York, NY, USA, 1992. ACM.

[89] Ibrahim Kamel and Christos Faloutsos. On packing r-trees. In *Proceedings of the second international conference on Information and knowledge management*, pages 490–499. ACM, 1993.

[90] Kamran Karimi, Neil G Dickson, and Firas Hamze. A perfomance comparison of cuda and opencl. *arXiv:1005.2581V3*, 2011.

[91] Twin Karmakharm, Paul Richmond, and Daniela M. Romano. Agent-based large scale simulation of pedestrians with adaptive realistic navigation vector fields. *Theory and Practice of Computer Graphics, The Eurographics Association 2010*, pages 66–74, 2010.

[92] Mariam Kiran, Paul Richmond, Mike Holcombe, Lee Shawn Chin, David Worth, and Chris Greenough. Flame: simulating large populations of agents on parallel hardware architectures. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, pages 1633–1636. International Foundation for Autonomous Agents and Multiagent Systems, 2010.

[93] Donald E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley Professional, 2nd edition, 1998. Answers to Exercises: Chapter 6.4, Exercise 20 (Page 731).

[94] G Kotusevski and KA Hawick. A review of traffic simulation software. 2009.

[95] Loudon Kyle. *Mastering Algorithms with C*, chapter 8, pages 144–145. O'Reilly & Associates, 1999.

[96] Thomas Larsson and Tomas Akenine-Möller. A dynamic bounding volume hierarchy for generalized collision detection. *Computers & Graphics*, 30(3):450–459, 2006.

[97] Victor W Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, et al. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 451–460. ACM, 2010.

[98] Sylvain Lefebvre and Hugues Hoppe. Perfect spatial hashing. In *ACM Transactions on Graphics (TOG)*, volume 25, pages 579–588. ACM, 2006.

[99] Francesco Lettich, Salvatore Orlando, and Claudio Silvestri. Processing streams of spatial k-nn queries and position updates on manycore gpus. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 26. ACM, 2015.

[100] Scott T Leutenegger, Mario Lopez, Jeffrey Edgington, et al. Str: A simple and efficient algorithm for r-tree packing. In *Data Engineering, 1997. Proceedings. 13th International Conference on*, pages 497–506. IEEE, 1997.

[101] David Levinthal. *Performance Analysis Guide for Intel® Core™ i7 Processor and Intel® Xeon™ 5500 processors*, page 22. Intel.

[102] Tyson J Lipscomb, Anqi Zou, and Samuel S Cho. Parallel verlet neighbor list algorithm for gpu-optimized md simulations. In *Proceedings of the ACM Conference on Bioinformatics, Computational Biology and Biomedicine*, pages 321–328. ACM, 2012.

[103] Elvis S Liu. On the scalability of agent-based modeling for medical nanorobotics. In *2015 Winter Simulation Conference (WSC)*, pages 1427–1435. IEEE, 2015.

[104] Elvis S Liu and Georgios K Theodoropoulos. An approach for parallel interest matching in distributed virtual environments. In *Proceedings of the 2009 13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*, pages 57–65. IEEE Computer Society, 2009.

[105] Leon B Lucy. A numerical approach to the testing of the fission hypothesis. *The astronomical journal*, 82:1013–1024, 1977.

[106] Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, Keith Sullivan, and Gabriel Balan. Mason: A multiagent simulation environment. *Simulation*, 81(7):517–527, 2005.

[107] Lijuan Luo, Martin Wong, and Wen-mei Hwu. An effective gpu implementation of breadth-first search. In *Proceedings of the 47th design automation conference*, pages 52–55. ACM, 2010.

[108] Lijuan Luo, Martin DF Wong, and Lance Leong. Parallel implementation of r-trees on the gpu. In *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, pages 353–358. IEEE, 2012.

[109] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. Nvidia tensor core programmability, performance & precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 522–531. IEEE, 2018.

[110] Donald JR Meagher. Octree encoding: A new technique for the representation, manipulation and display of arbitrary 3-d objects by computer. Technical Report IPL-TR-80-111, Image Processing Laboratory, Electrical and Systems Engineering Department, Rensseiaer Polytechnic Institute, 1980.

[111] Ramin Mehran, Akira Oyama, and Mubarak Shah. Abnormal crowd behavior detection using social force model. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 935–942. IEEE, 2009.

[112] Bruce Merry. A performance comparison of sort and scan libraries for gpus. *Parallel Processing Letters*, 25(04):1550007, 2015.

[113] Paulius Micikevicius. Local memory and register spilling. In *GPU Technology Conference*, 2011. `http://on-demand.gputechconf.com/gtc-express/2011/presentations/register_spilling.pdf`.

[114] Paulius Micikevicius. Performance optimization: Programming guidelines and gpu architecture reasons behind them. In *GPU Technology Conference*, 2013.

[115] Athanasios Mokos, Benedict D Rogers, Peter K Stansby, and José M Domínguez. Multi-phase sph modelling of violent hydrodynamics on gpus. *Computer Physics Communications*, 196:304–316, 2015.

[116] JJ Monaghan. Smoothed particle hydrodynamics and its diverse applications. *Annual Review of Fluid Mechanics*, 44:323–346, 2012.

[117] Michael J North, Nicholson T Collier, Jonathan Ozik, Eric R Tatara, Charles M Macal, Mark Bragen, and Pam Sydelko. Complex adaptive systems modeling with repast simphony. *Complex adaptive systems modeling*, 1(1):1–26, 2013.

[118] NVIDIA. Tuning cuda applications for maxwell. `http://docs.nvidia.com/cuda/maxwell-tuning-guide/`, 11 2015.

[119] NVIDIA. Nvidia nvidia tesla p100 whitepaper. Technical report, NVIDIA, 2016.

[120] NVIDIA. Nvidia tesla v100 gpu architecture whitepaper. Technical report, NVIDIA, 2017.

[121] NVIDIA. Nvidia turing gpu architecture whitepaper. Technical report, NVIDIA, 2018.

[122] NVIDIA. Tuning cuda applications for volta. `http://docs.nvidia.com/cuda/volta-tuning-guide/`, 11 2019.

[123] Institute of Transportation Systems at the German Aerospace Center. Github: Simulation of urban mobility (sumo). `https://github.com/planetsumo/sumo/tree/master/sumo/src/microsim`, 11 2015.

[124] Kazuhiko Ohno, Tomoki Nitta, and Hiroto Nakai. Sph-based fluid simulation on gpu using verlet list and subdivided cell-linked list. In *Computing and Networking (CANDAR), 2017 Fifth International Symposium on*, pages 132–138. IEEE, 2017.

[125] Wendy Osborn and Ken Barker. The 2dr-tree: a 2-dimensional spatial access method. 2004.

[126] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[127] Rasmus Pagh. Locality-sensitive hashing without false negatives. In *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 1–9. SIAM, 2016.

[128] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[129] Szilárd Páll and Berk Hess. A flexible algorithm for calculating pair interactions on simd architectures. *Computer Physics Communications*, 184(12):2641–2650, 2013.

[130] M Patella, P Ciaccia, and P Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proc. of the International Conference on Very Large Databases (VLDB). Athens, Greece*, pages 1241–1253, 1997.

[131] Nuria Pelechano, Jan M Allbeck, and Norman I Badler. Controlling individual agents in high-density crowd simulation. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 99–108. Eurographics Association, 2007.

[132] Gay Jane Perez, Giovanni Tapang, May Lim, and Caesar Saloma. Streaming, disruptive interference and power-law behavior in the exit dynamics of confined pedestrians. *Physica A: Statistical Mechanics and its Applications*, 312(3):609–618, 2002.

[133] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Stackless kd-tree traversal for high performance gpu ray tracing. In *Computer Graphics Forum*, volume 26, pages 415–424. Wiley Online Library, 2007.

[134] Zahed Rahmati and Timothy M Chan. A clustering-based approach to kinetic closest pair. *Algorithmica*, pages 1–15, 2018.

[135] Craig Reynolds. Big fast crowds on ps3. In *Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*, pages 113–121. ACM, 2006.

[136] Craig W. Reynolds. Flocks, herds and schools: A distributed behavioral model. *ACM SIGGRAPH Computer Graphics*, 21(4):25–34, 1987.

[137] Paul Richmond and Daniela Romano. Template-driven agent-based modeling and simulation with cu da. *GPU Computing Gems Emerald Edition*, page 313, 2011.

[138] Paul Richmond, Dawn Walker, Simon Coakley, and Daniela Romano. High performance cellular level agent-based simulation with flame for the gpu. *Briefings in bioinformatics*, 11(3):334–347, 2010.

[139] Kamel Rushaidat, Loren Schwiebert, Brock Jackman, Jason Mick, and Jeffrey Potoff. Evaluation of hybrid parallel cell list algorithms for monte carlo simulation. In *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on*, pages 1859–1864. IEEE, 2015.

[140] E. Rustico, G. Bilotta, A. Hérault, C. Del Negro, and G. Gallo. Advances in multi-gpu smoothed particle hydrodynamics simulations. *IEEE Transactions on Parallel and Distributed Systems*, 25(1):43–52, Jan 2014.

[141] Romelia Salomon-Ferrer, Andreas W Go tz, Duncan Poole, Scott Le Grand, and Ross C Walker. Routine microsecond molecular dynamics simulations with amber on gpus. 2. explicit solvent particle mesh ewald. *Journal of chemical theory and computation*, 9(9):3878–3888, 2013.

[142] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore gpus. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–10. IEEE, 2009.

[143] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D Nguyen, Victor W Lee, Daehyun Kim, and Pradeep Dubey. Fast sort on cpus, gpus and intel mic architectures. *Intel Labs*, pages 77–80, 2010.

[144] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. In *Proceedings of the 13th International Conference on Very Large Data Bases*, VLDB '87, pages 507–518, San Francisco, CA, USA, 1987. Morgan Kaufmann Publishers Inc.

[145] Wei Shao and Demetri Terzopoulos. Autonomous pedestrians. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 19–28. ACM, 2005.

[146] Maxim Shevtsov, Alexei Soupikov, and Alexander Kapustin. Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes. In *Computer Graphics Forum*, volume 26, pages 395–404. Wiley Online Library, 2007.

[147] Elias Stehle and Hans-Arno Jacobsen. A memory bandwidth-efficient hybrid radix sort on gpus. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 417–432. ACM, 2017.

[148] Avneesh Sud, Erik Andersen, Sean Curtis, Ming C Lin, and Dinesh Manocha. Real-time path planning in dynamic virtual environments using multiagent navigation graphs. *Visualization and Computer Graphics, IEEE Transactions on*, 14(3):526–538, 2008.

[149] Hongyu Sun, Yanshan Tian, Yulong Zhang, Jiong Wu, Sen Wang, Qiong Yang, and Qingguo Zhou. A special sorting method for neighbor search procedure in smoothed particle hydrodynamics on gpus. In *Parallel Processing Workshops (ICPPW), 44th International Conference on*, pages 81–85. IEEE, 2015.

[150] Shailesh Tamrakar. Performance optimization and statistical analysis of basic immune simulator (bis) using the flame gpu environment. 2015.

[151] Michael Bedford Taylor. Bitcoin and the age of bespoke silicon. In *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 1–10. IEEE, 2013.

[152] George Teodoro, Tahsin Kurc, Jun Kong, Lee Cooper, and Joel Saltz. Comparative performance analysis of intel xeon phi, gpu, and cpu. *arXiv preprint arXiv:1311.0378*, 2013.

[153] Seth Tisue and Uri Wilensky. Netlogo: A simple environment for modeling complexity. In *International conference on complex systems*, pages 16–21. Boston, MA, 2004.

[154] Yuri Torres, Arturo Gonzalez-Escribano, and Diego R Llanos. ubench: exposing the impact of cuda block geometry in terms of performance. *The Journal of Supercomputing*, 65(3):1150–1163, 2013.

[155] Adrien Treuille, Seth Cooper, and Zoran Popović. Continuum crowds. *ACM Transactions on Graphics (TOG)*, 25(3):1160–1168, 2006.

[156] Jacobus Antoon Van Meel, Axel Arnold, Daan Frenkel, SF Portegies Zwart, and Robert G Belleman. Harvesting graphics power for md simulations. *Molecular Simulation*, 34(3):259–266, 2008.

[157] Loup Verlet. Computer" experiments" on classical fluids. i. thermodynamical properties of lennard-jones molecules. *Physical review*, 159(1):98, 1967.

[158] Jeffrey Scott Vitter. Implementations for coalesced hashing. *Communications of the ACM*, 25(12):911–926, 1982.

[159] Vasily Volkov. Better performance at lower occupancy. In *GPU Technology Conference*, 2010. `http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf`.

[160] Vasily Volkov. Tutorial at sc11: Unrolling parallel loops. `http://www.cs.berkeley.edu/~volkov/volkov11-unrolling.pdf`, 11 2011.

[161] Vasily Volkov and James W Demmel. Benchmarking gpus to tune dense linear algebra. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–11. IEEE, 2008.

[162] John Von Neumann. The general and logical theory of automata. *Cerebral mechanisms in behavior*, pages 1–41, 1951.

[163] DC Walker, J Southgate, G Hill, M Holcombe, DR Hose, SM Wood, S Mac Neil, and RH Smallwood. The epitheliome: agent-based modelling of the social behaviour of cells. *Biosystems*, 76(1-3):89–100, 2004.

[164] Chao Wang, Lei Gong, Qi Yu, Xi Li, Yuan Xie, and Xuehai Zhou. Dlau: A scalable deep learning accelerator unit on fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(3):513–517, 2016.

[165] Daniel Winkler, Massoud Rezavand, and Wolfgang Rauch. Neighbour lists for smoothed particle hydrodynamics on gpus. *Computer Physics Communications*, 225:140–148, 2018.

[166] Mingyu Yang, Philipp Andelfinger, Wentong Cai, and Alois Knoll. Evaluation of conflict resolution methods for agent-based simulations on the gpu. In *Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '18, pages 129–132, New York, NY, USA, 2018. ACM.

[167] Tao Yang, Ralph R Martin, Ming C Lin, Jian Chang, and Shi-Min Hu. Pairwise force sph model for real-time multi-interaction applications. *IEEE transactions on visualization and computer graphics*, 23(10):2235–2247, 2017.

[168] Xiaoping Zheng, Tingkuan Zhong, and Mengting Liu. Modeling crowd evacuation of a building based on seven methodological approaches. *Building and Environment*, 44(3):437–445, 2009.